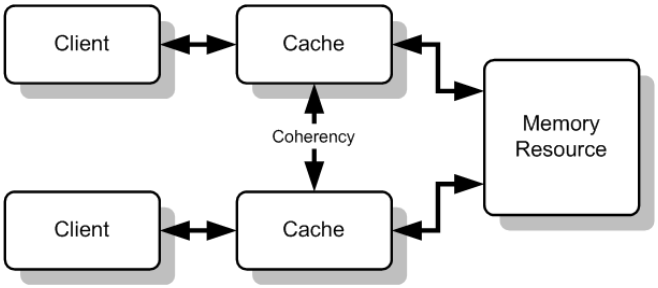
ECE459: Programming for Performance Winter 2019

Lecture 4 — Cache Coherency

Patrick Lam & Jeff Zarnett *2018-12-04*

Cache Coherency



—Wikipedia

We talked about memory ordering and fences last time. Today we'll look at what support the architecture provides for memory ordering, in particular in the form of cache coherence. Since this isn't an architecture course, we'll look at this material more from the point of view of a user, not an implementer.

The problem is of course that each CPU might have its own cache and if it does then these data may be out of sync. The simplest method, and a horrible solution, is to declare some read/write variables as being non-cacheable (is that a word? Uncacheable?...). The compiler and OS and such will require the data to be read from memory always. Think of this as being like `volatile` (which prevented a variable from being put in a register), just moved up a level to preventing the variable from being put in a cache. This will obviously result in lower cache hit ratios, increased bus traffic, and terrible, terrible performance. Let's avoid that. What we want instead is coherency.

Cache coherency means that:

- the values in all caches are consistent; and
- to some extent, the system behaves as if all CPUs are using shared memory.

Cache Coherence Example. We will use this example to illustrate different cache coherence algorithms and how they handle the same situation.

Initially in main memory: `x = 7`.

1. `CPU1` reads x, puts the value in its cache.
2. `CPU3` reads x, puts the value in its cache.
3. `CPU3` modifies `x := 42`
4. `CPU1` reads x . . . from its cache?
5. `CPU2` reads x. Which value does it get?

Unless we do something, `CPU1` is going to read invalid data.

High-Level Explanation of Snoopy Caches. The simplest way to "do something" is to use Snoopy caches [KMRS88]. No, not this kind of Snoopy (sadly):

This is a distributed approach; no centralized state is maintained. Each cache with a copy of data from a block of physical memory knows whether it is shared or not. All the CPUs are connected to a shared bus, and each CPU has its own cache controller. Whenever a CPU issues a memory write, the other CPUs are watching (snooping around) to observe if that memory location is in their cache. If it is, then the CPU will need to take action.

Write-Through Caches

Let's put that into practice using write-through caches, the simplest type of cache coherence.

- All cache writes are done to main memory.
- All cache writes also appear on the bus.
- If another CPU snoops and sees it has the same location in its cache, it will either *invalidate* or *update* the data.

Invalidation is the most common protocol. It means the data in the cache of other CPUs is not updated, it's just noted as being out of date (invalid). Normally, when you write to an invalidated location, you bypass the cache and go directly to memory (aka **write no-allocate**). If we want to do a read and there's a miss, we can poke around in other caches to see who has the most recent cached version. This is a bit like going into a room and yelling "Does anybody have block...?", in some sort of multicast version of the card game "Go Fish". Regardless, the most recent value appears in memory, always.

There are also write broadcast protocols, in which case all versions in all caches get updated when there is a write to a shared block. But it uses lots of bandwidth and is not necessarily a good idea. It does, however prevent the costly cache miss that follows an invalidate. Sadly, as we are mere users and not hardware architects, we don't get to decide which is better; we just have to live with whichever one is on the hardware we get to use. Bummer.

Write-Through Protocol. The protocol for implementing such caches looks like this. There are two possible states, **valid** and **invalid**, for each cached memory location. Events are either from a processor (**Pr**) or the **Bus**. Actions will be either a **Rd** (read) or **Wr** (write). We then implement the following state machine.

State	Observed	Generated	Next State
Valid	PrRd		Valid
Valid	PrWr	BusWr	Valid
Valid	BusWr		Invalid
Invalid	PrWr	BusWr	Valid
Invalid	PrRd	BusRd	Valid

Example. For simplicity (this isn't an architecture course), assume all cache reads/writes are atomic. [1] Using the same example as before:

Initially in main memory: `x = 7`.

1. `CPU1` reads x, puts the value in its cache. (valid)
2. `CPU3` reads x, puts the value in its cache. (valid)
3. `CPU3` modifies `x := 42`. (write to memory)
   - `CPU1` snoops and marks data as invalid.
4. `CPU1` reads x, from main memory. (valid)
5. `CPU2` reads x, from main memory. (valid)

Write-Back Caches

Let's try to improve performance. What if, in our example, `CPU3` writes to `x` 3 times? It's unpleasant to have to flush that to memory three times when we could do it only once. Let's try to delay the write to memory as long as possible. At minimum, we have to add a "dirty" bit, which indicates the our data has not yet been written to memory.

Write-Back Implementation. The simplest type of write-back protocol (MSI) uses 3 states instead of 2:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Shared**—location is unmodified, up-to-date with main memory; may be present in other caches (also up-to-date).
- **Invalid**—same as before.

The initial state for a memory location, upon its first read, is "shared". The implementation will only write the data to memory if another processor requests it. During write-back, a processor may read the data from the bus.

MSI Protocol. Here, bus write-back (or flush) is **BusWB**. Exclusive read on the bus is **BusRdX**.

State	Observed	Generated	Next State
Modified	PrRd		Modified
Modified	PrWr		Modified
Modified	BusRd	BusWB	Shared
Modified	BusRdX	BusWB	Invalid
Shared	PrRd		Shared
Shared	BusRd		Shared
Shared	BusRdX		Invalid
Shared	PrWr	BusRdX	Modified
Invalid	PrRd	BusRd	Shared
Invalid	PrWr	BusRdX	Modified

[1] If you're a hardware person, this line probably makes you cry. There's a whole lot that goes into making this work. There are potential write races, which have to be dealt with by contending for the bus and then completing the transaction, possibly restarting a command if necessary. If we have a split transaction bus it's really ugly, because we can have multiple interleaved misses. And down the rabbit hole we go.

MSI Example. Using the same example as before:

Initially in main memory: `x = 7`.

1. `CPU1` reads x from memory. (BusRd, shared)
2. `CPU3` reads x from memory. (BusRd, shared)
3. `CPU3` modifies `x = 42`:
   - Generates a BusRdX.
   - `CPU1` snoops and invalidates x.
4. `CPU1` reads x:
   - Generates a BusRd.
   - `CPU3` writes back the data and sets x to shared.
   - `CPU1` reads the new value from the bus as shared.
5. `CPU2` reads x from memory. (BusRd, shared)

An Extension to MSI: MESI

The most common protocol for cache coherence is MESI. This protocol adds yet another state:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Exclusive**—only this cache has a valid copy; main memory is **up-to-date**.
- **Shared**—same as before.
- **Invalid**—same as before.

MESI allows a processor to modify data exclusive to it, without having to communicate with the bus. MESI is safe. The key is that if memory is in the E state, no other processor has the data.

MSEIF: Even More States!

MESIF (used in latest i7 processors):

- **Forward**—basically a shared state; but, current cache is the only one that will respond to a request to transfer the data.

Hence: a processor requesting data that is already shared or exclusive will only get one response transferring the data. This permits more efficient usage of the bus.

Cache coherence vs flush

> Cache coherency seems to make sure my data is consistent. Why do I have to have something like flush or fence?

Sadly, no. Cache coherence isn't enough. Writes may be to registers rather than memory, and those won't be coherent. Use fences or flushes.

Well, I read that `volatile` variables aren't stored in registers, so then am I okay?

Again, sadly, no. Recall that `volatile` in C was only designed to [Rit13]:

- Allow access to memory mapped devices.
- Allow uses of variables between `setjmp` and `longjmp`.
- Allow uses of `sig_atomic_t` variables in signal handlers.

All of these things apply to single-threaded situations or cases where the compiler has re-ordered some steps. Remember, things can also be reordered by the compiler, and `volatile` doesn't prevent reordering. Also, it's likely your variables could be in registers the majority of the time, except in critical areas. All that volatile does is tell the compiler that an access of that variable should go to memory, not a register.

Coherence summary. We saw four cache coherence protocols, from MSI through MESIF. There are many other protocols for cache coherence, each with their own trade-offs.

Recall: OpenMP flush acts as a **memory barrier/fence** so the compiler and hardware don't reorder reads and writes. Neither cache coherence nor `volatile` will save you. Barriers/fences/flushes might be expensive in terms of "wasting time," but correctness is important.

References

[KMRS88] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988. URL: `http://dx.doi.org/10.1007/BF01762111`.

[Rit13] Peter Ritchie. volatile - you keep using that word, but i do not think you know what it means, 2013. Online; accessed 15-December-2015. URL: `http://blog.peterritchie.com/?p=1091`.