Memory access ordering - an introduction

Posted by **ARM Leif**,
22 March 2011

I recently gave a presentation at the Embedded Linux Conference Europe 2010 called *Software implications of high-performance memory systems*. This title was my sneaky (and fairly successful) way to get people to attend a presentation really about memory access (re)ordering and barriers. I would now like to follow that up with a few posts on the topic. In this post, I will be introducing a few concepts and explain the reasons behind them. In future posts, I will follow up with some practical examples.

The Sequential Execution Model

In the Good Old Days, computer programs behaved in practice pretty much the way you might instinctively expect them to from looking at the source code:

- Things happened in the way specified in the program.
- Things happened in the order specified in the program.
- Things happened the number of times specified in the program (no more, no less).
- Things happened one at a time.

In modern computer architecture, this nostalgic fantasy is sometimes referred to as the *Sequential Execution Model*. In order for existing programs and programming models to remain functional, even the most extreme modern processors will attempt to preserve the illusion of Sequential Execution from within the executing program. However, underneath your feet a lot of things will be going on that cannot be hidden from outside the processor.

Reality

The reality, however, is that in order to improve performance (both with regards to speed and power) a lot of optimizations are being performed at many different levels of your system.

Optimizing Compilers

An optimizing compiler can heavily refactor your code in order to hide pipeline latencies or take advantage of microarchitectural optimizations. It can decide to move a memory access earlier in order to give it more time to complete before the value is required, or later in order to balance out the accesses through the program. In a heavily-pipelined processor, the compiler might in fact rearrange all kinds of instructions in order for the results of previous instructions to be available when their results are required.

A classic example which is commonly used to explain the problem is the following:

```
int flag = BUSY;
int data = 0;

int somefunc(void)
{
        while (flag != DONE);

        return data;
}

void otherfunc(void)
{
        data = 42;
        flag = DONE;
}
```

Assume the above code runs in two separate threads, with thread A calling `otherfunc()` to update a value and indicate a completed operation and thread B calling `somefunc()` to wait until the signal of completion arrives before returning the value of variable `data`. There is nothing in the C language specification that guarantees that `somefunc()` will not generate code that reads `data` before it begins polling the value of `flag`. This means that it is perfectly legal for `somefunc()` to return either 0 or 42. While there are some possibilities to work around this for the code generation, this would still do nothing to prevent reordering done in hardware (see below).

Multi-issuing

Many modern processors support issuing (and executing) multiple instructions every clock cycle. Even if you explicitly placed an assembly instruction after another, they can end up being issued and executed in parallel.

Imagine the following instruction sequence in ARM assembly:

Cycle	Issue
0	`add r0, r0, #1`
1	`mul r2, r2, r3`
2	`ldr r1, [r0]`
3	`mov r4, r2`
4	`sub r1, r2, r5`
5	`str r1, [r0]`
6	`bx lr`

On a dual-issuing processor, this sequence might actually execute as this:

Cycle	Issue0	Issue1
0	`add r0, r0, #1`	`mul r2, r2, r3`
1	`ldr r1, [r0]`	`mov r4, r2`
2	`sub r1, r2, r5`	`*stall*`
3	`str r1, [r0]`	`bx lr`

In this example, nothing is issued by Issue1 in cycle 2, since the subsequent instruction requires the result from the **sub** that is issued into Issue0. This instruction is instead issued in cycle 3 – in parallel with the subroutine return.

Out-of-order execution

The first ARM processor to support out-of-order execution was the ARM1136J(F)-S™, which permitted non-dependent load and store operations to complete out of order with each other. In practise, this means that a data access that misses in the cache can be overtaken by other data accesses that hit (or miss) in the cache, as long as there are no data dependencies between them. It also permitted load-store instructions to complete out-of-order with data processing instructions where there were no data dependencies (for example where a load provided the address for a subsequent load or store).

Fast-forward a few years to the Cortex™-A9. This processor supports out-of-order execution of most non-dependent instructions in many situations. When an instruction stalls because it is waiting for the result of a preceding instruction, the core can continue executing subsequent instructions that do not need to wait for the unmet dependencies.

Consider the following code-snippet, which has a couple of instructions that could potentially take more than one cycle before the result is available to subsequent instructions. Both a **mul** and a **ldr** can on several architectures require multiple cycles before their results are available. In this case we assume 2 cycles for each.

```
        add     r0, r0, #4
        mul     r2, r2, r3
        str     r2, [r0]
        ldr     r4, [r1]
        sub     r1, r4, r2
        bx      lr
```

If we execute this code on an in-order processor, the execution will look something like the following:

Cycle	Issue
0	add r0, r0, #4
1	mul r2, r2, r3
2	*stall*
3	str r2, [r0]
4	ldr r4, [r1]
5	*stall*
6	sub r1, r4, r2
7	bx lr

While if we execute it on an out-of-order processor, we might see something more like:

Cycle	Issue
0	`add r0, r0, #4`
1	`mul r2, r2, r3`
2	`ldr r4, [r1]`
3	`str r2, [r0]`
4	`sub r1, r4, r2`
5	`bx lr`

By permitting the **ldr** to execute while we wait for the **mul** to complete so that the **str** can progress, we have also given more time for the **ldr** to complete before its value is needed.

Speculation

Speculation can be described simply as the core executing, or beginning to execute, an instruction before it knows whether that particular instruction should be executed or not. This means that the result will be available sooner if conditions resolve such that the speculation was correct. Examples of this would be when the code makes use of general-purpose conditional execution in either the ARM or Thumb instruction sets, or when the core encounters a conditional branch instruction. The core can then *speculatively* execute the conditional instruction, or instructions following the conditional branch instruction. If it does so, it must ensure that any signs of it doing this are erased if the speculation turns out to have been incorrect.

Where memory load instructions are concerned, speculation can go a bit further. A load from a cacheable location can be speculatively issued, and this can in turn lead to that location being copied in from external memory, potentially evicting an existing cache line. Many modern processors go one step further and monitor data accesses performed in order to detect patterns, and bring subsequent addresses in that pattern into the cache even before the instruction enters the processor pipeline.

Load-Store Optimizations

External memory accesses in a high-performance system tend to have significant latencies – even if the core:memory clock ratio is as good as 2:1, the actual data transfer can only take place after many cycles of setup times. This can be anywhere between 5–50 bus clock cycles, depending mainly on the power profile of the system – with caches or bridges on the way adding additional cycles on top of this. With a less favorable core-to-memory clock ratio, this effect is multiplied.

In order to reduce the effect of these latencies, processors very much attempt to optimize their memory accesses to (if possible) reduce the number of transactions by writing more data with each transaction – using bursts to transfer longer streams of data with only the latencies of a single transaction. This can for example mean that multiple writes to buffered memory can be merged together into one transaction.

Cache Coherent Multi-Core

When using a multi-core processor, hardware-based cache coherency management can cause cache lines to migrate transparently between cores. This can potentially cause different cores to see updates to cached memory locations in different order.

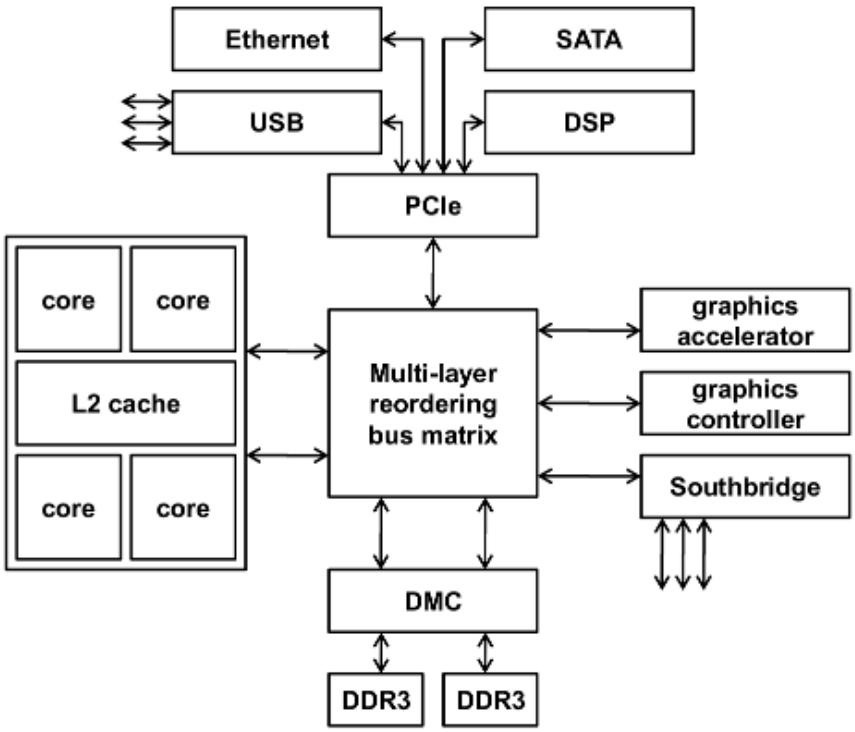
To give a concrete example of this: the example of `somefunc()` and `otherfunc()` above has another potential aspect when executing in a multi-core SMP system. If the two threads were executing on

separate cores, then the combination of hardware cache coherency management, speculation and out-of-order execution means that the order of memory accesses *may be seen differently by the different cores*.

Simply put, hardware cache coherency management means that cache lines can be moved across between the cores to be available wherever they are accessed. Since out-of-order capable processors can load one memory location from the cache while waiting for the result of another load (or store) to complete, it would be perfectly permissible for the core executing `somefunc()` to speculatively load the value of `data` before the value of `flag` actually changed to `DONE` - even if this was not the order in which the instructions were within the compiled application.

External Memory Systems

The complexities continue even once you get out into the external memory system. To achieve high performance in a system with many bus masters (agents), the interconnect might be configured in a multi-layer system. This effectively means that different agents (or *masters*) can have different routes to the various devices (or *slaves*) in the system. Also, peripherals like memory controllers might have more than one slave interface, allowing multiple agents to access it simultaneously.



Finally, memory transactions that are permitted to be buffered can be so at pretty much any point along the way, and potentially more than once. This can cause accesses from different masters to take different amounts of time to complete on different occasions.

Conclusion

In modern computer systems, a lot of things happen in a different order than one might intuitively assume – and not every agent in the system will agree on what that order is. In following posts I will cover what this will mean in practice and what you might need to do about it.

References

Kourosh Gharachorloo - MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS (PDF)

UPDATE:
Memory Access Ordering Part 2 - Barriers and the Linux Kernel
Memory Access Ordering Part 3 - Memory Access Ordering in the ARM Architecture

Leif, Principal Software Engineer, ARM, he first started playing around with UNIX in 1994, installing Linux in 1995, in 1999 graduated to Linux on embedded systems, and moved to doing it for a living in 2001. After four years of that, he packed up and moved to Cambridge, UK, to work for ARM. He is currently working on improving ARM multi-core support in (mostly) open source software. He is way too fond of blue LEDs (but any color will do if they are flashing), and is cowardly refusing to pick sides in the Linux/BSD war.