Software implications of high-performance memory systems

Leif Lindholm

ARM Ltd.

Embedded Linux Conference Europe 2010

Alternative titles

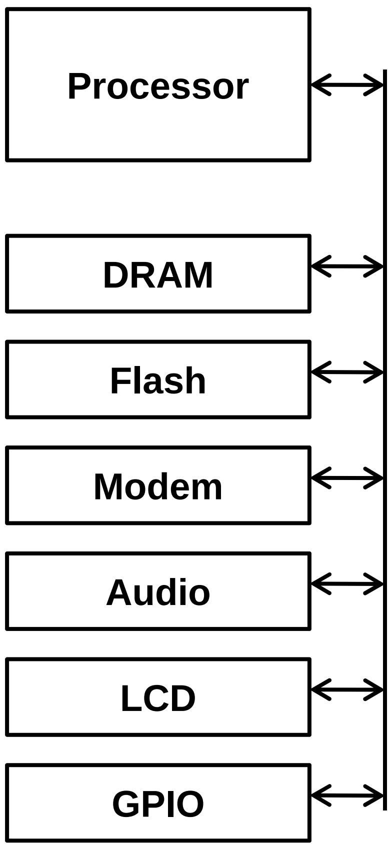
- Barriers – the what, the how, and the by all that is holy why?
- What you're going to wish you didn't know about modern computer systems if you don't already
- Of course it couldn't do that! …could it?

Overview

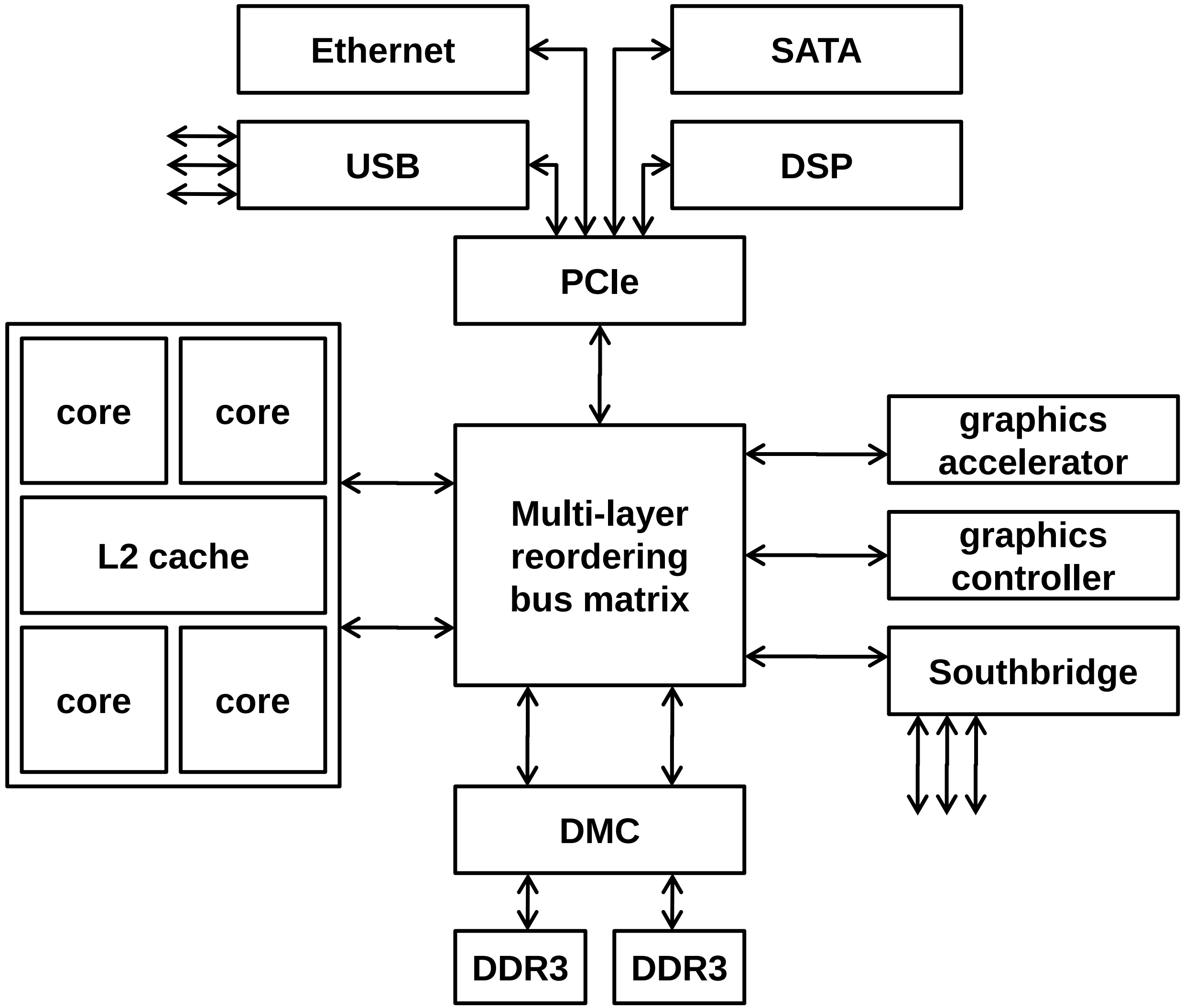
- This presentation aims to explain some of what goes on underneath your feet when developing software for modern computer systems.
- The good news is that if you are an application developer, you normally don’t need to be aware of this. Congratulations!
- The bad news is that if you are developing or debugging kernel code, drivers, system libraries, execution environments, JIT compilers, …, you do. Sorry.

It all used to be so simple

- Single core
  - In-order
  - Single-issue
  - No speculation
  - No caches?
- Only slave peripherals
  - No DMA
- Simple operating systems
  - Bare metal?
- Few, if any, user-accessible expansion ports



The world today



The world today

- Multi-core processors
- Speculative multi-issue out-of-order cores
- Multiple levels of caches
  - Some with hardware coherency management
- Multi-layered bus interconnects
- Memory access merging (reads and writes)
- Many agents/bus-masters in system
- End-user accessible expansion busses
- Highly optimizing compilers

- Most of this available today in devices amusingly still referred to as phones, as well as set-top-boxes, TVs, …

So what does all that stuff mean in practise?

In the good old days…

- Things happened in the way specified by the program
- Things happened the number of times specified in the program (no more, no less)
- Only one thing happened at once

- This is now referred to as "the sequential execution model"
  - For software to work at all, this model must still appear to be in place within the scope of a single process executing on a single core
    - But throw in some SMP and the world changes…
    - And this will not necessarily be true for an external observer comparing the bus traffic to the program code

Multi-issue (superscalar)

- More than one instruction can be issued per clock cycle, where not prevented by data-dependencies
- Offers new and exciting ways for compilers to improve code performance by shuffling instructions around

```
1    add     r0, r0, #1
2    mul     r2, r2, r3
3    load    r1, [r0]
4    mov     r4, r2
5    sub     r1, r2, r5
6    store   r1, [r0]
7    return
```

Executing on a dual-issue core

```
1    add     mul
2    load    mov
3    sub     *stall*
4    store   return
```

Speculation

- The core executes things before it is determined if they are actually meant to execute
  - Pretending that nothing happened if it turns out the speculation was not the actual case

```
          add     r0, r0, #1
          cmp     r0, #42
          bne     skip
          load    r1, [r2]
          b       proceed
skip:     load    r1, [r3]
proceed:  store   r1, [r4]
```

- The core fetches code or data it determines might be used soon into cache ahead of time (prefetching)

Out-of-order execution

- When core detects an unresolved data-dependency preventing it from issuing an instruction, it just issues the next instruction instead of stalling waiting for the result to come back
  - Continues executing until there are no non-dependent operations available

```
1    add     r0, r0, #1
2    mul     r2, r2, r3
3    store   r2, [r0]
4    load    r4, [r1]
5    sub     r1, r4, r2
6    return
```

In-order

```
1    add     r0, r0, #1
2    mul     r2, r2, r3
     *stall*
3    store   r2, [r0]
4    load    r4, [r1]
     *stall*
5    sub     r1, r4, r2
6    return
```

Out-of-order

```
1    add     r0, r0, #1
2    mul     r2, r2, r3
4    load    r4, [r1]
3    store   r2, [r0]
5    sub     r1, r4, r2
6    return
```

Coherency-managed SMP

- Lines can migrate between (data) caches at any time
- Write buffers can affect externally visible ordering of memory accesses (between cores as well as in the outside system).

```
send_ipi: (core0)
        load     r3, #IPI_ID
        store    r2, [r1]          @ set payload
        store    r3, [r0]          @ send IPI
```

```
recv_ipi: (core1)
        load     r1, [r0]
        cmp      r1, #VALUE        @ should contain what
                                   @ was in r2 on core0
```

External masters

- Typical use of a DMA controller:
  - You write a bunch of data into a shared buffer, and clean your caches after completion if using cached memory
  - Then you signal the DMA controller to start transferring
  - Things will work a whole lot better if the DMA controller sees these operations in this order
- Using a DSP to do video decode into a shared buffer?

And let's not forget the compilers

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

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

        return data;
}

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

- Ignoring all of the magic I've mentioned underneath the hood, what would you expect `somefunc()` to return?
  - 42, yes, that's possible.
  - So is 0.

All in all

- Reading architecture specifications these days, you frequently come across interesting terms and phrases like:
  - …is observed to…
  - …must appear to…
- The comfy world of sequential execution is no more. One must now think of whether the effect of an instruction can be detected rather than if it has "executed"
  - If you dual-issue a NOP with an ADD … does it take any time to execute?
- Where correct operation requires something to appear in the same order to multiple agents, this must be explicitly ordered

So how come anything actually works?

How come anything works?

- Because within each core, the sequential execution model must still (appear to) hold true
  - Dependent/overlapping accesses cannot be reordered*

```
void somefunc(void)
{
        unsigned char *cptr = iptr;
        *iptr = 0x12345678;
        cptr[1] = 0xff;
}
```

- Because of barriers

- And because library functions that require barriers for correct operation already use them where necessary

Barrier and Fence instructions

- Barriers make it possible to write software that actually works
- Instructions that explicitly order memory accesses
  - Prevent reordering of any memory accesses past the barrier
  - Prevent reordering of specific memory accesses past the barrier
  - Ensure synchronization between data and instruction side
  - Ensure synchronization between instruction stream and memory accesses

Architecture	Barriers
Alpha	IMB, MB, WMB
ARMv7	DMB, DSB, ISB
IA64	MF
PPC	SYNC, LWSYNC, EIEIO
x86/AMD64	LFENCE, MFENCE, SFENCE

Compiler barriers

- An optimizing compiler is free to reorder non-volatile memory accesses in any way it sees fit in order to improve performance.
  - And remember `Documentation/volatile-considered-harmful.txt`

```
while (*hold == 1);
return *ret;
```

```
        load    r0, [ret]
1:      load    r1, [hold]
        cmp     r1, #1
        beq     1b
        b       LR
```

- This can be prevented by introducing a compiler scheduling barrier:
  - `barrier()` defined in `include/linux/compiler-gcc.h`
    - `#define barrier() __asm__ __volatile__("": : :"memory")`

Linux generic memory barriers

- Linux defines a set of generic memory barrier macros, common both to SMP and uniprocessor systems

- Since the DEC Alpha had the weakest memory model of all platforms in the kernel, this became the template for the architecture-independent model within Linux
  - "If it works on the Alpha, it'll work anywhere"

- Guaranteed to be at minimum a compiler barrier
  - But where architecturally required, it will output the necessary barrier instruction

Macro	Functionality
mb()	No memory accesses can overtake.
rmb()	No reads can overtake.
wmb()	No writes can overtake.

Linux SMP memory barriers

- Linux also defines a set of barriers that ensure correct operation in SMP systems – in practise where hardware coherency management is in place
- Only guaranteed for cached memory, system bus effects ignored
- NOT a superset of generic barriers – usually weaker
- Turned into compiler barriers when CONFIG_SMP is not enabled

Macro	Functionality
smp_mb()	No memory accesses can overtake.
smp_rmb()	No reads can overtake.
smp_wmb()	No writes can overtake.

Read dependency barriers

- *The DEC Alpha processor amazingly permitted reordering dependent loads
  - The read_barrier_depends() macros were introduced to deal with this – these turn into NULL statements on all other architectures (not even compiler barriers)

Macro	Functionality
read_barrier_depends()	Ensures values from previous reads are usable.
smp_read_barrier_depends()	Ensures values from previous reads are usable.

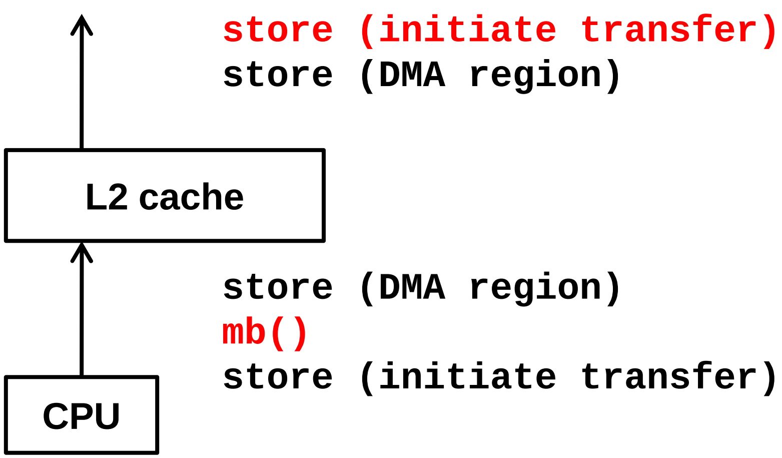
mmiowb()

- `mmiowb()` forces global ordering of memory mapped I/O accesses

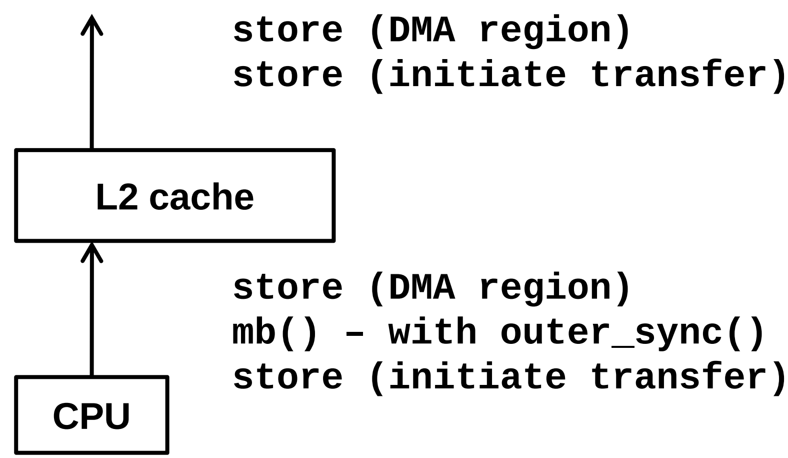
Macro	Functionality
mmiowb()	Synchronize I/O globally.

outer_sync()

- When barriers only reach the external bus interface of the processor, the interconnect can still reorder bufferable memory accesses
  - Cortex-A9 does not have an integrated Level 2 cache – most implementations supplemented with external PL310 controller.
  - ARM-specific `outer_sync()` macro included in `mb()` when DMA memory treated as bufferable
    - `arch/arm/include/asm/outercache.h`



mb() without outer_sync()



mb() with outer_sync()

Shameless marketing

- Cortex-A15 has an integrated L2 cache, but also implements external bus interfaces following the new AMBA4 AXI specification

- These AMBA4 AXI interfaces also support ACE (AMBA Coherency Extensions)
  - ACE includes support for having the interconnect propagating barriers
  - Barriers can be specified with a limit. Backwards-compatible in that unimplemented barrier variants will execute as System-wide barriers.
    - Non-shareable (NSH)
    - Inner-shareable (ISH)
    - Outer-shareable (OSH)
    - System-wide (SY)

I/O accessors

- A long thread ("USB mass storage and ARM cache coherency") spanned several kernel lists earlier this year
  - Uncovered that actually quite a few drivers do not really use barriers everywhere they should be
  - The pragmatic solution was to add barriers to ARM I/O accessors
    - `read{b,w,l}()`
    - `write{b,w,l}()`
    - `ioread{8,16,32}()`
    - `iowrite{8,16,32}()`

Synchronization primitives

- `spin_{lock,unlock}()` contain `smp_mb()`
  - This ensures ordering between acquiring the lock and accessing the protected resource, and between modifying the resource and releasing the lock

- `atomic_{inc,dec,add,sub}()` make no such promises

In summary

- So, barriers are great – I should put `mb()` everywhere just to make sure?
  - Well, no … barriers are sometimes required to make software work as expected, but they do come at a cost
    - An `smp_rb()` might have no visible impact even on an SMP system, whereas an `mb()` can force an `outer_sync()` as well as forcing a drain of the write buffer.
    - Always use the weakest barrier possible – even if there is no noticeabe difference on your current platform between using `smp_rmb()` or `mb()`, that is not necessarily the case for other platforms. Some of which you might be using in your next project.

References

- `Documentation/memory-barriers.txt`
- Paul E. McKenney - Memory Barriers: a Hardware View for Software Hackers http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf
  - Evolution of “Memory Ordering in Modern Microprocessors” LJ articles
- Kourosh Gharachorloo - Memory consistency models for shared-memory multiprocessors
- Barrier Litmus Test and Cookbok - infocenter.arm.com