Lecture 19 - Cache Coherency

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

February 17, 2012

Previous Lecture

- More cases were giving the compiler more information gives you better code
- Data structures can be very important, more so than complexity
- **Low-level code != Efficient**
- Low-level details however, can hugely change your performance

Previous Example

- Vectors outperformed lists doing operations lists are good at
- You wondered that if we increased the size of the elements to be equivalent the cache line size (64 bytes) we should see lists pull ahead

For **N = 5000** on my laptop

Block Size (bytes)	**Vector (s)**	**List (s)**
256	0.4	0.64
416	0.65	0.67
512	0.8	0.7
1024	1.59	0.83

- At 64 bytes, vectors still won by a large margin
- It didn't even back out until about 416 bytes per element

Introduction

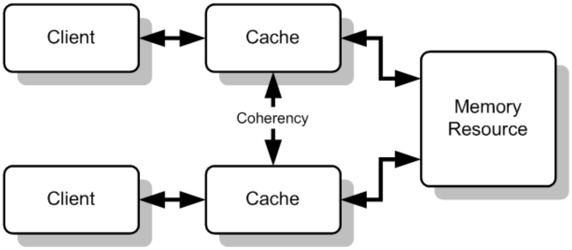

Image courtesy of Wikipedia

Coherency

- The values in all the caches match
- Act as if all CPUs are using shared memory

Example

Main memory: `x = 7`

- `CPU1` reads `x`, puts the value in its cache
- `CPU3` reads `x`, puts the value in its cache
- `CPU3` modifies `x = 42`
- `CPU1` reads `x`, from its cache?
- `CPU2` reads `x`, what value does it get?

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

Snoopy Cache, At a High Level

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

What happens?

- Each CPU reads the bus, sees if any memory operation is relevant, if it is, the controller takes appropriate action

Write-Through Cache

- The simpliest 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, it will either *invalidate* or *update* the data (we'll be looking at invaldating)
- For write-through caches, normally when you write to an invalidate location, you bypass the cache and go directly to memory (**write no-allocate**)

Write-Through Protocol

- Two states, **valid** and **invalid** for each memory location
- The 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 Example

- For simplicity (this isn't an architecture course) all cache reads/writes are atomic

Using the same example as before:
Main memory: `x = 7`

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

Write-Back Cache

- What if, in our example `CPU3` writes `x` 3 times?
- Main goal is to avoid the write to memory as long as possible
- Minimum we have to add a “dirty” bit
- Indicates the our data has not yet been written to memory

Write-Back Implementation

- The simpliest type of write-back protocol, 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
- The initial state when data is read is shared

- Basically, it will only write the data to memory if another processor requests it
- During the write-back, a processor may read the data from the bus

MSI Protocol

- The bus write-back or flush is **BusWB**
- The 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:

Main memory: `x = 7`

- `CPU1` reads x from memory (BusRd, shared)
- `CPU3` reads x from memory (BusRd, shared)
- `CPU3` modifies `x = 42`
  - Generates a BusRdX
  - `CPU1` snoops and invalidates x
  - `CPU3` changes `x`'s state to modified
- `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
- `CPU2` reads x from memory (BusRd, shared)

MSI Extension

- 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

- This allows a processor to modify data that is exclusive, without having to communicate with the bus
- We can do this, because we know no other processor has a copy of the data

Even More States

- MESIF (used in latest i7 processors)
  - **Forward** - basically a shared state, this cache is the only one that will respond to a request to transfer the data

- A processor requesting data that is already shared or exclusive, will only get one response to transfer the data
- This permits more efficient usage of the bus

Possible Questions (1)

Cache coherency seems to make sure my data is consistent, why do I have to have something equivalent to OpenMP's flush?

- You might be ok, if all of the writes of the processor were to the cache, but they're not
- Cache coherency won't update any values modified in registers

Possible Questions (2)

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

Possible Questions (2)

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

ಠ_ಠ

Possible 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 critcal areas

Summary

- The OpenMP flush also acts as a **memory barrier/fence** so the compiler and hardware do not reorder your reads and writes
- Neither cache coherence nor `volatile` will save you
- Basics of cache coherence (good to know, but more of an architecture thing)
- There's many other protocols for cache coherence, each with their own trade-offs