Lecture 4 — Cache Coherency

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 8, 2018

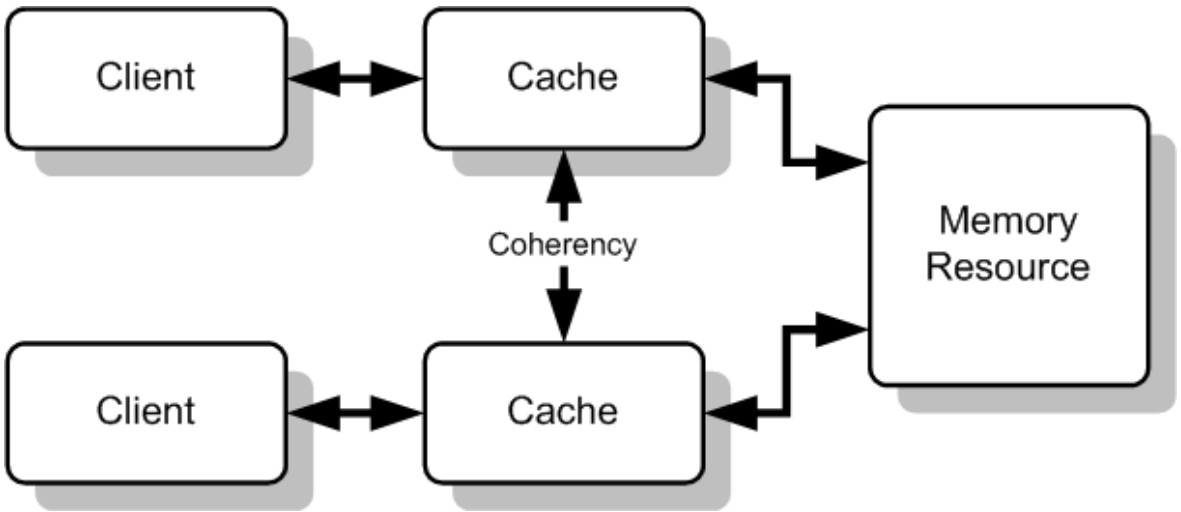


Image courtesy of Wikipedia.

Each CPU has its own cache: coordination is needed!

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.

Planes and Trains

Snoopy Caches

The simplest way to “do something” is to use Snoopy caches.

No, not this kind of Snoopy (sadly):

High-Level Explanation of Snoopy Caches

Snoopy: the caches are spying on each other.

- 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.

A Distributed Approach

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.

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.

We Need Some Action!

What does action mean?

The Air Canada action was update.

The Deutsche Bahn action was invalidate.

Write-Through Cache

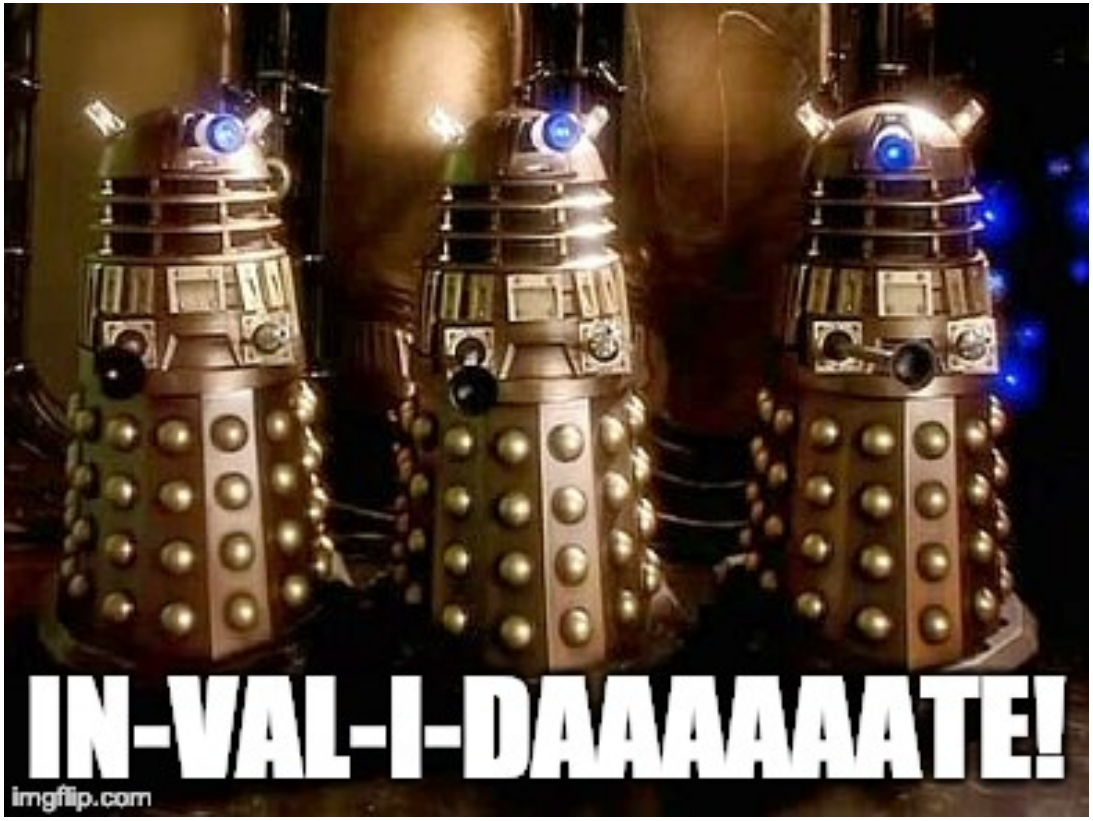
Simplest type of cache coherence:

- 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.

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**).

The Daaaaaaaaaaleks!

Our chosen approach:



In-val-i-daaaaaate!

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.

Let's Yell

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; we just have to live with whichever one is on the hardware we get. Bummer.

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	Valid
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.

False Sharing

False sharing: program has two unrelated data elements that are mapped to the same cache line/location.

```
char a[10];
char b[10];
```

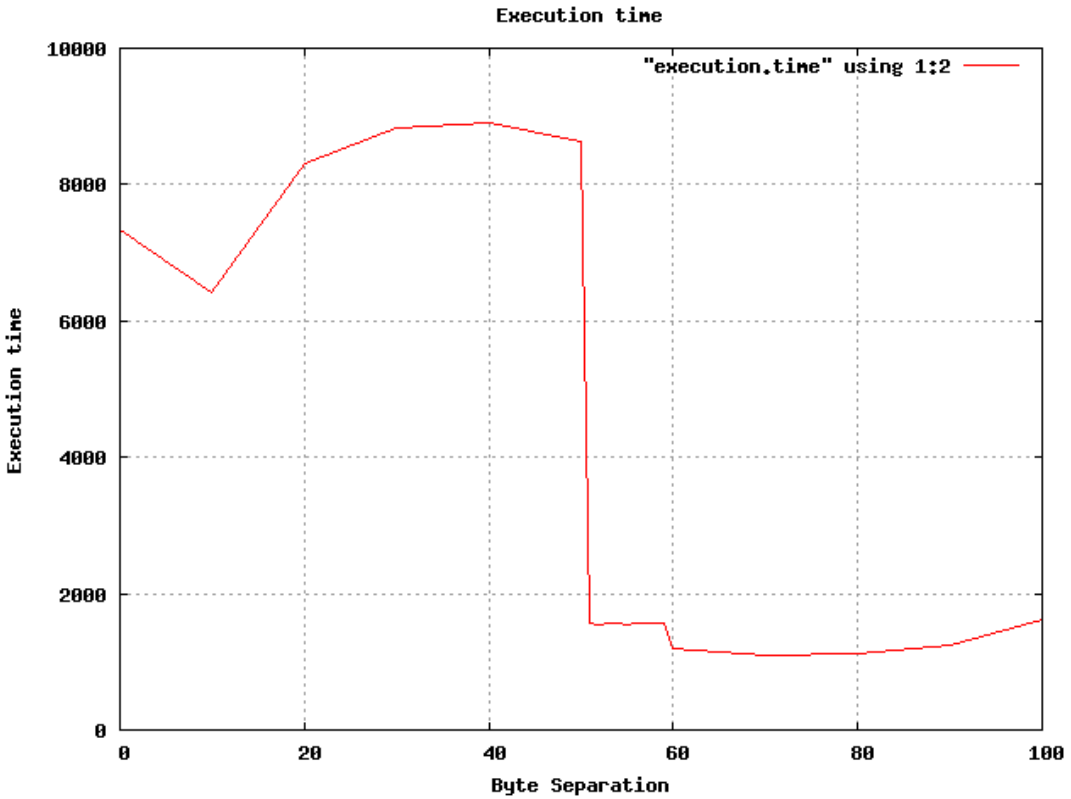
They don’t overlap but are located next to each other in memory.

False Sharing

Solution: Heap allocation? Maybe...

Make both arrays bigger, wasting some space...?

False Sharing



Is wasting a little space worth it? Yes!

He’s a little volatile...

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

He’s a little volatile...

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

He's a little volatile...

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.