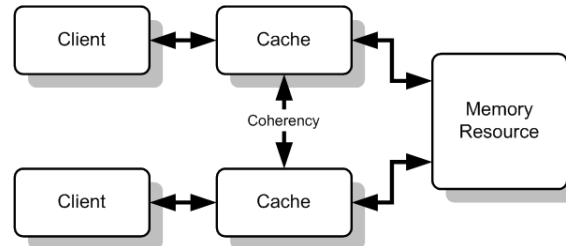


## Lecture 17 — Cache Coherency

Patrick Lam

## 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.

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  $x \dots$  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. The setup is as follows:

- Each CPU is connected to a simple bus.
- Each CPU “snoops” to observe if a memory location is read or written by another CPU.
- We need a cache controller for every CPU.

Then:

- Each CPU reads the bus to see if any memory operation is relevant. If it is, the controller takes appropriate 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.  
(We'll be looking at invalidating.)

Normally, when you write to an invalidated location, you bypass the cache and go directly to memory (aka **write no-allocate**).

**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**. 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	Invalid
Invalid	PrRd	BusRd	Valid

**Example.** For simplicity (this isn't an architecture course), assume all cache reads/writes are atomic. 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?

**Main goal.** 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. Let's do that.

**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

**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:

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

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.

**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.