Advanced Computer Architecture

Chapter 10 – Multicore, parallel, and cache coherency

Part 2:

Cache coherency protocols – "snooping"

November 2025

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd, 4th and 5th eds),* and on the lecture slides of David Patterson, John Kubiatowicz and Yujia Jin at Berkeley

Hennessy and Patterson 6th ed: Section 5.2, pp377

What you should get from this

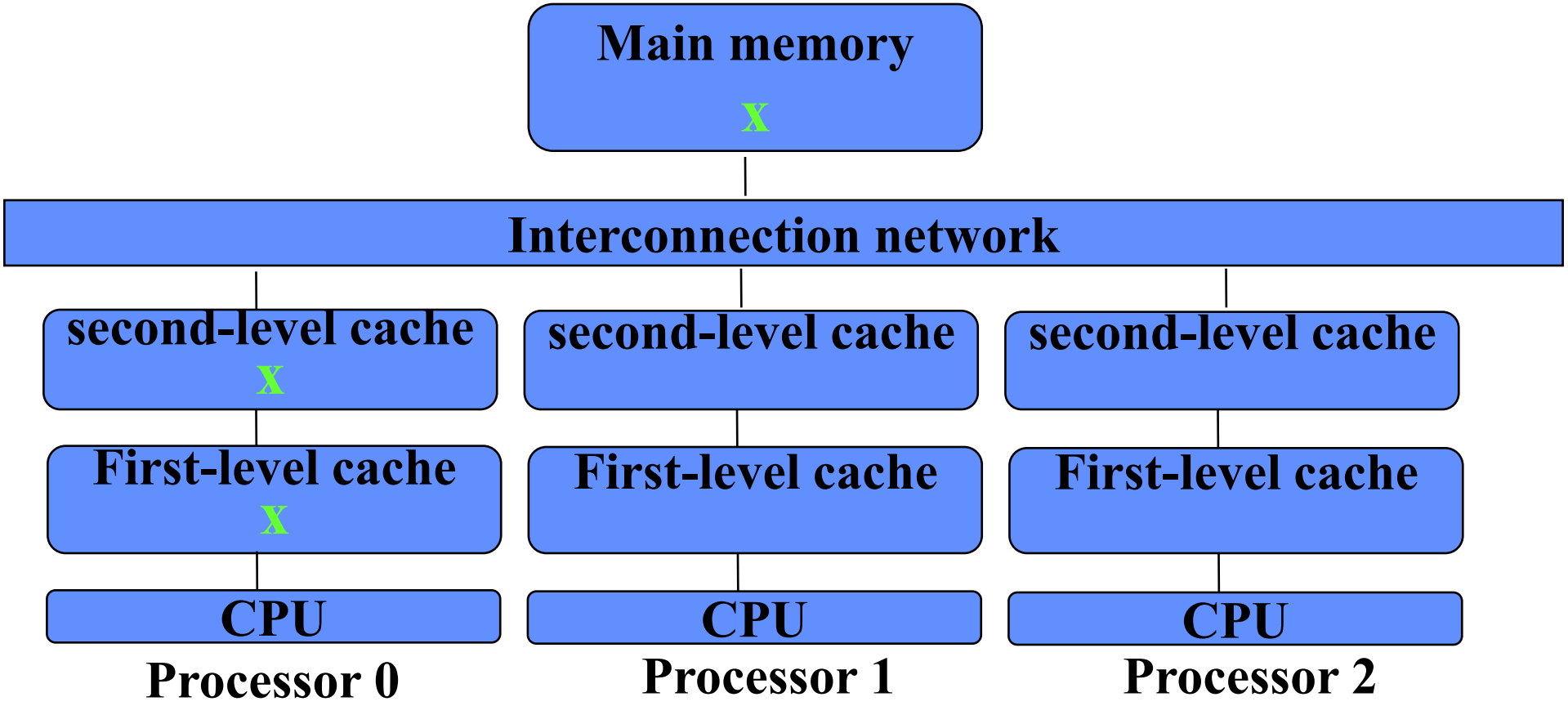
Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

- Why power considerations motivate multicore
- Why is shared-memory parallel programming attractive?
  - How is dynamic load-balancing implemented?
  - Why is distributed-memory parallel programming harder but more likely to yield robust performance?

Part 2

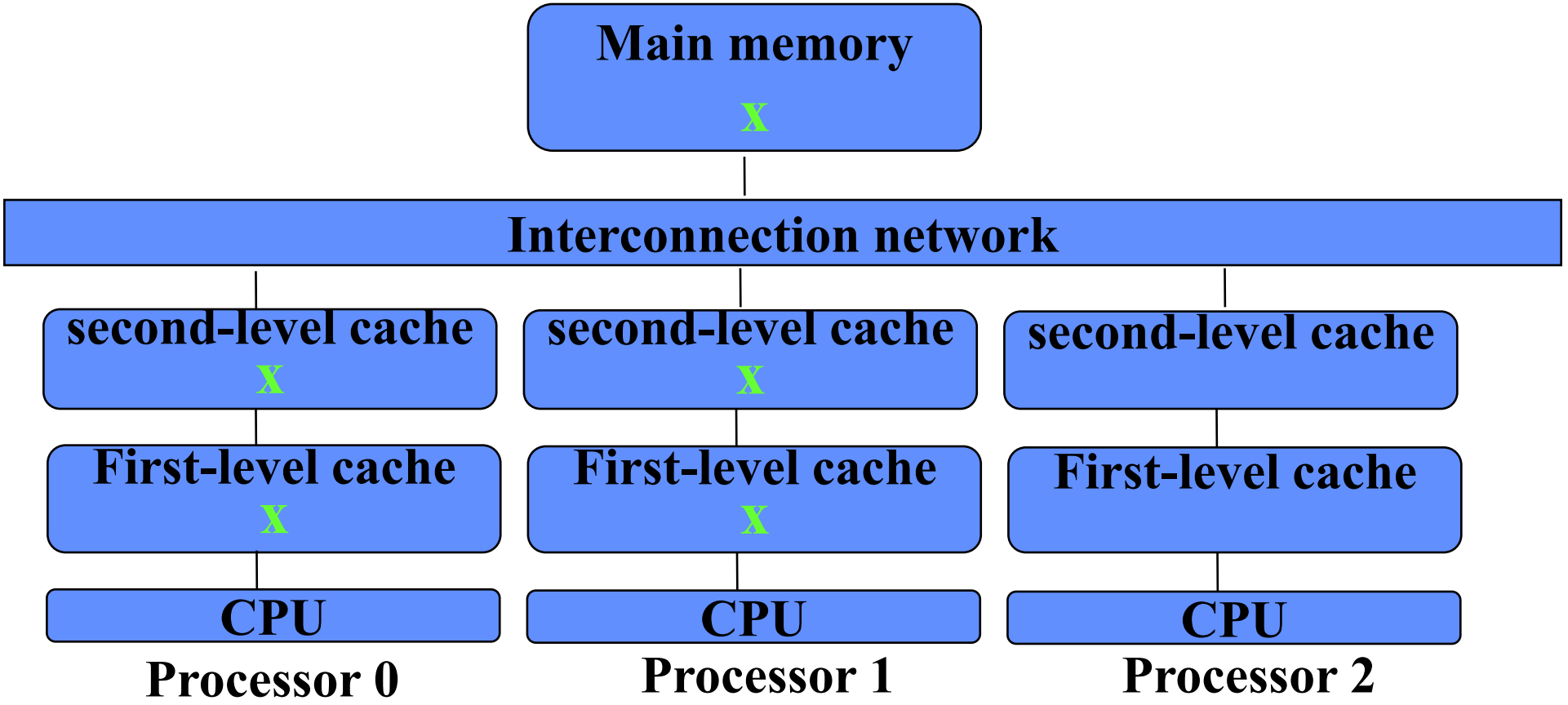
- What is the cache coherency problem
  - There is a design-space of “snooping” protocols based on broadcasting invalidations and requests
- How are atomic operations and locks implemented?
  - Eg load-linked, store conditional
- What is sequential consistency?
  - Why might you prefer a memory model with weaker consistency?
- For larger systems, some kind of “directory” is needed to avoid/reduce the broadcasting

Implementing shared memory: multiple caches



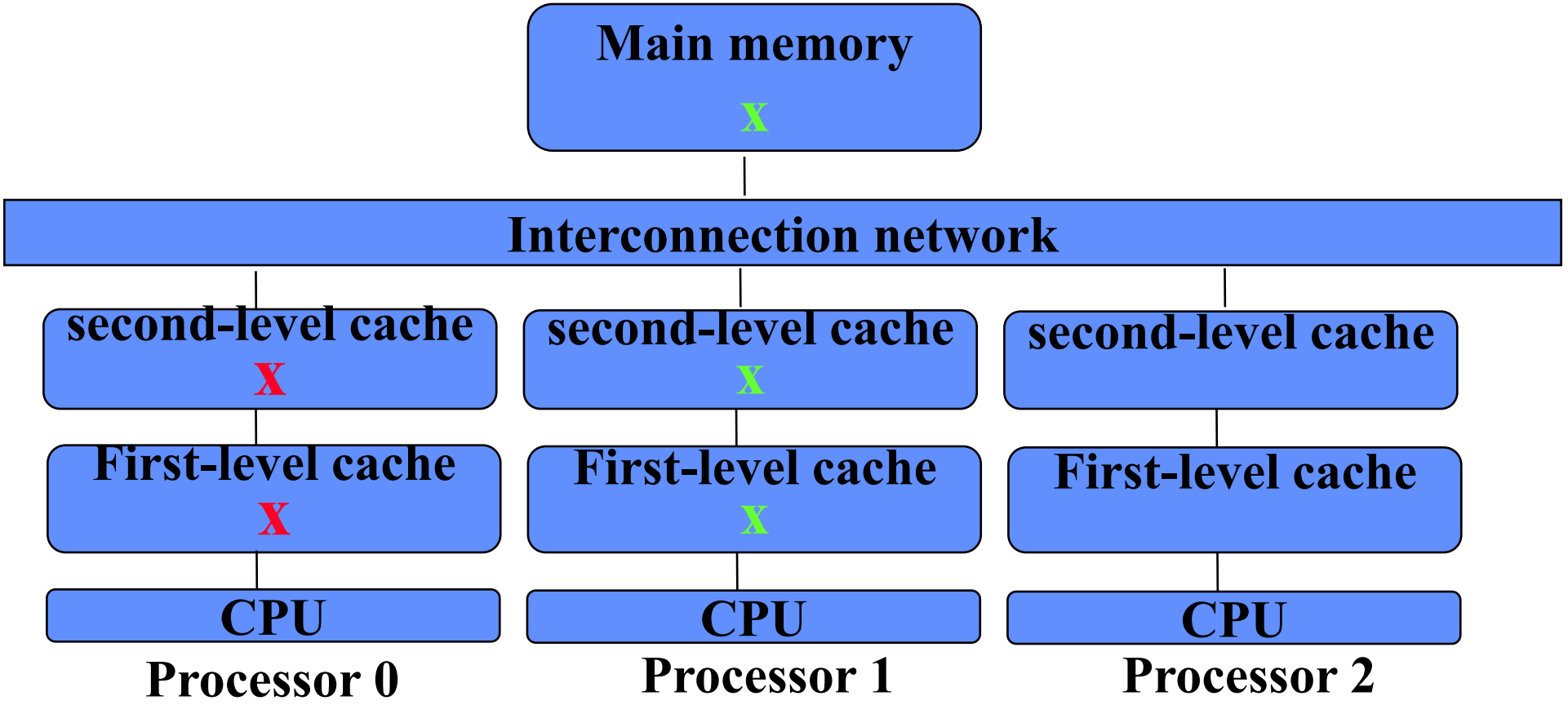
- Suppose processor 0 loads memory location **x**
- **x** is fetched from main memory and allocated into processor 0's cache(s)

Multiple caches… and trouble



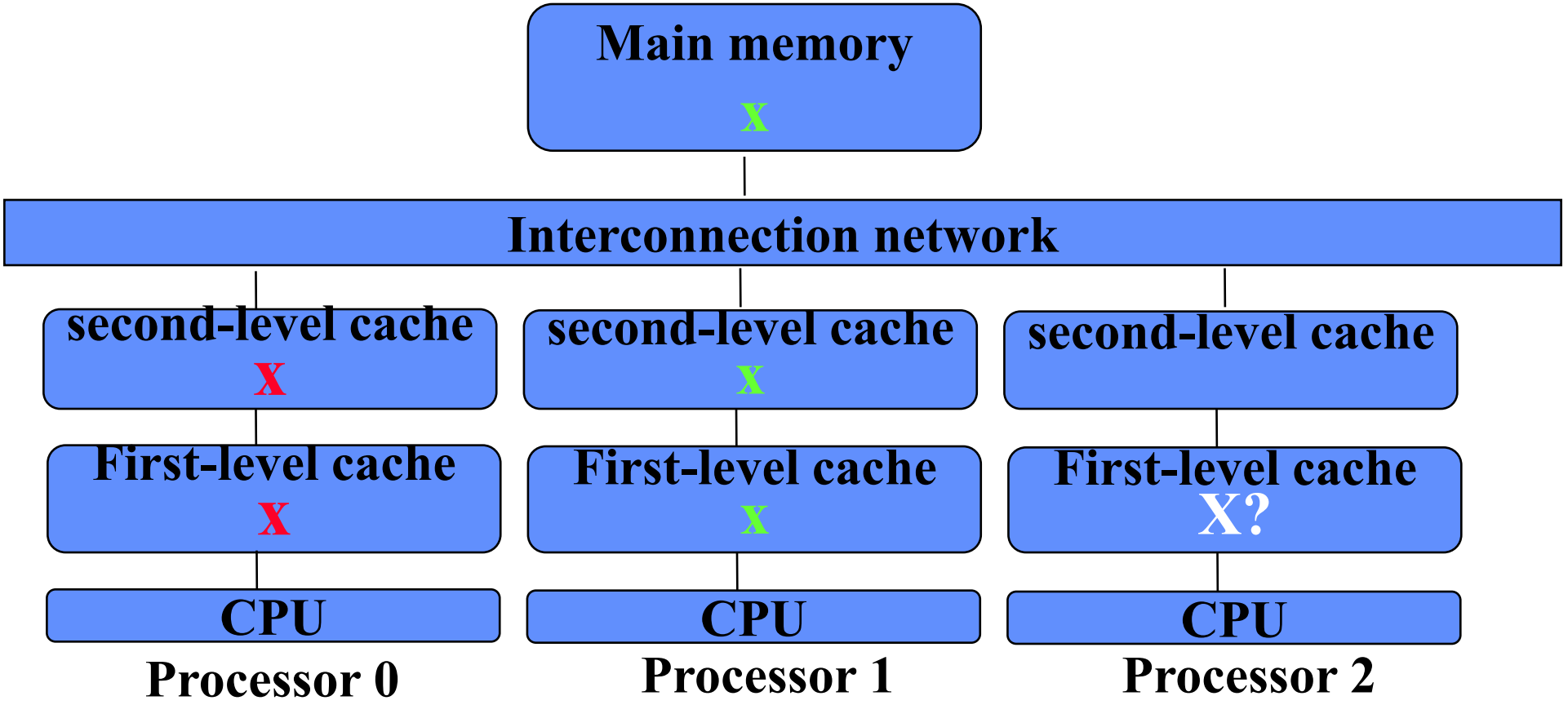
- Suppose processor 1 loads memory location **x**
- **x** is fetched from main memory and allocated into processor 1's cache(s) as well

Multiple caches… and trouble



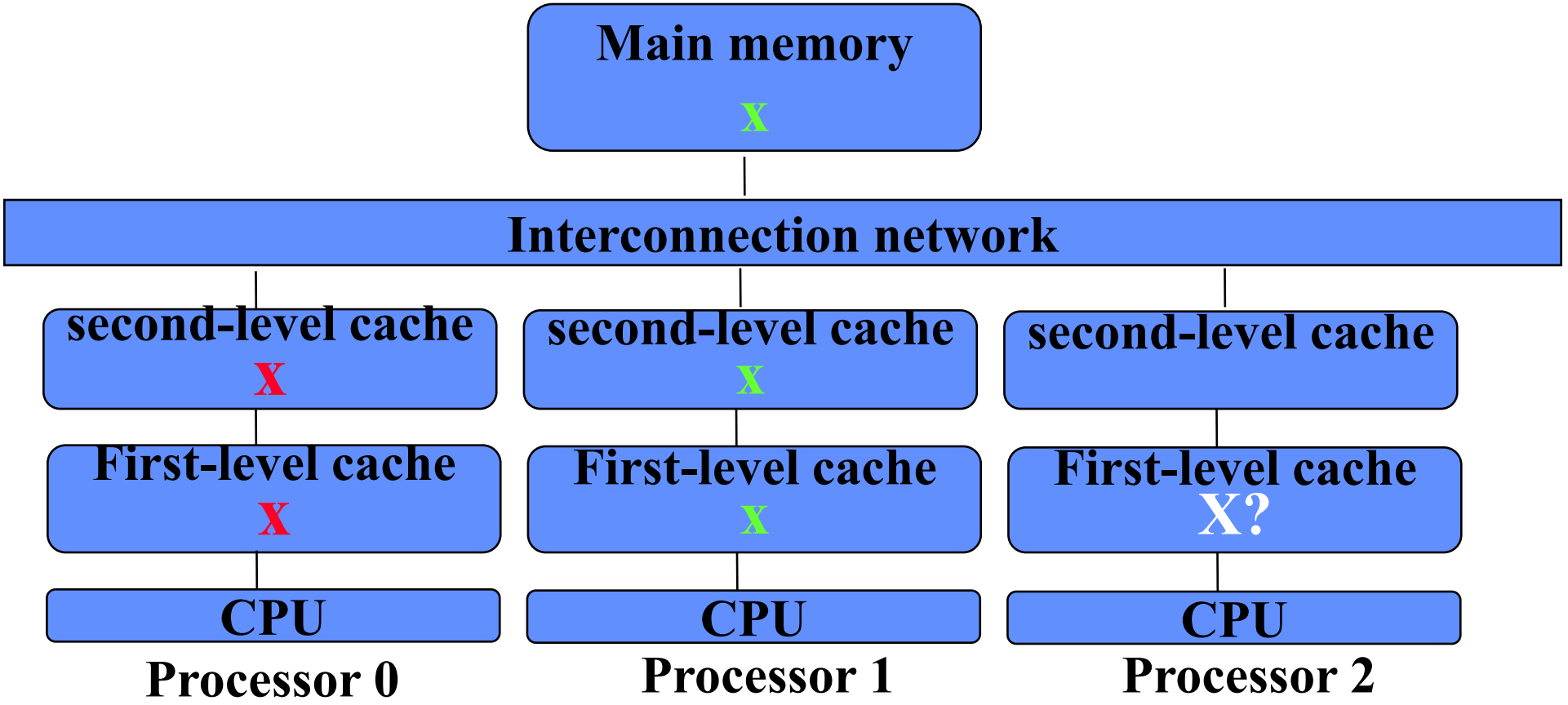
- Suppose processor 0 stores to memory location **x**
- Processor 0's cached copy of **x** is updated
- Processor 1 continues to used the old value of **x**

Multiple caches… and trouble



- Suppose processor 2 loads memory location **x**
- **How does it know whether to get x from main memory, processor 0 or processor 1?**

Multiple caches… and trouble



Two issues:

- How do you know where to find the latest version of the cache line?
- How do you know when you can use your cached copy – and when you have to look for a more up-to-date version?

Cache consistency (aka cache coherency)

- Goal (?):
  - "Processors should not continue to use out-of-date data indefinitely"
- Goal (?):
  - "Every load instruction should yield the result of the most recent store to that address"
- Goal (?): (definition: **Sequential Consistency**)
  - "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"

(Leslie Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs" (IEEE Trans Computers Vol.C-28(9) Sept 1979)

Two pages. 3,300 citations. 70 citations in 2020… https://en.wikipedia.org/wiki/Leslie_Lamport

Implementing Strong Consistency: update

- How about when a store to address x occurs, we **update** all the remote cached copies?
- To do this we need either:
  - To broadcast every store to every remote cache
  - Or to keep a list of which remote caches hold the cache line
  - Or at least keep a note of whether there are *any* remote cached copies of this line ("SHARED" bit per line)
- But first…how well does this update idea work?

Implementing Strong Consistency: update…

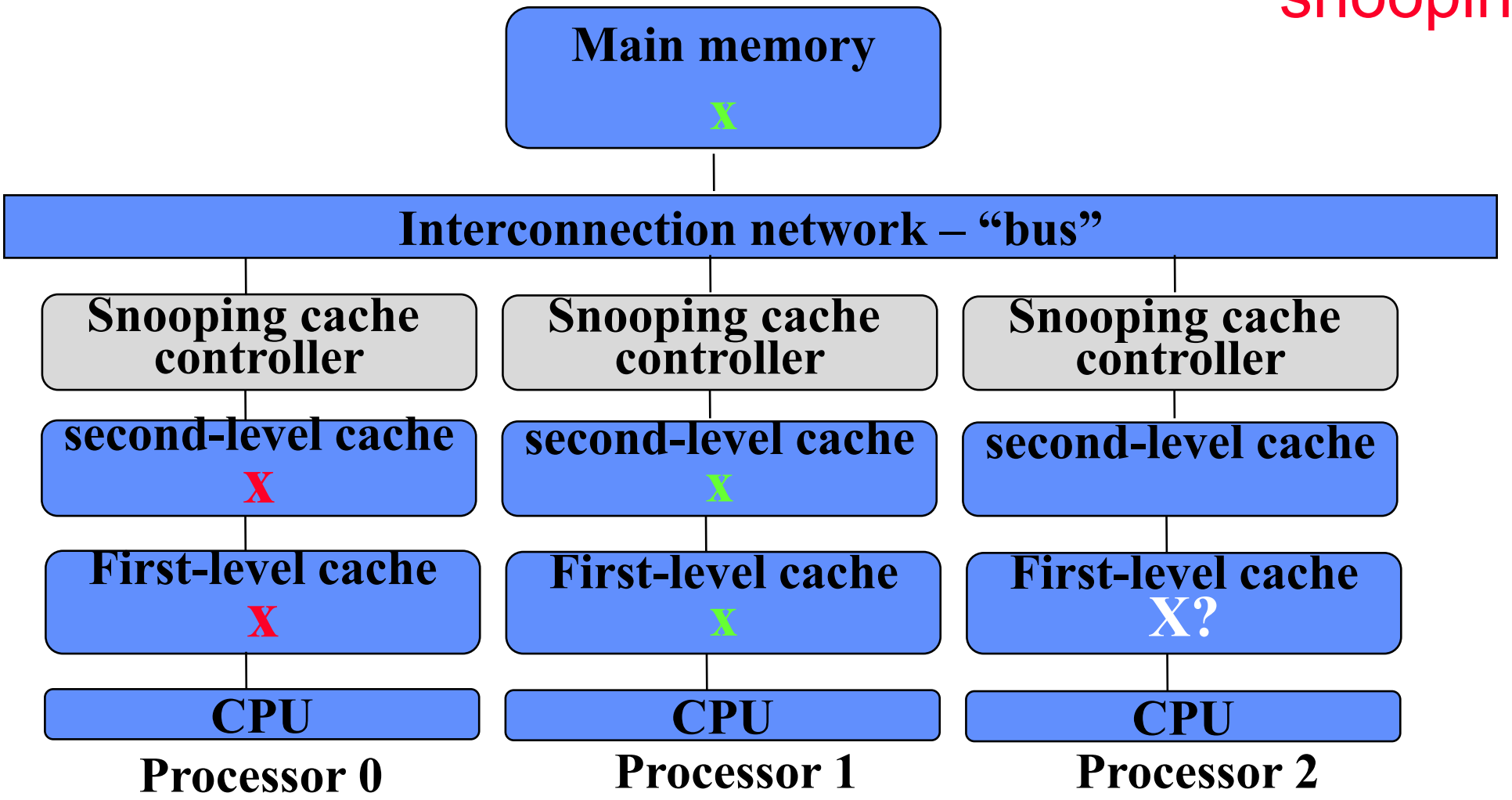
Problems with update

1. What about if the cache line is several words long?
   - Each update to each word in the line leads to a broadcast
2. What about old data which other processors are no longer interested in?
   - We'll keep broadcasting updates indefinitely…
   - Do we really have to broadcast *every* store?
   - It would be nice to know that we have exclusive access to the cacheline so we don't have to broadcast updates…

A more cunning plan… invalidation

- Suppose instead of ***updating*** remote cache lines, we ***invalidate*** them all when a store occurs?
- After the first write to a cache line we know there are no remote copies – so subsequent writes don't lead to communication
  - After invalidation we *know* we have the *only* copy
- Is invalidate always better than update?
  - Often
  - But not if the other processors really need the new data as soon as possible
- To exploit this, we need a couple of bits for each cache line to track its sharing state
- (analogous to write-back vs write-through caches)

snooping



- Snooping cache controller has to monitor *all* bus transactions
- And check them against the tags of its cache(s)

The "Berkeley" Protocol

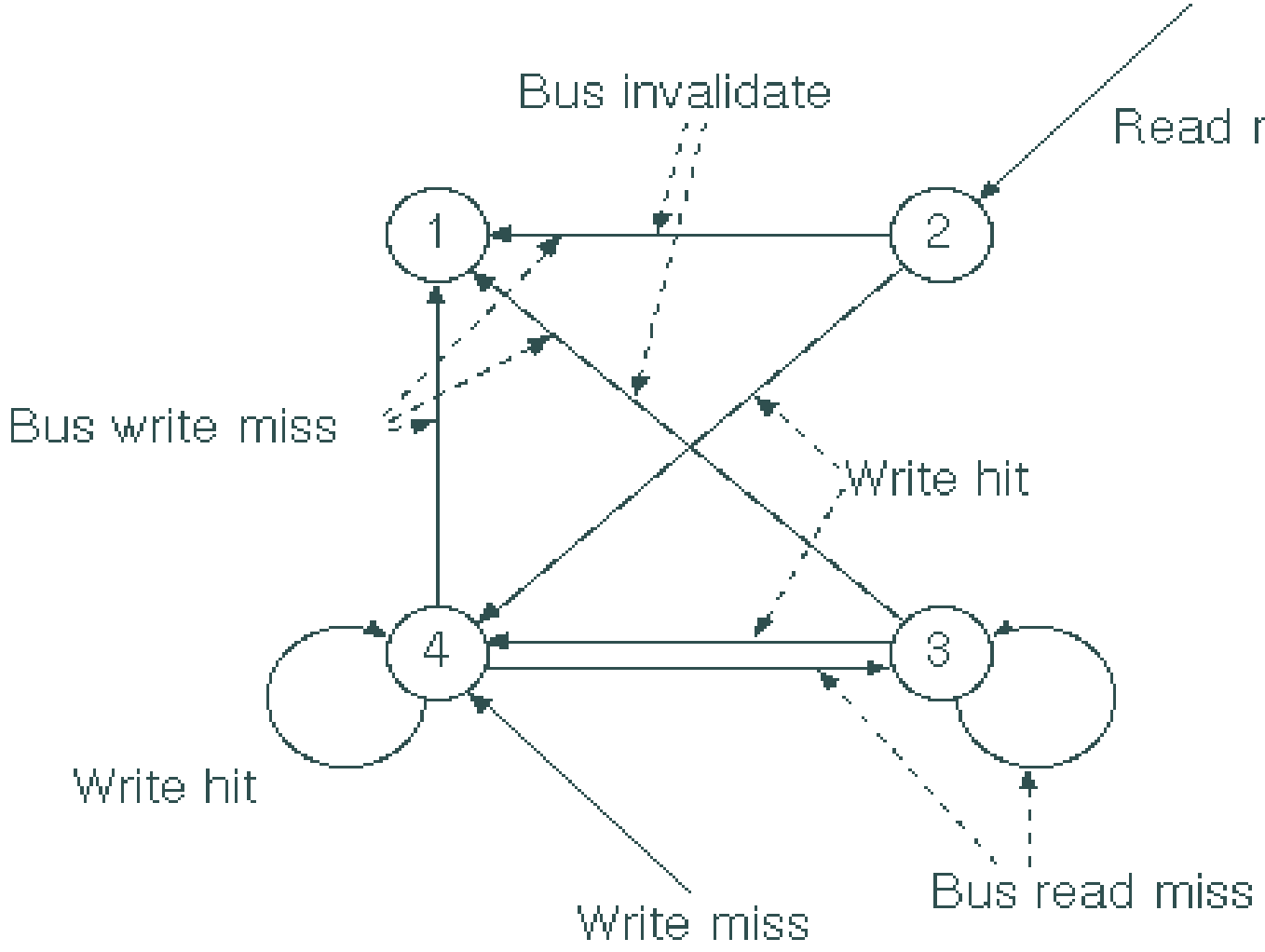
Each cacheline can be in one of four states:

- INVALID
- VALID : clean, potentially shared, unowned
- SHARED-DIRTY : modified, possibly shared, owned
- DIRTY : modified, only copy, owned

Idea: When a store to this cacheline occurs, broadcast an invalidation on the bus unless the cache line is exclusively "owned" (DIRTY)

Read hits are easy. The interesting cases are:

- **Read miss:**
  - We broadcast the request on the bus
  - If another cache has the line in SHARED-DIRTY or DIRTY,
    - it supplies it
    - It sets its line's state to SHARED-DIRTY. We set our copy to VALID
  - Otherwise
    - the line comes from memory. The state of the
    - line is set to VALID
- **Write hit:**
  - No action if line is DIRTY
  - If VALID or SHARED-DIRTY,
    - an invalidation is sent, and
    - the local state set to DIRTY
- **Write miss:**
  - Line comes from owner (as with read miss).
  - All other copies set to INVALID, and line in requesting cache is set to DIRTY



Berkeley cache coherence protocol: state transition diagram

The Berkeley protocol is representative of how typical bus-based SMPs work

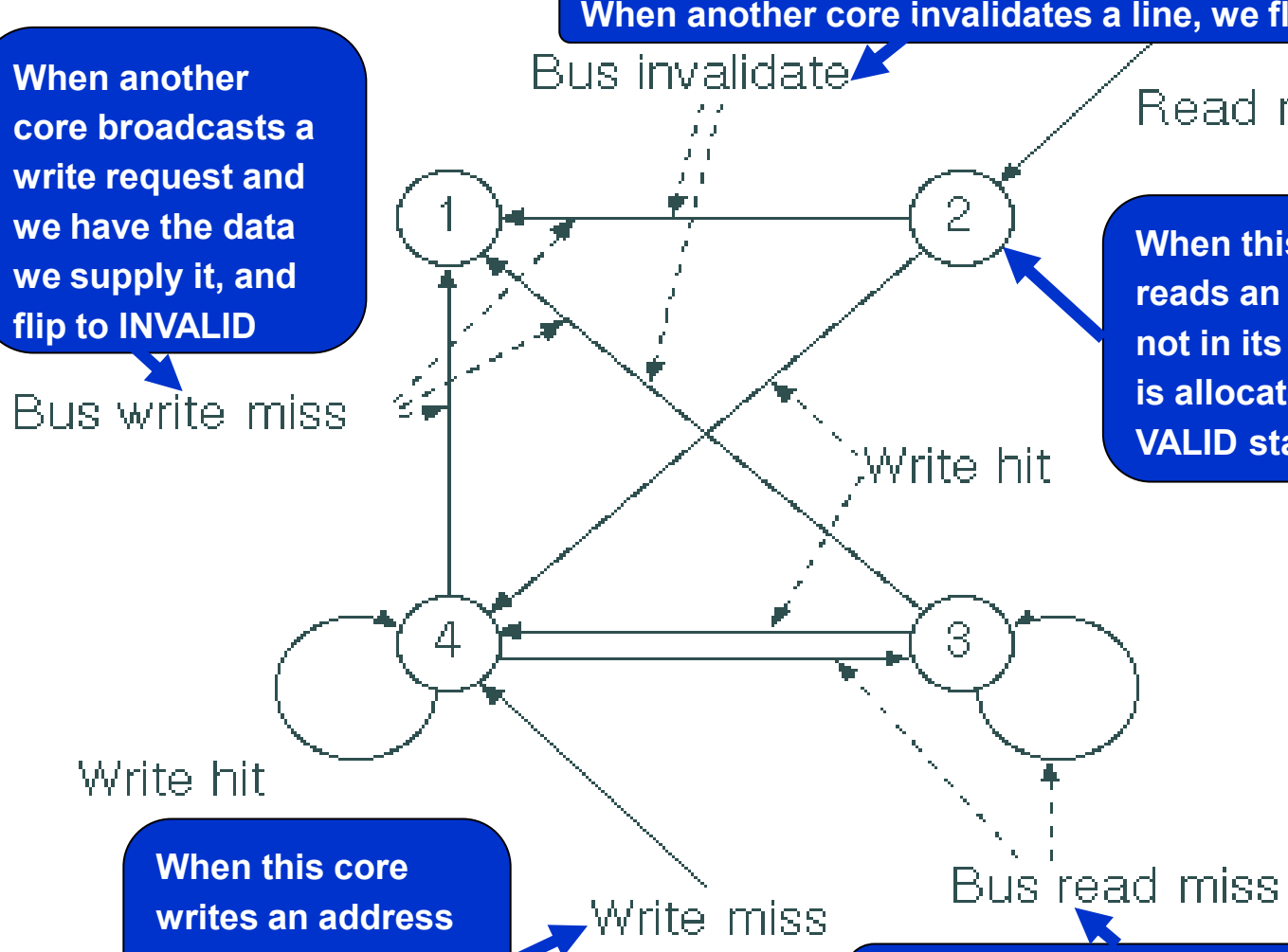
1. INVALID
2. VALID: clean, potentially shared, unowned
3. SHARED-DIRTY: modified, possibly shared, owned
4. DIRTY: modified, only copy, owned

Berkeley cache coherence protocol: state transition diagram

When another core invalidates a line, we flip our copy to INVALID

When another core broadcasts a write request and we have the data we supply it, and flip to INVALID

When this core reads an address not in its cache, it is allocated in the VALID state



The Berkeley protocol is representative of how typical bus-based SMPs work

When this core writes an address not in its cache, it is allocated in the DIRTY state

When another core broadcasts a read request – and we have the line (DIRTY or SHARED-DIRTY) we supply it and flip to SHARED-DIRTY

1. INVALID
2. VALID: clean, potentially shared, unowned
3. SHARED-DIRTY: modified, possibly shared, owned
4. DIRTY: modified, only copy, owned

When a core requests a line but no core holds it, it is supplied from main memory (no “owner”)

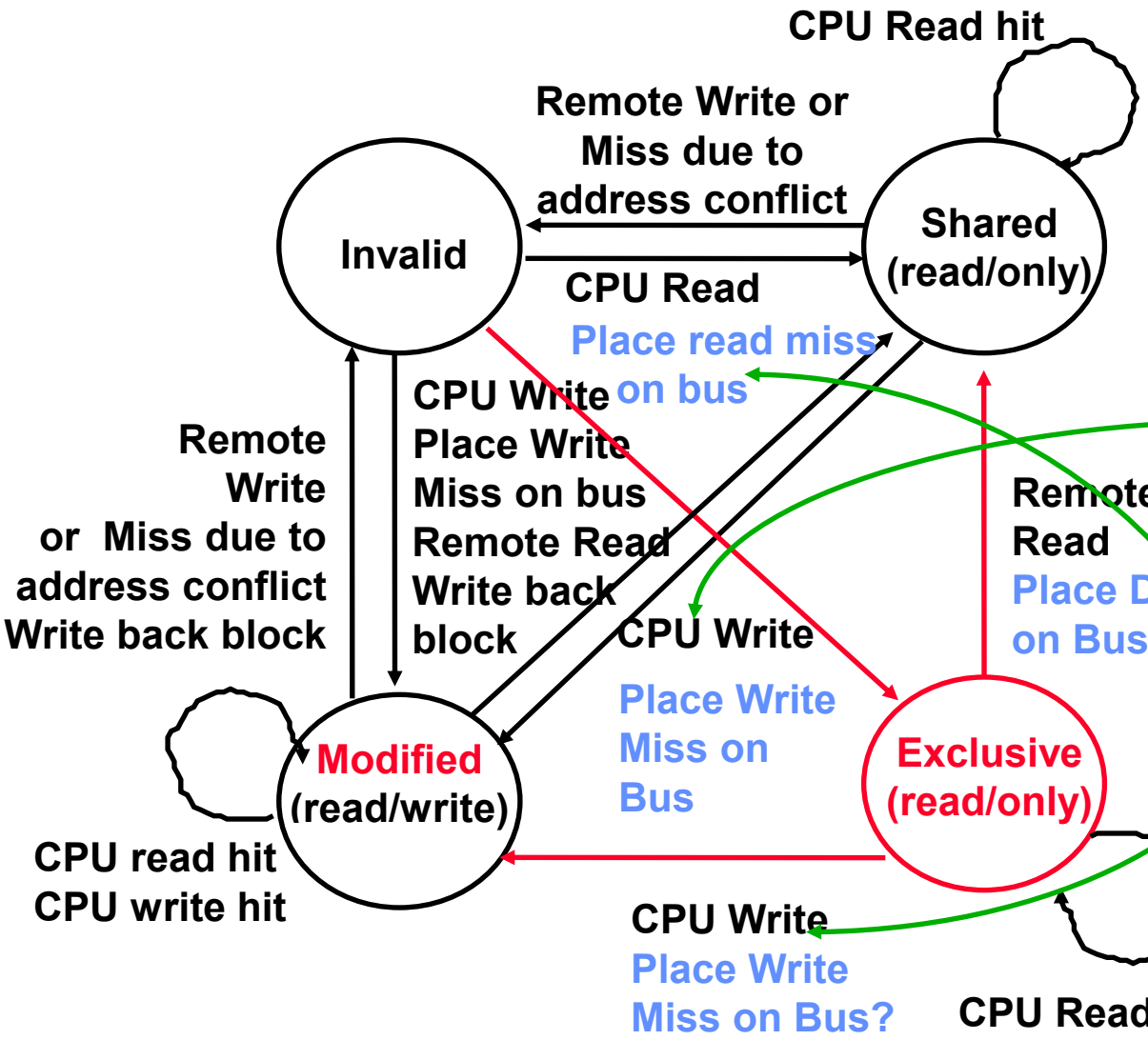
The job of the cache controller - snooping

- The protocol state transitions are implemented by the cache controller – which “snoops” all the bus traffic
- Transitions are triggered either by
  - the bus (Bus invalidate, Bus write miss, Bus read miss)
  - The CPU (Read hit, Read miss, Write hit, Write miss)
- For every bus transaction, it looks up the directory (cache line state) information for the specified address
  - If this processor holds the only valid data (DIRTY), it responds to a “Bus read miss” by providing the data to the requesting CPU
  - If the memory copy is out of date, one of the CPUs will have the cache line in the SHARED-DIRTY state (because it updated it last) – so must provide data to requesting CPU
  - State transition diagram doesn’t show what happens when a cache line is displaced…

Berkeley protocol - summary

- Invalidate is usually better than update
- Cache line state “DIRTY” bit records whether remote copies exist
  - If so, remote copies are invalidated by broadcasting message on bus – cache controllers snoop all traffic
- Where to get the up-to-date data from?
  - Broadcast read miss request on the bus
  - If this CPU’s copy is DIRTY, it responds
  - If no cache copies exist, main memory responds
  - If several copies exist, the CPU which holds it in “SHARED-DIRTY” state responds
  - If a SHARED-DIRTY cache line is displaced, … need a plan
- How well does it work?
  - See extensive analysis in Hennessy and Patterson

There is a design-space of snooping cache protocols…



Extensions:

- Fourth State: Ownership

- **Shared-> Modified, need invalidate only (upgrade request), don't read memory**
  Berkeley Protocol

- **Clean exclusive state (no miss for private data on write)**
  MESI Protocol
- **Cache supplies data when shared state (no memory access)**
  Illinois Protocol

Implementing Snooping Caches

- All processors must be on the bus, with access to both addresses and data
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, there could be contention between bus and CPU access:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: Use the L2 cache to "filter" invalidations
    - *If* everything in L1 is also in L2 (multi-level inclusion)
    - Then we only have to check L1 if the L2 tag matches
  - Many systems enforce cache inclusivity
    - Constrains cache design - block size, associativity
    - Alternative: snoop filter (https://sites.utexas.edu/jdm4372/2019/01/07/sc18-paper-hpl-and-dgemm-performance-variability-on-intel-xeon-platinum-8160-processors/)

Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - Bus transaction is not atomic: can have multiple outstanding transactions for a block
    - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
- Must support interventions and invalidations

Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write-back cache, may have the desired copy and it’s dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

Summary

- Implementing a consistent view of shared memory is tricky
  - Hit-under-miss reorders reads
  - Prefetching re-orders reads
  - Do we wait for invalidations to be acknowledged before continuing?
- We can only promise a limited notion of consistency
- For small systems, a “snooping” scheme works
  - Each core’s cache controller sees all read misses, write misses and invalidations
  - And checks tags of its caches in case action is required
    - Exploiting multi-level inclusion to filter the check
  - There is a design space of cache coherency protocols, that track exclusive ownership, and determine which core supplies data – or whether it comes from memory
  - Implementation issues are much more complex than described here

Student question: “SHARED-DIRTY”

Q: If two cores are sharing a dirty bit, and core 1 decided to go ahead and write this to MM, would this make core 1 have flag VALID? Also, what happens to core 2? What's it's state? Also just a sanity check, if core j requests from core i, they both share the tag SHARED DITY right? Just want to make sure

A: re: "they both share the tag SHARED DIRTY" - **not** right

- Each cache line can be in the SHARED DIRTY state in at most one cache.
- "Shared-dirty" means that

(1) the data is "dirty": the main-memory copy is out of date due to a write [also true of DIRTY]

(2) The cache that holds the line in the "SHARED-DIRTY" line is responsible for providing the data if another core makes a read request on the shared bus. The cache controller sees the read request, checks and sees that the read address matches a line that it holds, and that that line is in the SHARED-DIRTY state - so it responds by providing the data. [also true of DIRTY, but afterwards the line transitions from DIRTY to SHARED-DIRTY] (this "responsible for providing the data" is also called "ownership").

(3) SHARED DIRTY means that another cache might hold a copy of the line (in the VALID state) (because of (2) above). So if this core wants to write to this cache line, it needs to invalidate the remote copies. [in contrast, DIRTY means that there are no remote copies, so the write can proceed without sending an invalidation message]

You might (*might*....) find this rather scary-looking Wikipedia page helpful - search for the Berkeley protocol section: Cache coherency protocols (examples) – Wikipedia

Student question: Instruction cache coherency

Hello

In the slides from Qualcomm about the Snapdragon processor, the L1 instruction cache and Load-Store data cache are described as being fully coherent. What does that mean in this context?

Comment Edit Delete Endorse •••

1 Answer

P

Paul Kelly STAFF

24 seconds ago

As discussed briefly in-class I think, this is about the case when a core (perhaps the same core, perhaps another) stores to an address which is cached in a core's instruction cache.

So we could imagine a thread that writes some code into its memory then jumps to it (as a JIT might do)

Or a thread in a (mult-threaded) Java Virtual Machine (JVM) process, that JITs a Java bytecode method into memory, and then another thread in that JVM tries to execute that code.

We also have the situation where the operating system kernel loads code from a file into a process's virtual address space (eg when you launch a process). It then finds an idle core to actually run that code. That core may have an I-cache full of instructions from some other now-dead process.

But with a fully-coherent I-cache, the physical addresses of the recycled region of the physical address space will be invalidated when the code is loaded, so the core where the process runs will fetch the up-to-date instructions.