Parallelism and the Memory Hierarchy

Todd C. Mowry & Dave Eckhardt

I. Impact of Parallelism on Memory Protection

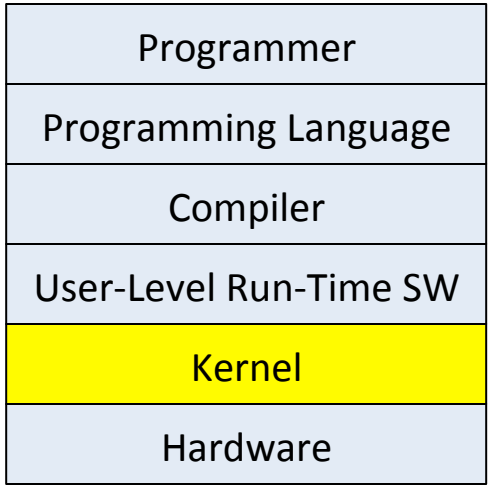
II. Cache Coherence

III. Scheduling Revisited

Impact of Parallel Processing on the Kernel (vs. Other Layers)

- Kernel itself becomes a parallel program
  - avoid bottlenecks when accessing data structures
    - lock contention, communication, load balancing, etc.
  - use all of the standard parallel programming tricks
- Thread scheduling gets more complicated
  - programmers usually assume:
    - all threads running *simultaneously*
      - load balancing, avoiding synchronization problems
    - threads *don't move* between processors
      - for optimizing communication and cache locality
- Primitives for naming, communicating, and coordinating need to be *fast*
  - Shared Address Space: virtual memory management across threads
  - Message Passing: low-latency send/receive primitives

System Layers



Case Study: Protection

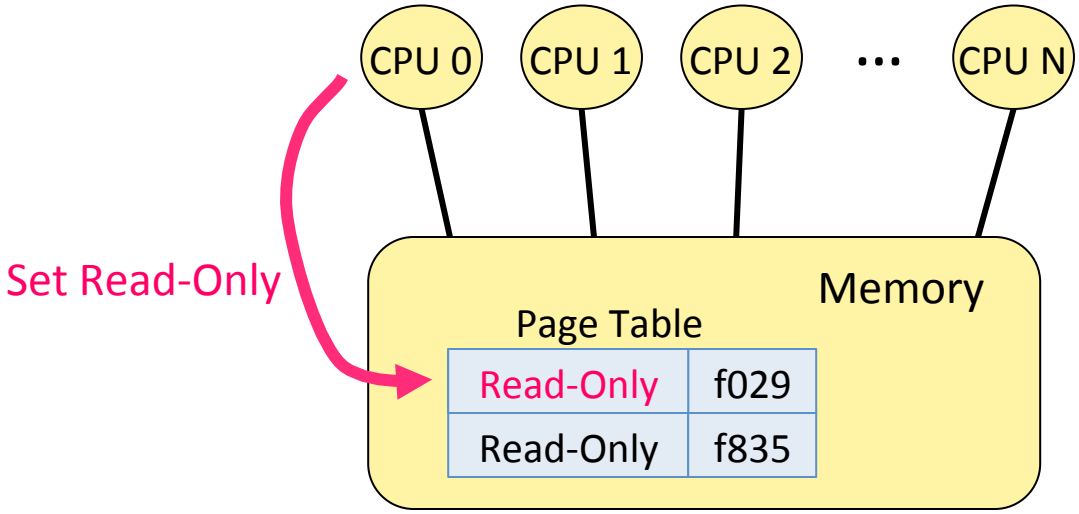
- One important role of the OS:
  - provide *protection* so that buggy processes don't corrupt other processes

- Shared Address Space:
  - access permissions for virtual memory pages
    - e.g., set pages to read-only during copy-on-write optimization

- Message Passing:
  - ensure that target thread of send() message is a valid recipient

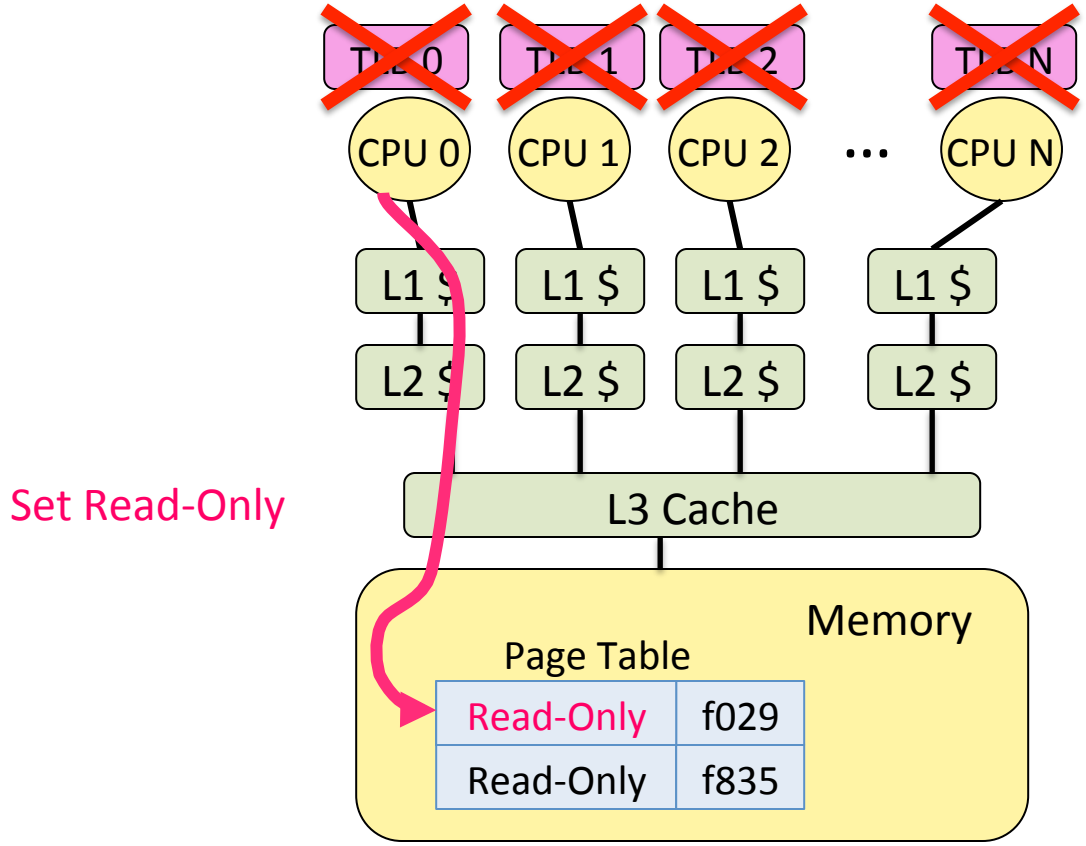
How Do We Propagate Changes to Access Permissions?

- What if parallel machines (and VM management) simply looked like this:



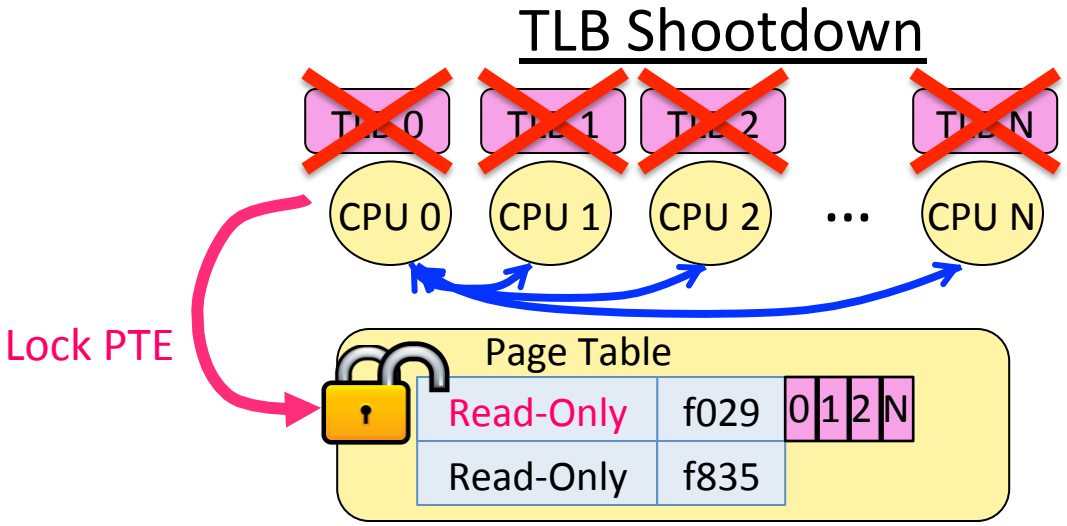
- Updates to the page tables (in shared memory) could be read by other threads
- But this would be a very slow machine!
  - Why?

VM Management in a Multicore Processor



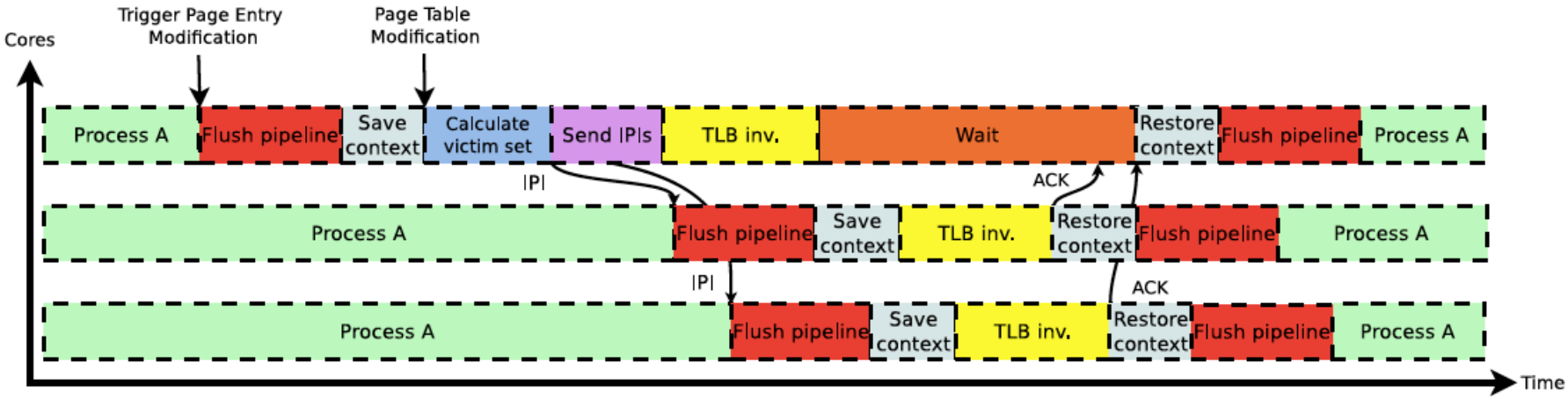
"TLB Shootdown":

- relevant entries in the TLBs of other processor cores need to be flushed



1. Initiating core triggers OS to lock the corresponding Page Table Entry (PTE)
2. OS generates a list of cores that may be using this PTE (erring conservatively)
3. Initiating core sends an Inter-Processor Interrupt (IPI) to those other cores
   - requesting that they invalidate their corresponding TLB entries
4. Initiating core invalidates local TLB entry; waits for acknowledgements
5. Other cores receive interrupts, execute interrupt handler which invalidates TLBs
   - send an acknowledgement back to the initiating core
6. Once initiating core receives all acknowledgements, it unlocks the PTE

TLB Shootdown Timeline



- Image from: Villavieja *et al*, "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory." In *Proceedings of PACT 2011*.

Performance of TLB Shootdown

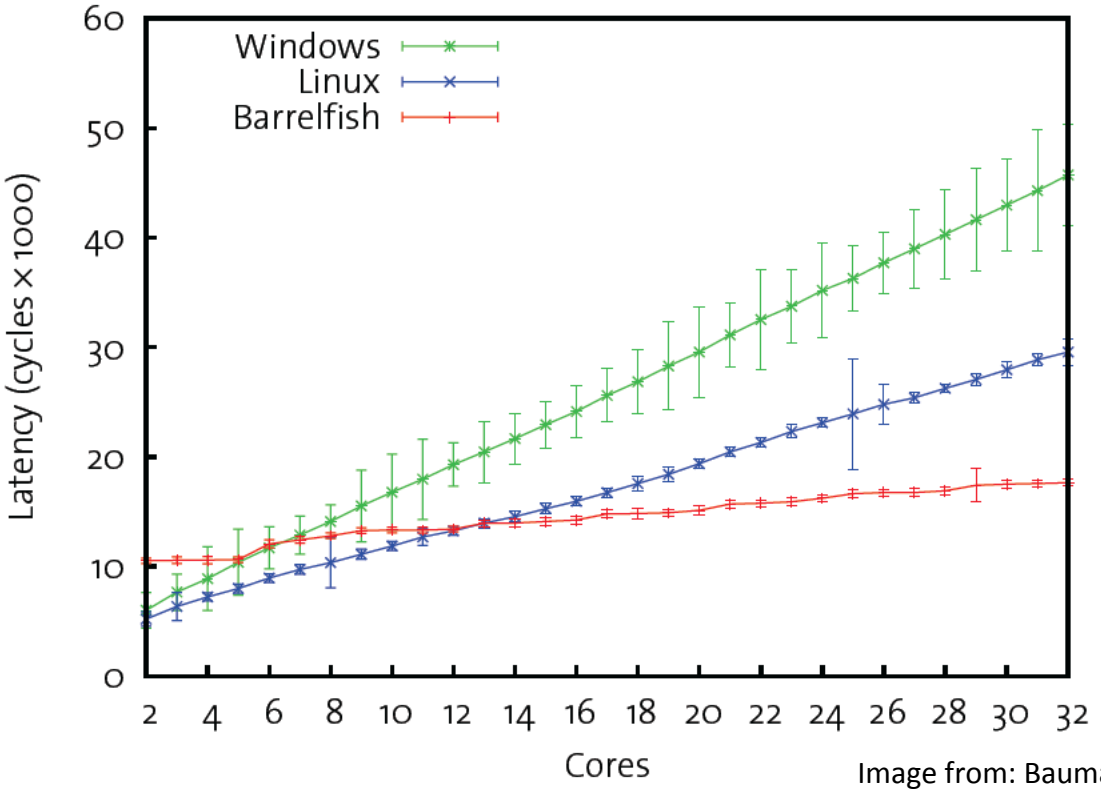


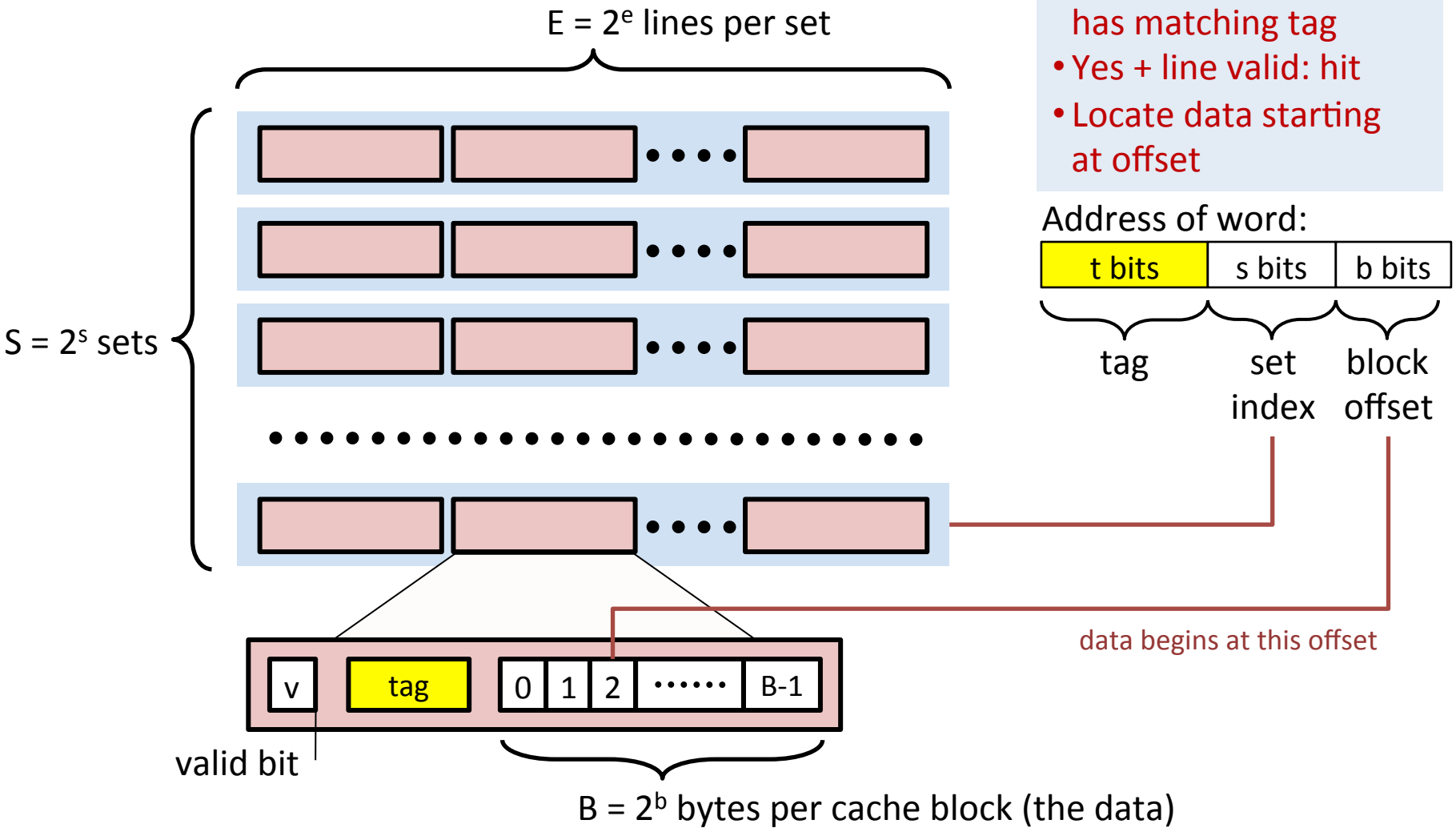
Image from: Baumann *et al*,"The Multikernel: a New OS Architecture for Scalable Multicore Systems." In Proceedings of SOSP '09.

- Expensive operation
  - e.g., over 10,000 cycles on 8 or more cores
- Gets more expensive with increasing numbers of cores

Caches in a Single-Processor Machine (Review from 213)

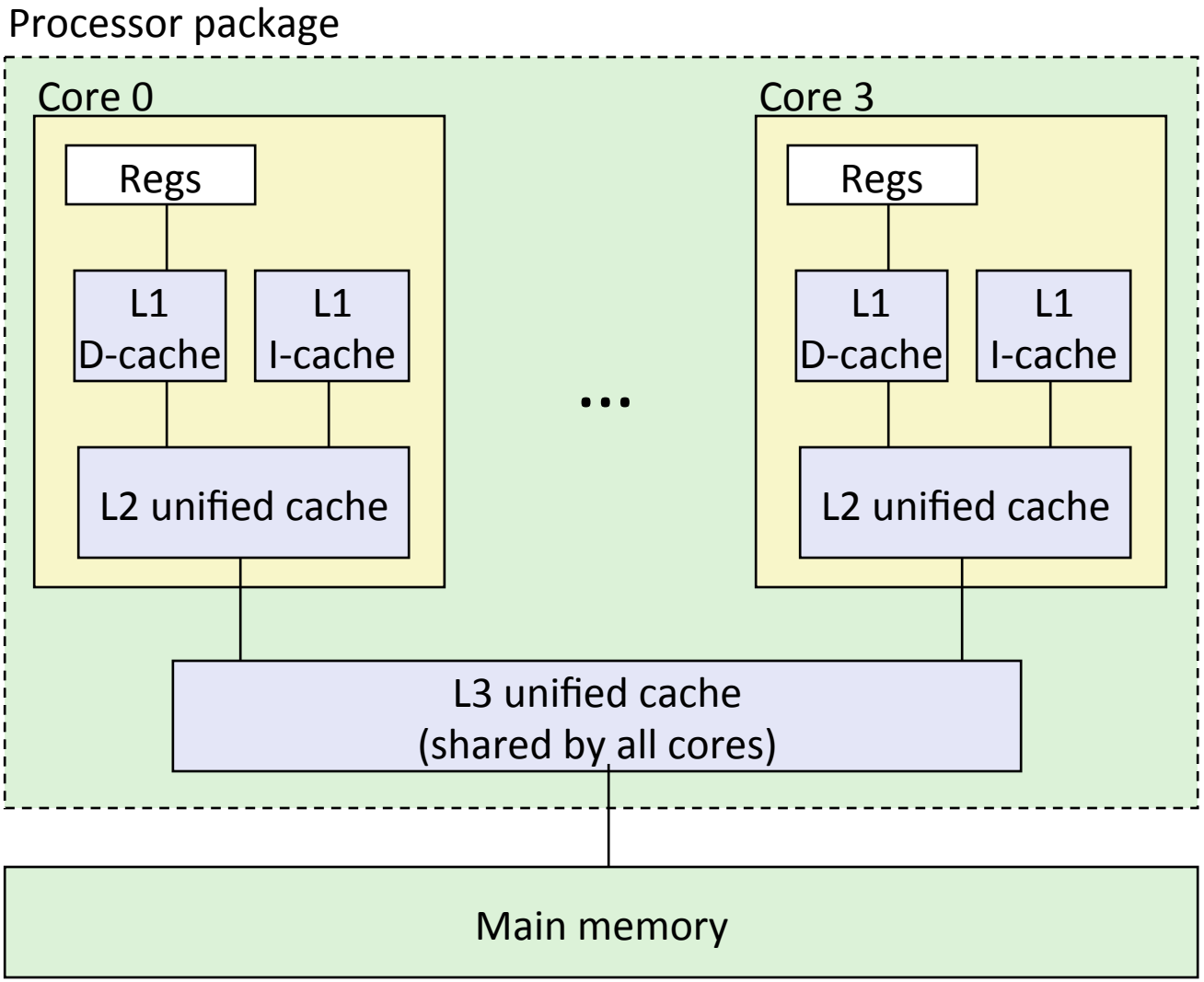
- Ideally, memory would be arbitrarily fast, large, and cheap
  - unfortunately, you can't have all three (e.g., fast → small, large → slow)
  - cache hierarchies are a hybrid approach
    - if all goes well, they will behave as though they are both fast and large
- Cache hierarchies work due to *locality*
  - temporal locality → even relatively small caches may have high hit rates
  - spatial locality → move data in *blocks* (e.g., 64 bytes)
- Locating the data:
  - Main memory: directly addressed
    - we know exactly where to look:
      - at the unique location corresponding to the address
  - Cache: may or may not be somewhere in the cache
    - need *tags* to identify the data
    - may need to check multiple locations, depending on the degree of associativity

Cache Read



- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Intel Quad Core i7 Cache Hierarchy



L1 I-cache and D-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

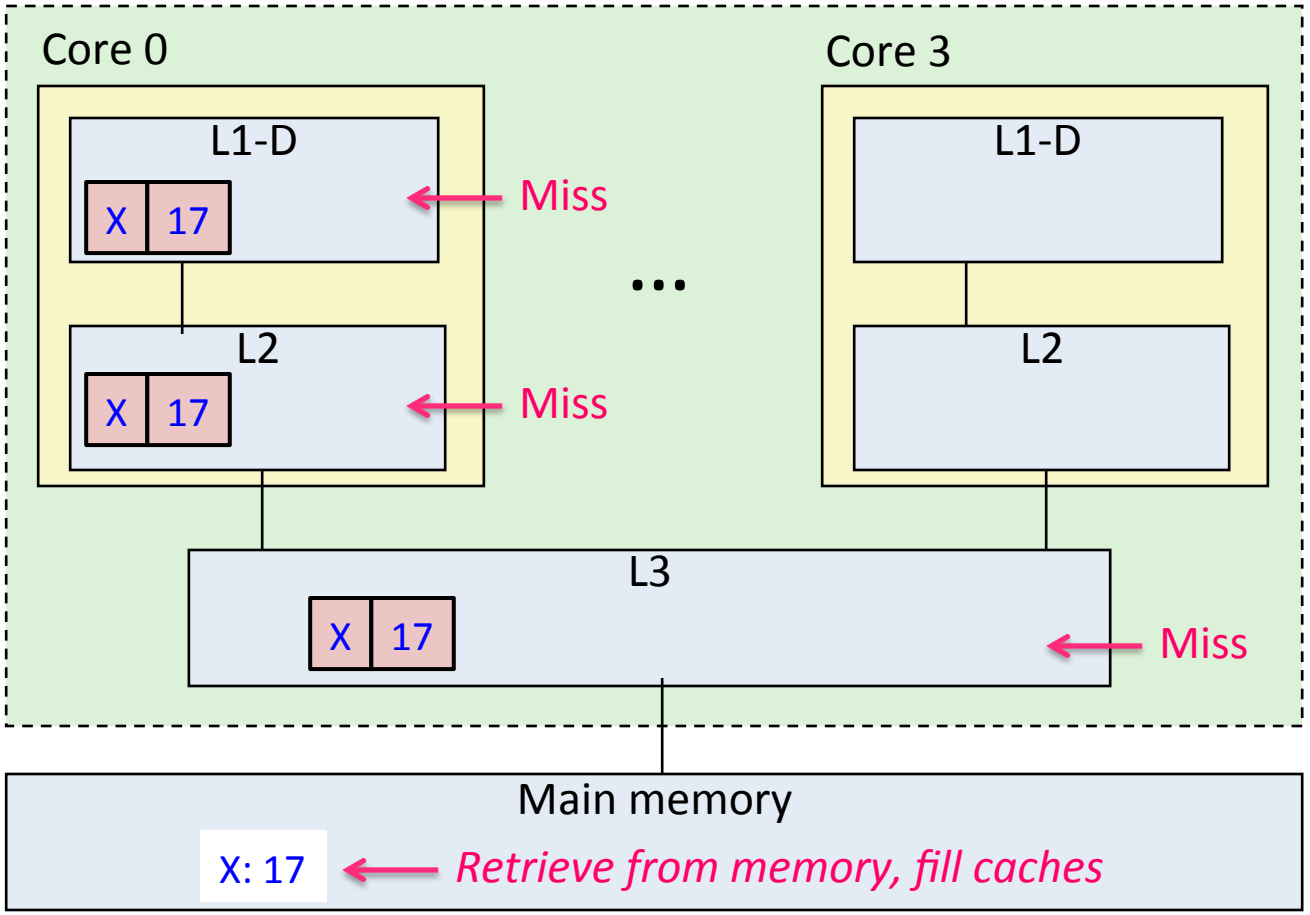
L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

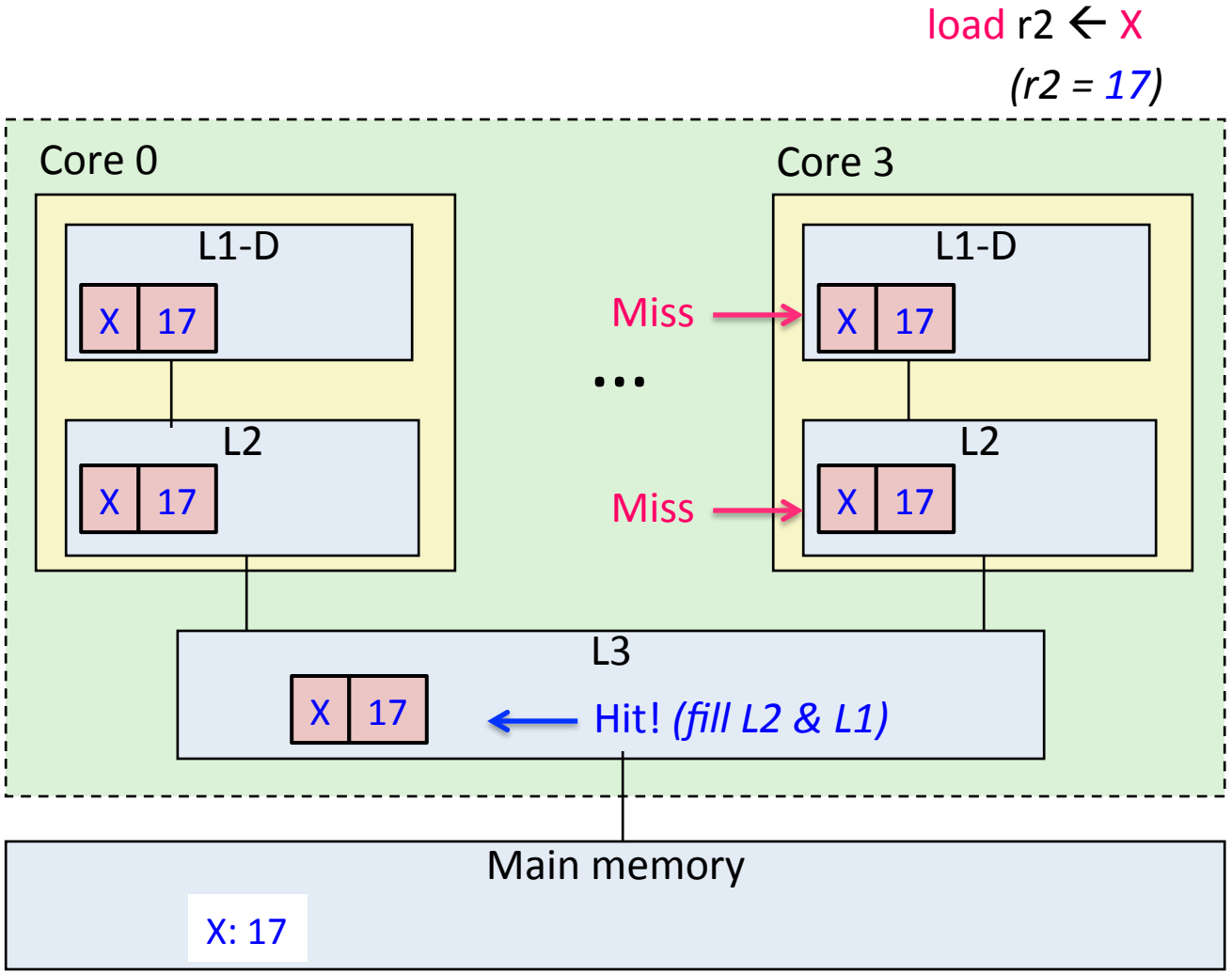
Simple Multicore Example: Core 0 Loads X

load r1 ← X

(r1 = 17)



Example Continued: Core 3 Also Does a Load of X

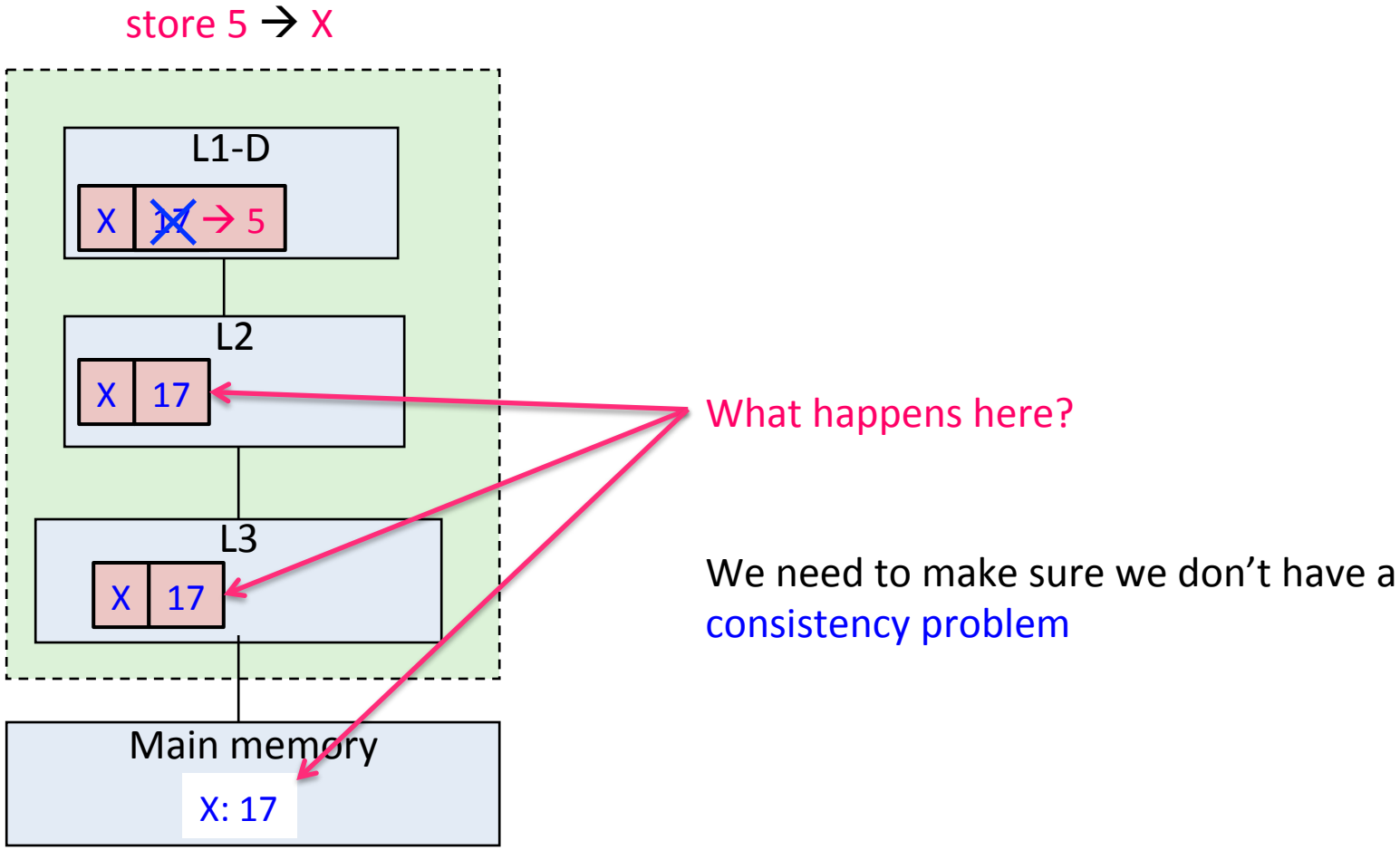


Example of constructive sharing:

- Core 3 benefited from Core 0 bringing X into the L3 cache

Fairly straightforward with only loads. But what happens when stores occur?

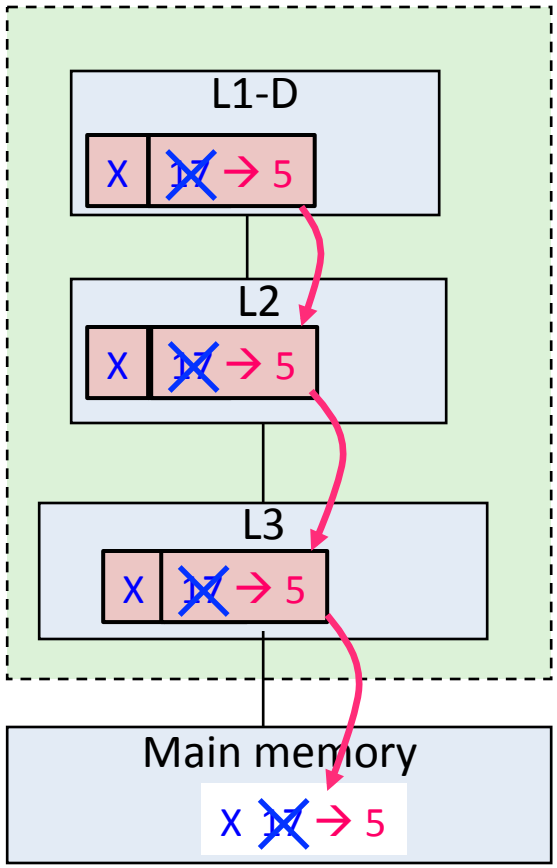
Review: How Are Stores Handled in Uniprocessor Caches?



Options for Handling Stores in Uniprocessor Caches

Option 1: Write-Through Caches
→ propagate immediately

store 5 → X

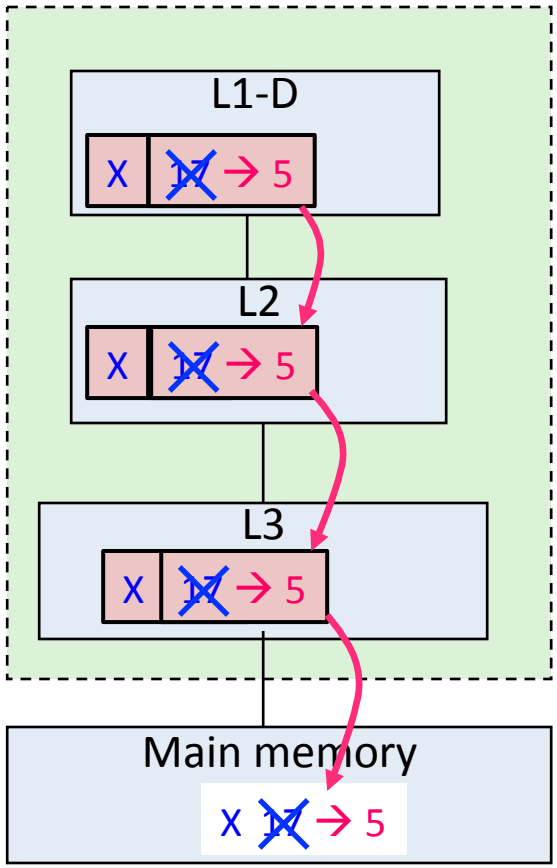


What are the advantages and disadvantages of this approach?

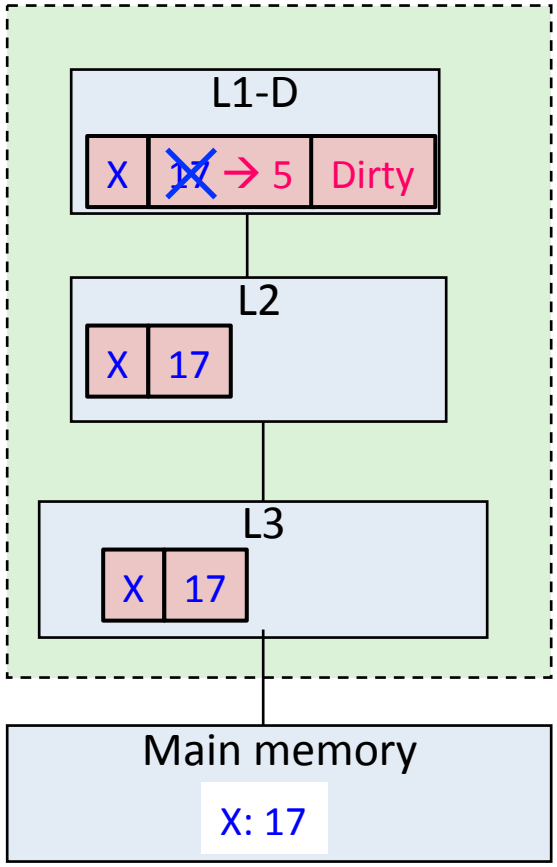
Options for Handling Stores in Uniprocessor Caches

Option 1: Write-Through Caches
- propagate immediately

store 5 → X



Option 2: Write-Back Caches
- defer propagation until eviction
- keep track of *dirty* status in tags



Upon eviction, if data is dirty, write it back.

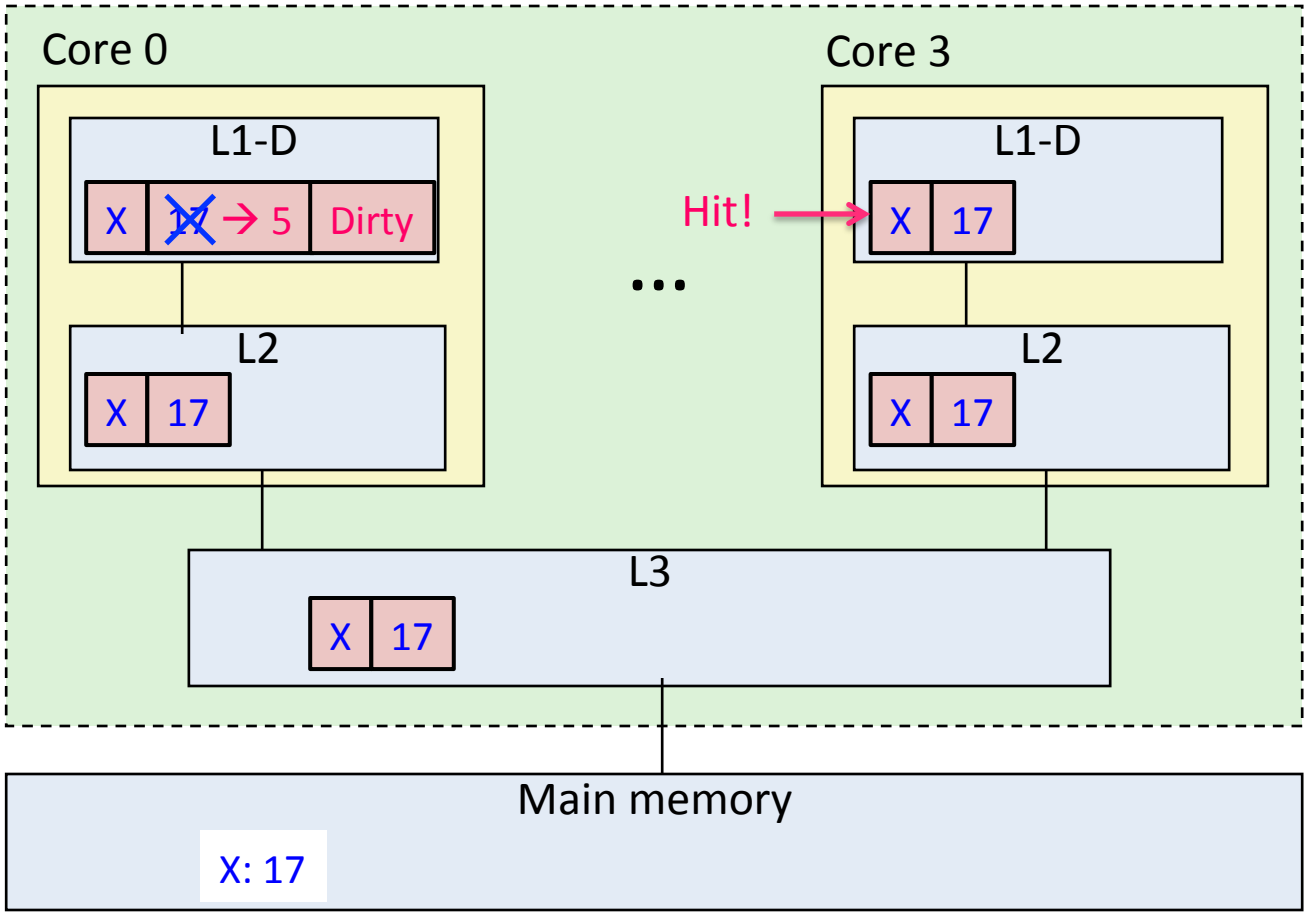
Write-back is more commonly used in practice (due to bandwidth limitations)

Resuming Multicore Example

1. store 5 → X

2. load r2 ← X

(r2 = 17) Hmm…



What is supposed to happen in this case?

Is it incorrect behavior for r2 to equal 17?

- if not, when would it be?

(Note: core-to-core communication often takes tens of cycles.)

What is Correct Behavior for a Parallel Memory Hierarchy?

- Note: side-effects of writes are only observable when *reads* occur
  - so we will focus on the values returned by reads

- Intuitive answer:
  - reading a location should return the latest value written (by any thread)

- Hmm… what does "latest" mean exactly?
  - within a thread, it can be defined by program order
  - but what about across threads?
    - the most recent write in physical time?
      - hopefully not, because there is no way that the hardware can pull that off
        - » e.g., if it takes >10 cycles to communicate between processors, there is no way that processor 0 can know what processor 1 did 2 clock ticks ago
    - most recent based upon something else?
      - Hmm…

Refining Our Intuition

Thread 0

```
// write evens to X
for (i=0; i<N; i+=2) {
   X = i;
   …
}
```

Thread 1

```
// write odds to X
for (j=1; j<N; j+=2) {
   X = j;
   …
}
```

Thread 2

```
…
A = X;
…
B = X;
…
C = X;
…
```

(Assume: X=0 initially, and these are the only writes to X.)

- What would be some clearly illegal combinations of (A,B,C)?
- How about:

  (4,8,1)? (9,12,3)? (7,19,31)?
- What can we generalize from this?
  - writes from any particular thread must be consistent with program order
    - in this example, observed even numbers must be increasing (ditto for odds)
  - across threads: writes must be consistent with *a valid interleaving of threads*
    - not physical time! (programmer cannot rely upon that)

Visualizing Our Intuition

Thread 0

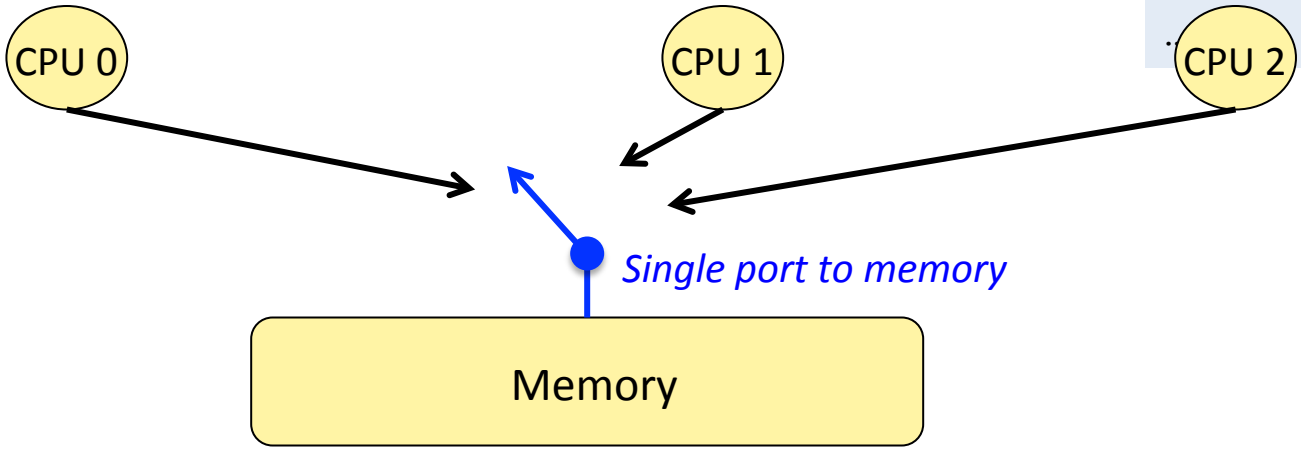
```
// write evens to X
for (i=0; i<N; i+=2) {
   X = i;
   …
}
```

Thread 1

```
// write odds to X
for (j=1; j<N; j+=2) {
   X = j;
   …
}
```

Thread 2

```
…
A = X;
…
B = X;
…
C = X;
…
```



- Each thread proceeds in program order
- Memory accesses interleaved (one at a time) to a single-ported memory
  - rate of progress of each thread is unpredictable

Correctness Revisited

Thread 0

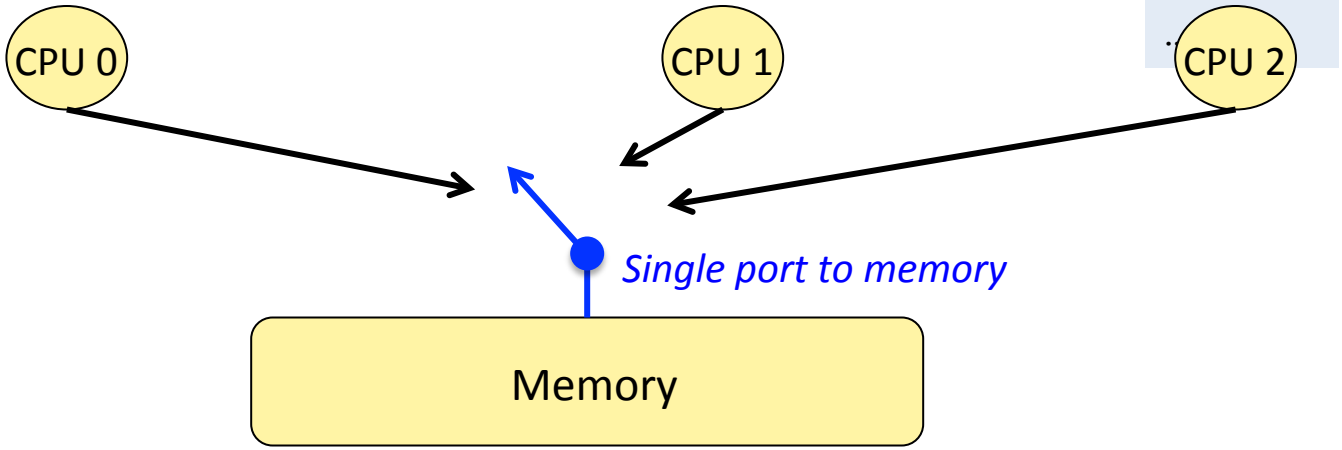
```
// write evens to X
for (i=0; i<N; i+=2) {
   X = i;
   …
}
```

Thread 1

```
// write odds to X
for (j=1; j<N; j+=2) {
   X = j;
   …
}
```

Thread 2

```
…
A = X;
…
B = X;
…
C = X;
…
```



Recall: "reading a location should return the latest value written (by any thread)"

→ "latest" means consistent with some interleaving that matches this model

– this is a hypothetical interleaving; the machine didn't necessary do this!

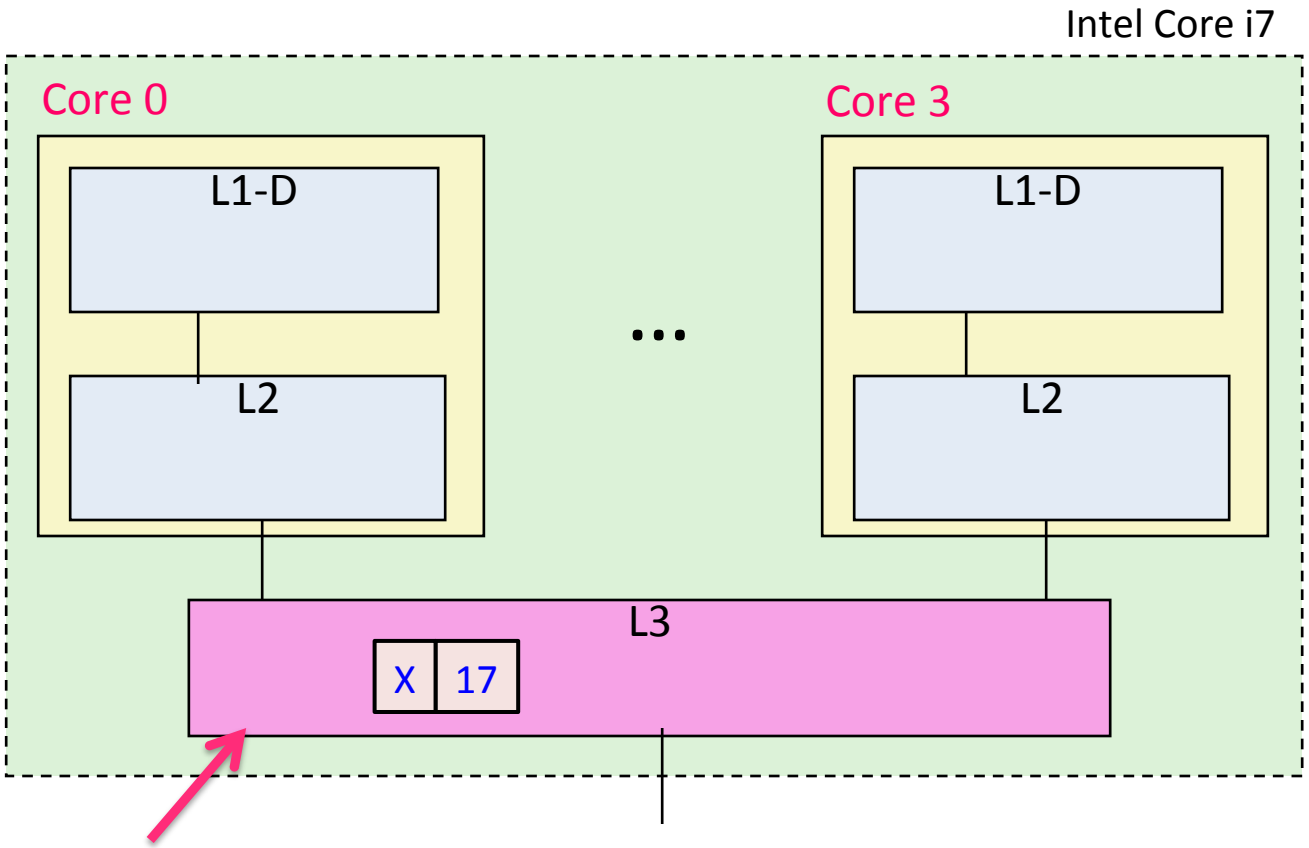
Two Parts to Memory Hierarchy Correctness

1. "Cache Coherence"
   - do all loads and stores to a given cache block behave correctly?
     - i.e. are they consistent with our interleaving intuition?
     - important: separate cache blocks have independent, unrelated interleavings!

2. "Memory Consistency Model"
   - do all loads and stores, even to separate cache blocks, behave correctly?
     - builds on top of cache coherence
     - especially important for synchronization, causality assumptions, etc.

Cache Coherence (Easy Case)

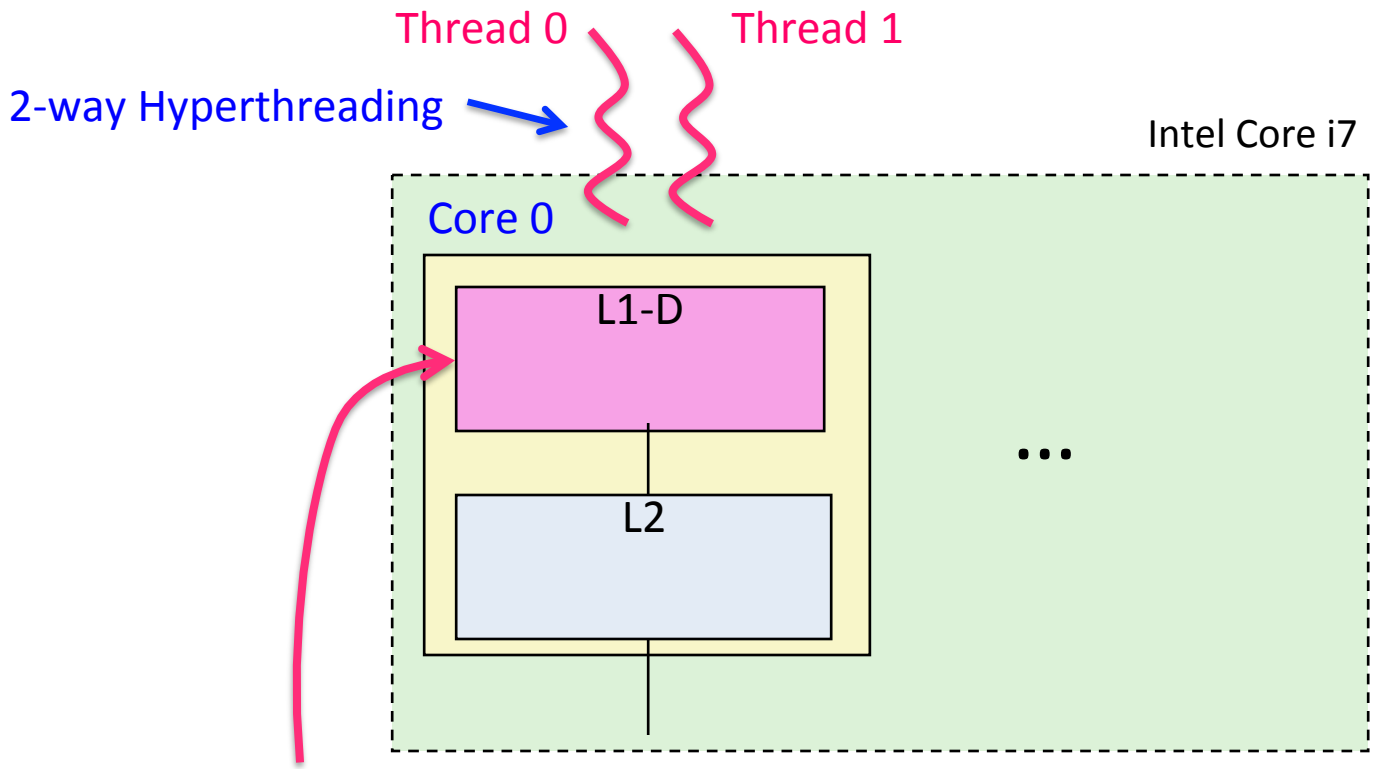
- One easy case: a physically shared cache



L3 cache is physically shared by on-chip cores

Cache Coherence (Easy Case)

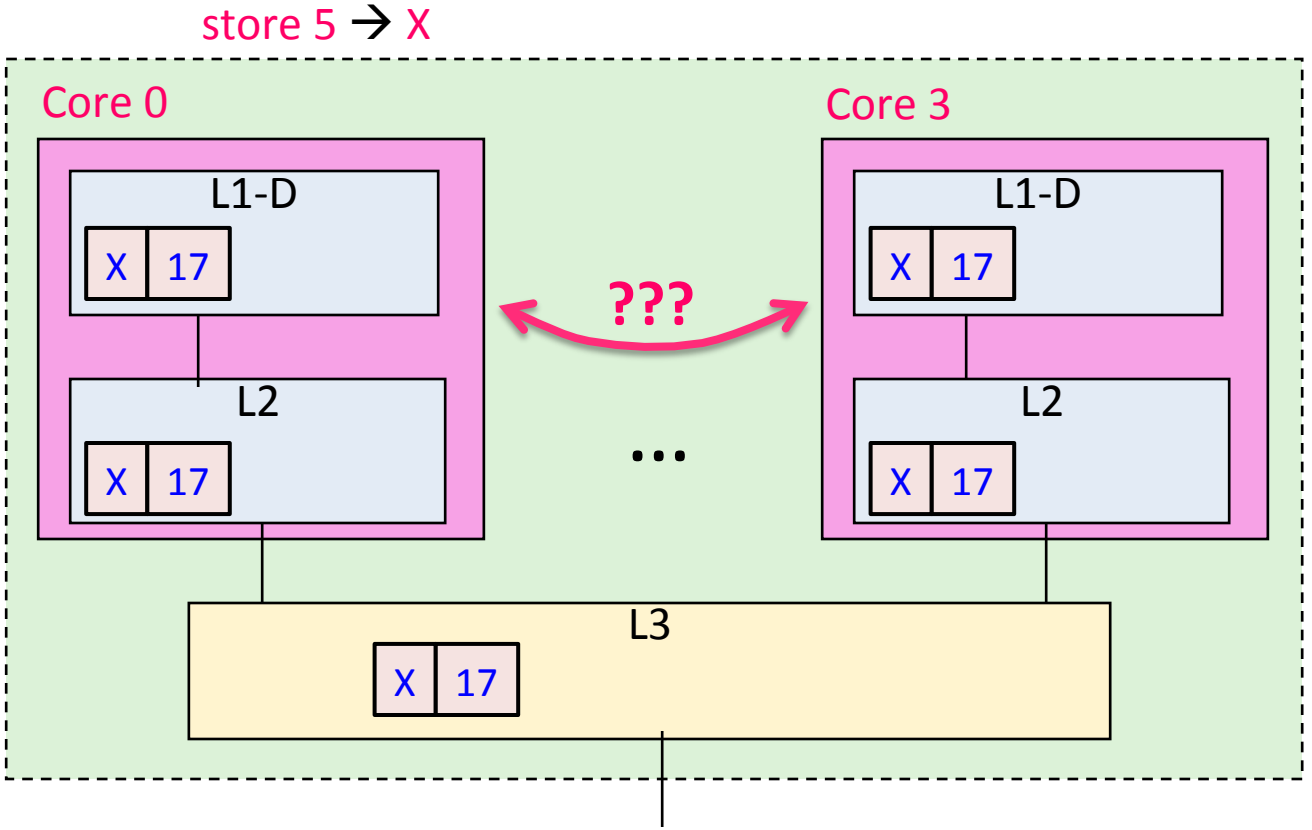
- One easy case: a physically shared cache



L1 cache (plus L2, L3) is physically shared by Thread 0 and Thread 1

Cache Coherence: Beyond the Easy Case

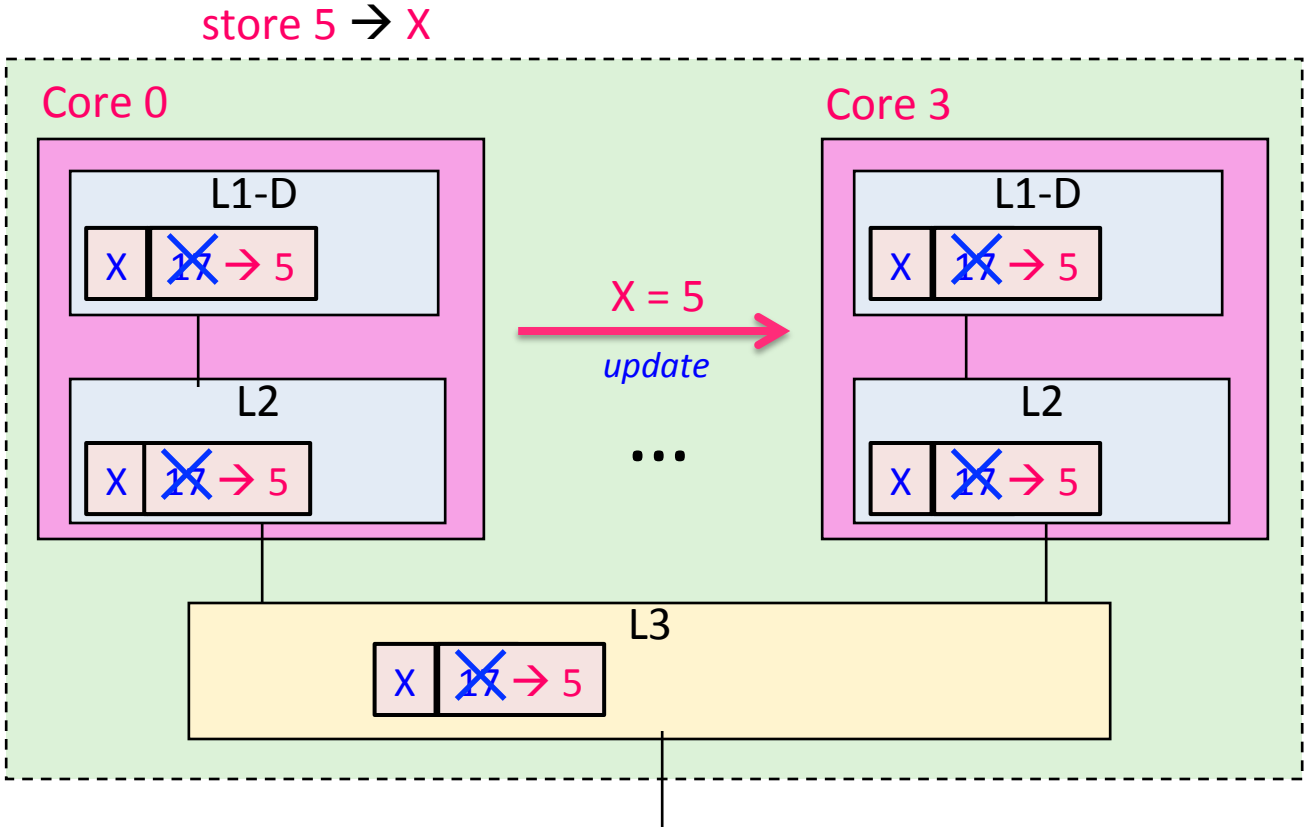
- How do we implement L1 & L2 cache coherence between the cores?



- Common approaches: update or invalidate protocols

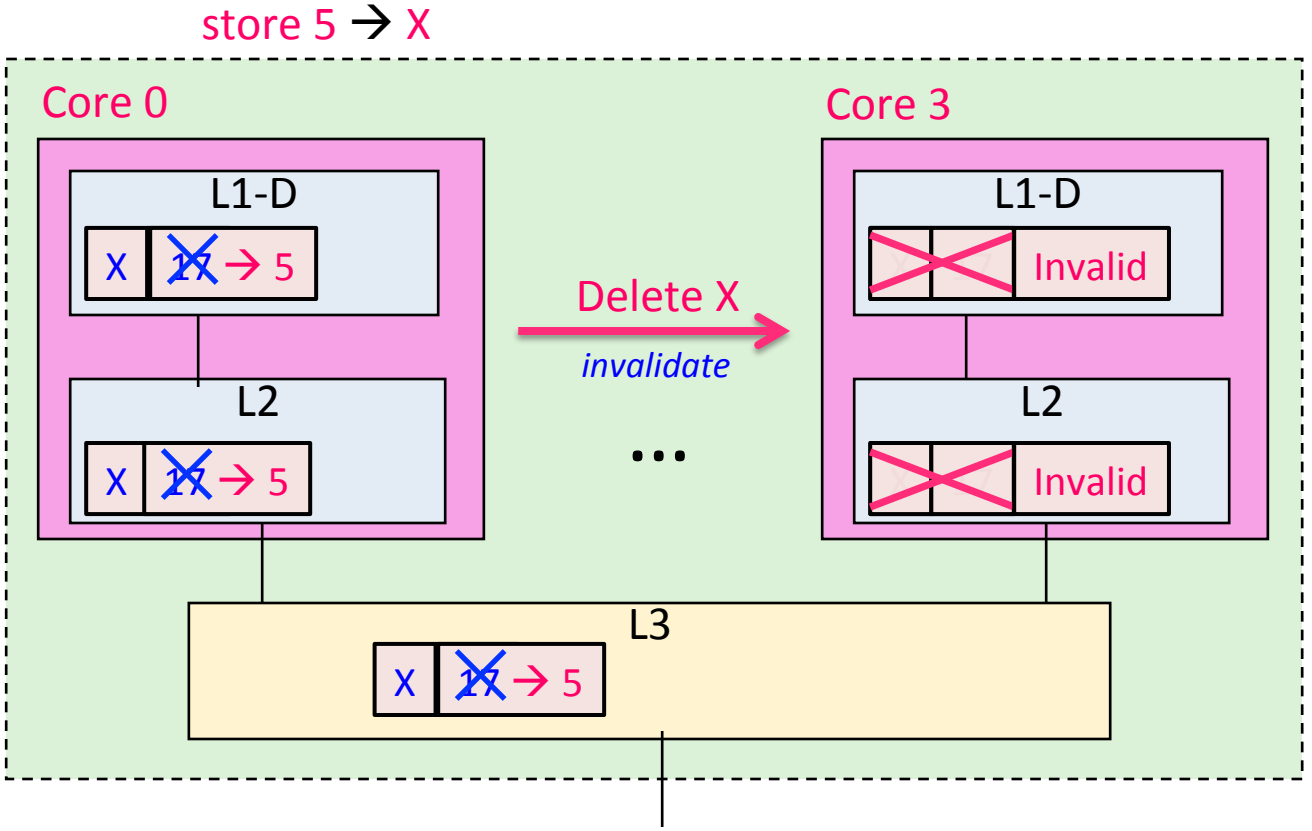
One Approach: Update Protocol

- Basic idea: upon a write, propagate new value to shared copies in peer caches



Another Approach: Invalidate Protocol

- Basic idea: upon a write, delete any shared copies in peer caches



Update vs. Invalidate

- When is one approach better than the other?
  - *(hint: the answer depends upon program behavior)*

- Key question:
  - Is a block written by one processor read by others before it is rewritten?

  - if so, then update may win:
    - readers that already had copies will not suffer cache misses
  - if not, then invalidate wins:
    - avoids useless updates (including to dead copies)

- Which one is used in practice?
  - invalidate (due to hardware complexities of update)
    - although some machines have supported both (configurable per-page by the OS)

How Invalidation-Based Cache Coherence Works (Short Version)

- Cache tags contain additional coherence state ("MESI" example below):
  - Invalid:
    - nothing here (often as the result of receiving an invalidation message)
  - Shared (Clean):
    - matches the value in memory; other processors may have shared copies also
    - I can read this, but cannot write it until I get an exclusive/modified copy
  - Exclusive (Clean):
    - matches the value in memory; I have the only copy
    - I can read or write this (a write causes a transition to the Modified state)
  - Modified (aka Dirty):
    - has been modified, and does not match memory; I have the only copy
    - I can read or write this; I must supply the block if another processor wants to read

- The hardware keeps track of this automatically
  - using either broadcast (if interconnect is a bus) or a directory of sharers

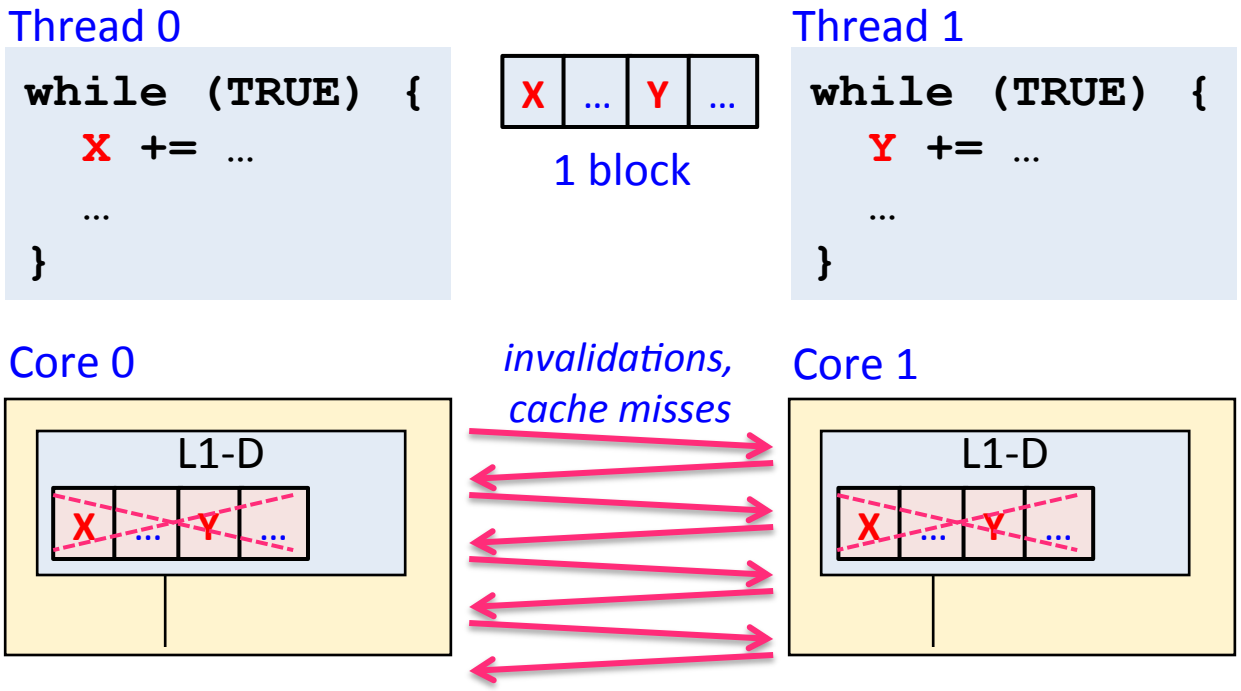
Performance Impact of Invalidation-Based Cache Coherence

- Invalidations result in a new source of cache misses!

- Recall that uniprocessor cache misses can be categorized as:
  - (i) cold/compulsory misses, (ii) capacity misses, (iii) conflict misses

- Due to the sharing of data, parallel machines also have misses due to:
  - (iv) true sharing misses
    - e.g., Core A reads X → Core B writes X → Core A reads X again (cache miss)
      - nothing surprising here; this is true communication
  - (v) false sharing misses
    - e.g., Core A reads X → Core B writes **Y** → Core A reads X again (cache miss)
      - What???
      - where X and Y unfortunately fell within the same cache block

Beware of False Sharing!



- It can result in a devastating ping-pong effect with a very high miss rate
  - plus wasted communication bandwidth
- Pay attention to data layout:
  - the threads above appear to be working independently, but they are not

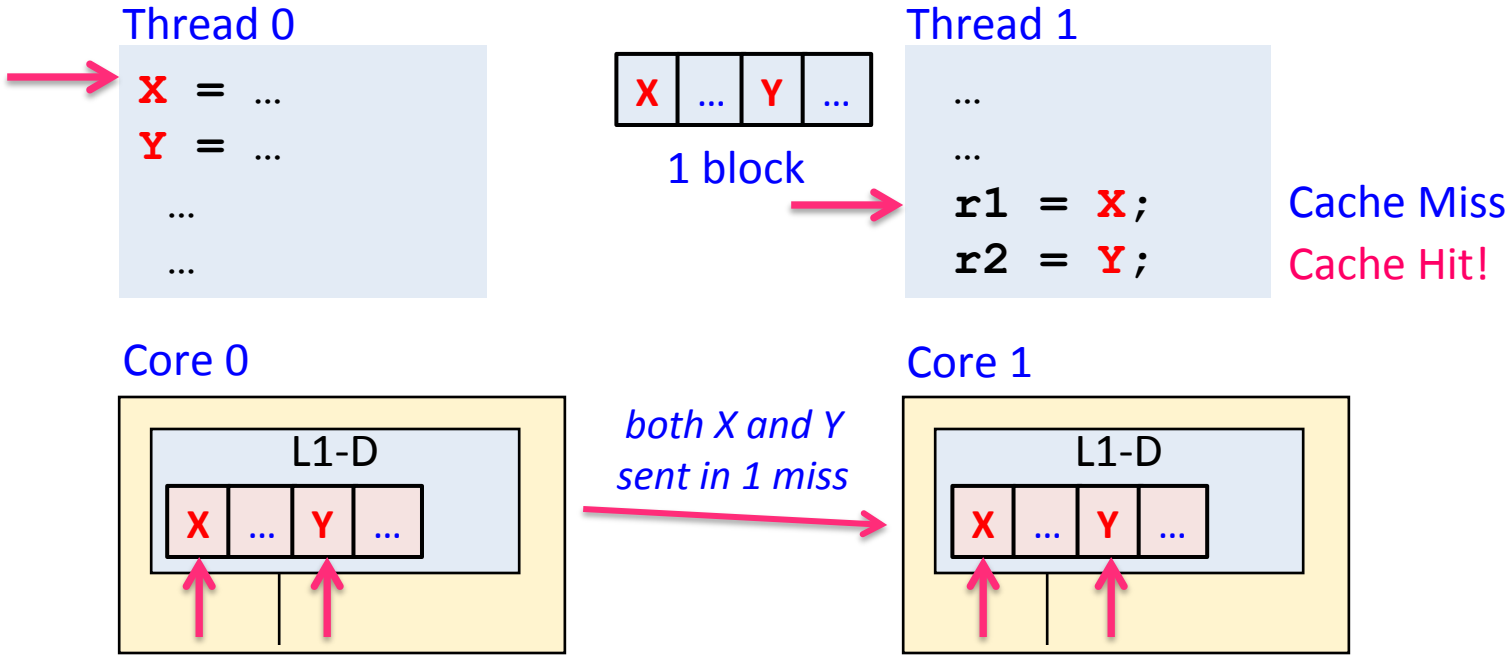
How False Sharing Might Occur in the OS

- Operating systems contain lots of counters (to count various types of events)
  - these counters are frequently updated (e.g., `event1_count++`)
  - but infrequently read
    - e.g., when an application makes a system call to read a particular count
- Simplest implementation: a centralized counter
  - this is the obvious thing to do in a non-parallel OS
  - but it performs very poorly on a parallel machine. Why?

    → lock contention when updating counter
- "Improved" implementation: an array of counters (indexed by processor ID)
  - now a read operation sums up all of the counters in the array
  - effectively eliminates lock contention
  - but performance may still be very poor. Why?

    → false sharing!
- Better solution: padded array of counters (one cache block per processor)
  - any downsides to this?

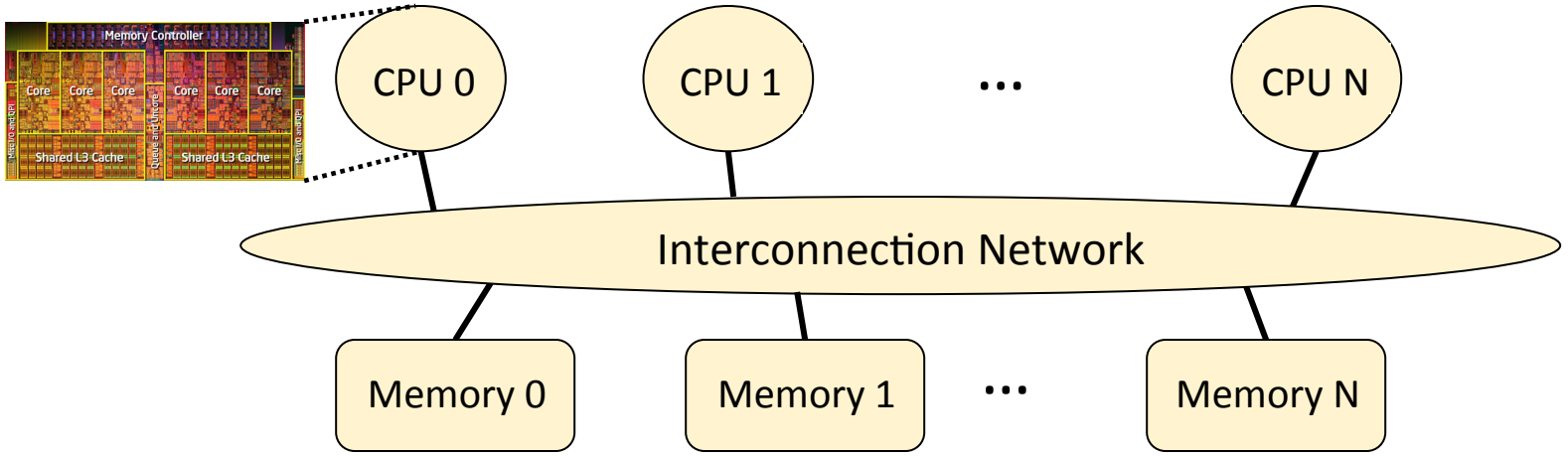
True Sharing Can Benefit from Spatial Locality



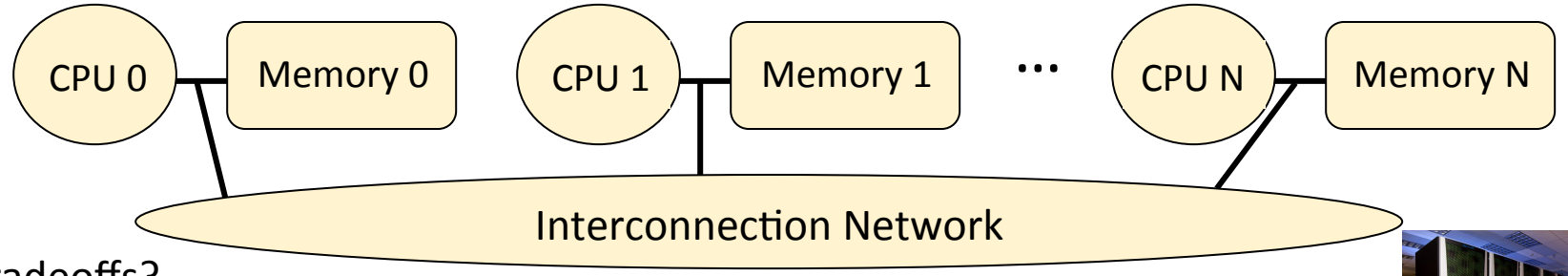
- With true sharing, spatial locality can result in a prefetching benefit
- Hence data layout can help or harm sharing-related misses in parallel software

Options for Building Even Larger Shared Address Machines

Symmetric Multiprocessor (SMP):



Non-Uniform Memory Access (NUMA):



- Tradeoffs?
- NUMA is more commonly used at large scales (e.g., Blacklight at the PSC)

Summary

- Case study: memory protection on a parallel machine
  - TLB shootdown
    - involves Inter-Processor Interrupts to flush TLBs
- Part 1 of Memory Correctness: Cache Coherence
  - reading "latest" value does not correspond to physical time!
    - corresponds to latest in hypothetical interleaving of accesses
  - new sources of cache misses due to invalidations:
    - true sharing misses
    - false sharing misses

- Looking ahead: Scheduling Revisited, Part 2 of Memory Correctness