Lecture 15—Cache Coherency

ECE 459: Programming for Performance

March 6, 2015

Part I

Cache Coherency

Introduction

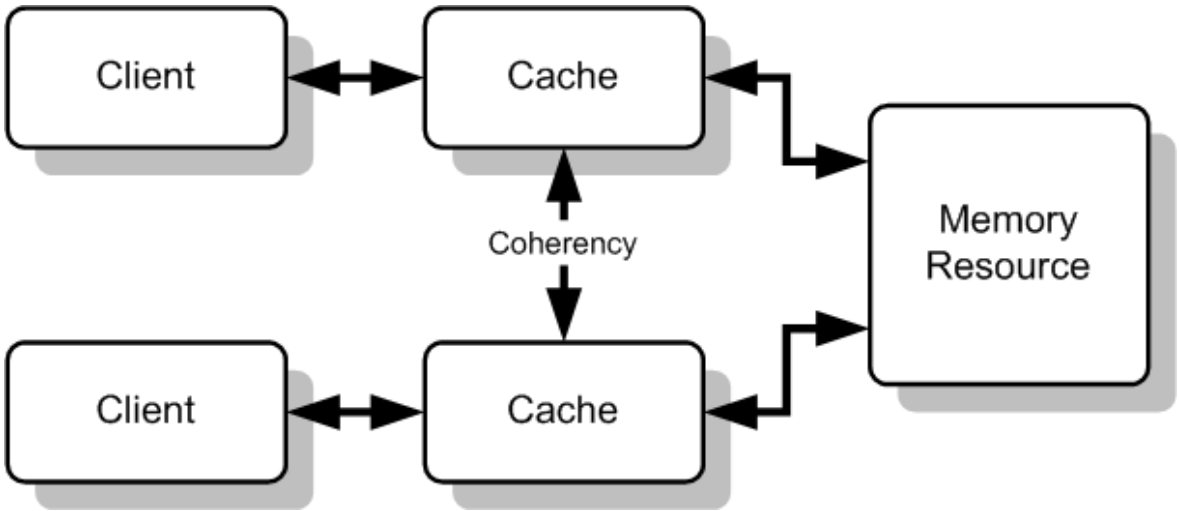

Image courtesy of Wikipedia.

Coherency:

- Values in all caches are consistent;
- System behaves as if all CPUs are using shared memory.

Cache Coherence Example

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

- 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.

What happens?

- Each CPU reads the bus to see if any memory operation is relevant. If it is, the controller takes appropriate action.

Write-Through Cache

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.)

For write-through caches: normally, when you write to an invalidated location, you bypass the cache and go directly to memory (aka **write no-allocate**).

Write-Through Protocol

- Two states, **valid** and **invalid**, for each memory location.
- Events are either from a processor (**Pr**) or the **Bus**.

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

Write-Through Protocol 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 Cache

- 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:
  Indicates the our data has not yet been written to memory.

Write-Back Implementation

The simplest type of write-back protocol (MSI), with 3 states:

- **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.

Initial state, upon first read, is "shared".

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

- 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.
   - `CPU3` changes `x`'s state to modified.
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.

Adds 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: in E state, no other processor has the data.

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.

Permits more efficient usage of the bus.

Good Questions (1)

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

- You might be ok, if all of the writes on processors are to the cache. But they're not!
- Cache coherency won't update any values modified in registers.

Good Questions (2)

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

Good Questions (2)

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

Good Questions (2)

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

`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, `volatile` doesn't prevent this.

Also, it's likely your variables could be in registers the majority of the time, except in critical areas.

Cache Coherency Summary

We saw the basics of cache coherence (good to know, but more of an architecture thing).

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.