MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 13 - Memory Consistency Issues

Graham Riley

The Consistency Problem

- We have seen the need for cache coherence in multi-core processors
- Cache coherence ensures that, if one core writes (stores) to a particular memory location then any subsequent (in real time) read (load) from that location, by any other core, will see the latest written value
- However, this is all concerned with a single memory location; when we look at the more general case there are other potential problems

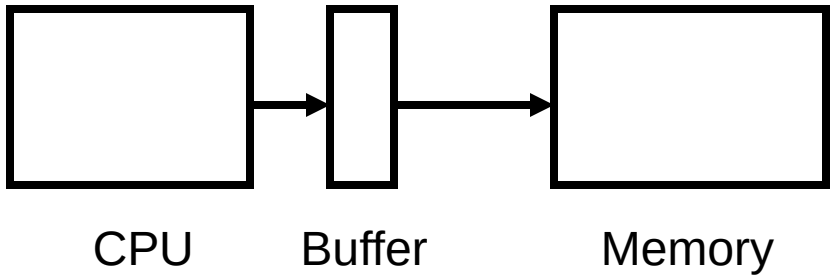
Memory Write Buffer (1)

- Assume a CPU (core) **without any caches**
- A program running on that CPU executes the following

```
str r0, x
add r0,#4
```

- The store will take a long time to execute and the next instruction doesn't need to wait for it, so we buffer the store and start executing the next instruction(s) immediately
- Can we execute *any* following instructions?

Memory Write Buffer (2)

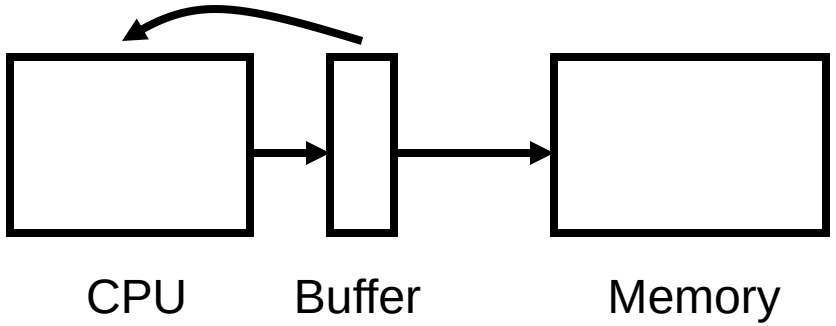


- Add a hardware buffer to hold the address and data
- A store instruction simply writes to the buffer
- Autonomous hardware can then complete the memory transfer in due course
- CPU can get on with executing the program

Loads Out-of-Order?

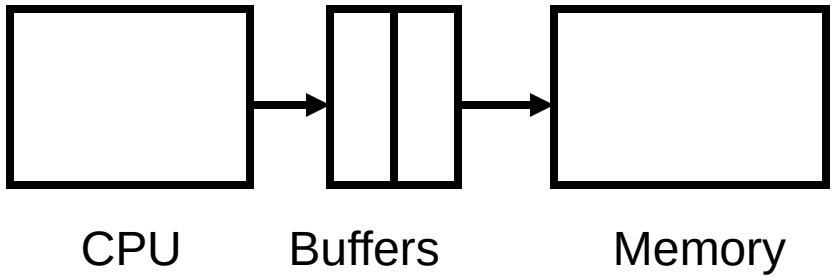
- It is clearly possible to let arithmetic etc. instructions using data in registers run ahead of the store completing (i.e. value actually reaching memory)
- But what about a load (read from memory)?
- In a sequence of instructions, any load must always return the value written by the last store to the same address
- If there is a write **to the same address** in the memory write buffer, the load must return that value

Loads and Memory Write Buffer



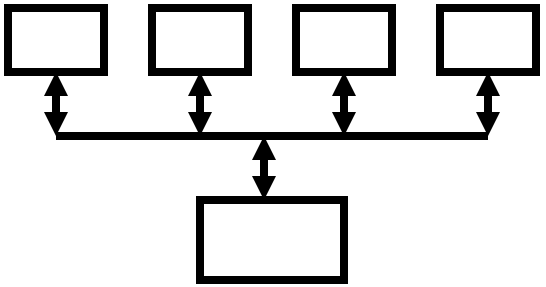
- We could wait until the store reaches memory, then execute the load
  - But this would waste time
- Alternatively, we can check the address in the memory write buffer and use the value there if the address matches
  - Called ‘store-to-load forwarding’
  - Allows loads to ‘overtake’ stores (in terms of completion)

Multiple Memory Write Buffers



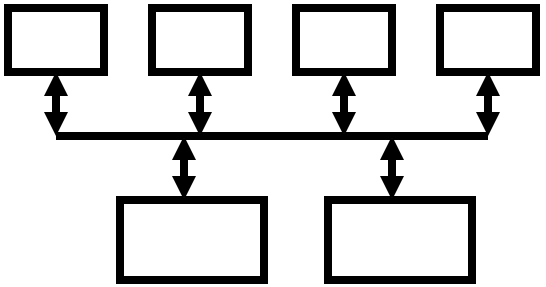
- But what if two stores follow in close succession?
- Now need to wait until memory write buffer is empty before executing the second store
- Then multiple stores will back up if they occur too frequently
- Therefore use multiple memory write buffers
  - Have to decide whether to allow multiple buffered stores to the same address
  - Loads must check all buffers for a matching address

Multi-Core Memory



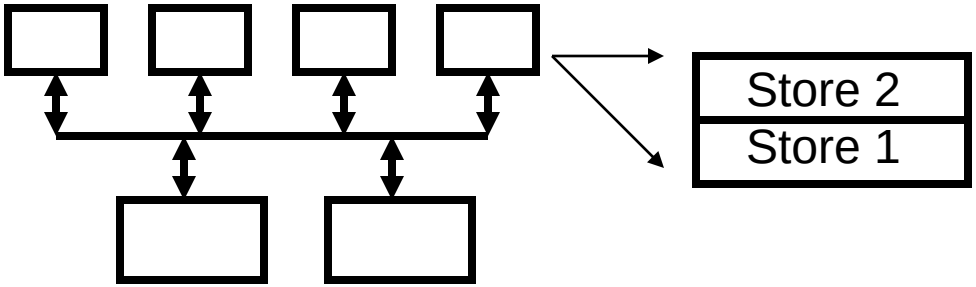
- Now assume we have a multi-core processor with such buffers
- In principle, there is no problem, they just take their turn to write to memory
- But, in a multi-core processor, a single memory can easily become a bottleneck – increased memory traffic from multiple cores

Multi-Banked Memory



- Solution is apparently simple
- Use multiple memory banks – split the address space between them
- If accesses are split between addresses in two distinct memory banks, we can get twice the speed (bandwidth)

Out-of-Order Stores

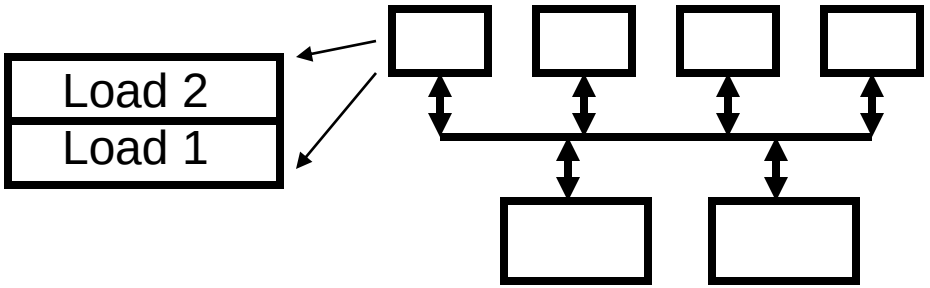


- Now assume that a core has two stores in its buffer
- The earliest one, Store 1, is to a memory bank which is busy
- But Store 2 is to a memory bank which is free
- Do Store 2 first – if we wait, we gain little from our multiple memory structure

A Problem?

- Is this a problem?
- For a single core, subsequent load will always either get the value from the memory write buffer or go to memory
- For a given address the load will always arrive at memory after any previous store to the same address
- As long as we service requests to a given memory in order, we always get the correct value
- But what about other cores?

Erroneous Loads



- Now consider loads from another core
- Because Store 1 occurs before Store 2 in program order, we would never expect to see Load 1 reading an ‘older’ value than Load 2
- But, because Store 1 got held up, the stores have completed out-of-order and so it could happen

Health Warning

- This has been a highly simplified analysis of how stores can appear to be out-of-order
- In modern super-scalar (parallel pipeline) processors it is more likely to be due to a combination of out-of-order instruction execution and multi-banked caches
- However, the detail of all this is too complex for us to study in the time available
- The important message is that apparently safe optimisations in one core can cause its stores to appear out-of-order to another

Does this Matter?

- Suppose we have a program with two threads sharing an array protected by a single lock
  - Thread 1 acquires the lock
  - Thread 1 writes to the array (multiple elements)
  - Thread 1 releases the lock
  - Thread 2 acquires the lock
  - Thread 2 reads the data
  - Thread 2 releases the lock
- The lock ensures that Thread 2 cannot read elements of the array until Thread 1 has finished writing

Thread Detail

- Part of the code might look like:

Thread 1	Thread 2
get lock	get lock
str r1, [r3,#0]	ldr r1, [r3,#0]
str r2, [r3,#4]	ldr r2, [r3,#4]
str #1, lock	release lock

- Thread 1 gets the lock then writes 2 elements of the array and releases the lock – Thread 2 gets the lock then reads the 2 elements and releases the lock

What Would We Expect?

- We haven't said how the two threads execute relative to each other
- If Thread 2 manages to get the lock and read the array before Thread 1 gets the lock, we will see 2 old values
- If Thread 1 gets the lock first and writes to the array before Thread 2 reads it, we will see 2 new values
- What we would never expect is to see one old value and one new value – the lock was supposed to make the 2 array writes atomic

Reversed Order

Thread 1	Thread 2
get lock	get lock
str r1, [r3,#0]	ldr r1, [r3,#0]
str #1, lock	ldr r2, [r3,#4]
str r2, [r3,#4]	str #1, lock

- But we might see some stores as having completed out-of-order, as above – what happens now depends on how instructions interleave between the two cores

Erroneous Interleaving

```
get lock
str r1, [r3,#0]
str #1, lock
                                get lock
                                ldr r1, [r3,#0]
                                ldr r2, [r3,#4]
str r2, [r3,#4]
```

Inconsistency

- Because we have allowed the stores to complete out-of-order, we have seen a state of the array (one old value and one new value) that should never happen
- This occurred because we were trying to optimise execution and not stall all our stores because one part of memory was busy
- If we want to do such things we must define the **consistency model** that we are implementing and understand the consequences of it

Sequential Consistency

- *"the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"*
- Defined by Leslie Lamport in 1979
- Basically prohibits any out-of-order memory operations
- But is too restrictive – doesn't allow optimisation

A Possible Compromise

- In previous example
  - Thread 1 acquires the lock
  - Thread 1 writes to the array (multiple elements)
  - Thread 1 releases the lock
  - Thread 2 acquires the lock and reads the data
- No other thread can read the array until the lock is released by Thread 1
- Order of writes to the array elements doesn't matter, as long as they have all completed by the time the lock is released

Release Consistency

- In systems where access to shared data is protected by a synchronisation mechanism which ensures atomicity of critical sections
- Writes within the critical section can complete in any order when viewed externally (they still need to appear in correct order internally)
- However, they must have all completed before the synchronisation mechanism is released
- This allows optimisation within (possibly large) critical sections

Memory Fence Instructions

- Processors which allow out-of-order stores – which include many recent x86 architectures – provide special ‘memory fence’ instructions
- These vary in exact detail
- The simplest to understand is a ‘full memory fence’
- This can be inserted in an instruction sequence
- It guarantees that all loads and stores before the fence complete before any following it

Use of Memory Fence in Example Code

Thread 1

```
get lock
str r1, [r3,#0]
str r2, [r3,#4]
fence
str #1, lock
```

- Now the first two stores must complete before the third (in example architecture, will flush the store buffers)

Notes

- This has been a simplified explanation of memory consistency
- Out-of-order memory operations, in pursuit of optimisation, can occur at various levels in a modern processor (i.e. it is not just memory write buffers)
- There are several other consistency models which can be made to work – depending on the programming model
- Important to realise that consistency is **not** the same as (cache) coherence …

Notes (continued)

- Coherence vs. Consistency
  - Coherence is to do with a store to a single memory location – it ensures that all cores see the latest update to a particular location
  - Consistency is to do with ensuring that the overall memory state across multiple operations, executed concurrently, is that intended by the program
  - Consistency may involve a combination of both hardware and software (programming model)
- Relaxed (as opposed to sequential) consistency is important for modern multi-core performance

Next Lecture

- Improved performance is increasingly being devolved to ‘attached processors’ – add-on devices such as the Intel MIC, ARM’s Mali, and general purpose graphics processing units (GP-GPUs) such as NVIDIA’s Tesla, Fermi, Kepler, Maxwell, Pascal…
- The next lecture will introduce this kind of device and the resulting parallel programming model of NVIDIA