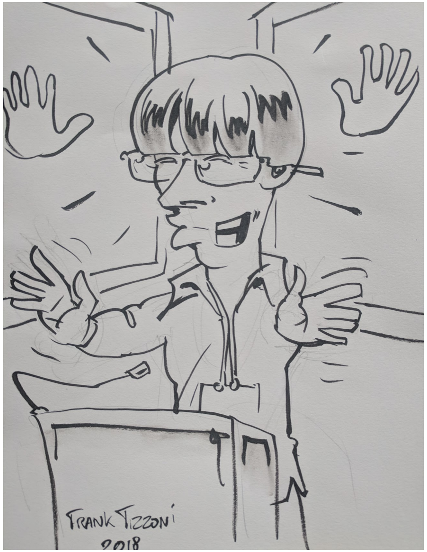
Uh-oh; it’s I/O ordering!

ELCE, Edinburgh

Will Deacon <will.deacon@arm.com>

October, 2018

$ whoami



- Co-maintainer of `arm64` architecture, `ARM` perf backends, `SMMU` drivers, atomics, locking, memory model, TLB invalidation…
- Developer in the Open-Source Software group at Arm
- Close working relationship with Architecture and Technology Group
- Co-author of Armv8 architectural memory model
- Involved in C/C++ memory model working group
- Spoke at ELCE '13 about memory ordering

This time, I'm going to talk about I/O ordering.

My idea of paradise

A tropical desert island?

My idea of paradise

A uniprocessor tropical desert island!

The grim reality

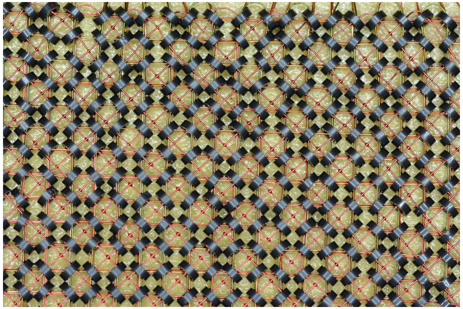
In reality, we cram thousands of CPUs together in air-conditioned warehouses deprived of natural light and attach them all to a network.

So much for our island dreams.

Challenges of concurrency

Even with a single, coherent, shared memory (like you might expect for CPUs!); concurrency is hard:

Reasoning	about programs is no longer ‘stepwise’
Reordering	of memory accesses
Heisenbugs	which disappear when instrumented
Performance	is balanced against correctness
Limited tools	to validate code



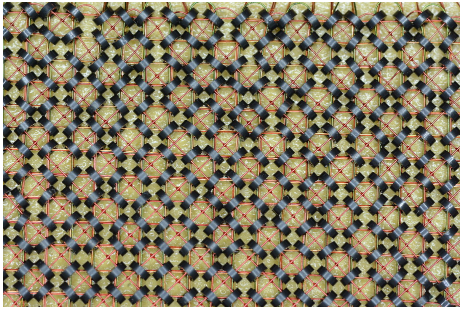
In other words, the CPU doesn’t actually do what you ask it to do.

Can it really get worse than this?

Challenges of concurrency

Even with a single, coherent, shared memory (like you might expect for CPUs!); concurrency is hard:

Reasoning	about programs is no longer 'stepwise'
Reordering	of memory accesses
Heisenbugs	which disappear when instrumented
Performance	is balanced against correctness
Limited tools	to validate code



In other words, the CPU doesn't actually do what you ask it to do.

Can it really get worse than this? Of course it can ;)

The theory: memory consistency models (in 5 minutes)

Example: store buffering

Initially, `*x` and `*y` are 0 in memory; `foo` and `bar` are local (register) variables:

CPU0

```
a: WRITE_ONCE(*x, 1);
b: foo = READ_ONCE(*y);
```

CPU1

```
c: WRITE_ONCE(*y, 1);
d: bar = READ_ONCE(*x);
```

What are the permissible values for `foo` and `bar`?

Example: store buffering

Initially, `*x` and `*y` are 0 in memory; `foo` and `bar` are local (register) variables:

CPU0

```
a: WRITE_ONCE(*x, 1);
b: foo = READ_ONCE(*y);
```

CPU1

```
c: WRITE_ONCE(*y, 1);
d: bar = READ_ONCE(*x);
```

What are the permissible values for `foo` and `bar`?

All production architectures permit `foo == bar == 0`.

Example: store buffering

Initially, `*x` and `*y` are 0 in memory; `foo` and `bar` are local (register) variables:

CPU0

```
a: WRITE_ONCE(*x, 1);
b: foo = READ_ONCE(*y);
```

CPU1

```
c: WRITE_ONCE(*y, 1);
d: bar = READ_ONCE(*x);
```

What are the permissible values for `foo` and `bar`?

All production architectures permit `foo == bar == 0`. How?

Lies, damned lies and sequential consistency

CPU0	CPU1	Interleavings
`a: WRITE_ONCE(*x, 1);`	`c: WRITE_ONCE(*y, 1);`	`{a,b,c,d}`
`b: foo = READ_ONCE(*y);`	`d: bar = READ_ONCE(*x);`	`{c,d,a,b}`
		`{a,c,b,d}`
		`...`

'A multiprocessor is sequentially consistent if 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.'
– Leslie Lamport (1979)

Sequential consistency (SC) is 'easy' to reason about, as there is a single global ordering consistent with program order for each thread.

Lies, damned lies and sequential consistency

CPU0	**CPU1**	**Interleavings**
`a: WRITE_ONCE(*x, 1);`	`c: WRITE_ONCE(*y, 1);`	`{a,b,c,d}`
`b: foo = READ_ONCE(*y);`	`d: bar = READ_ONCE(*x);`	`{c,d,a,b}`
		`{a,c,b,d}`
		`...`

‘A multiprocessor is sequentially consistent if 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.’

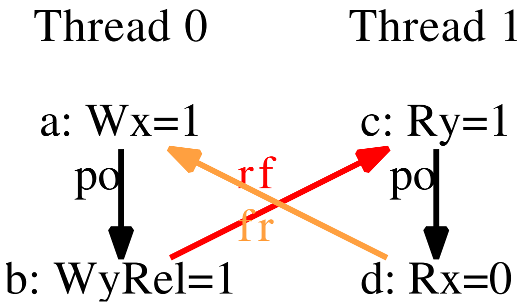
– Leslie Lamport (1979)

Sequential consistency (SC) is ‘easy’ to reason about, as there is a single global ordering consistent with program order for each thread.

It also tells us that `foo == bar == 0` is forbidden in the previous example.

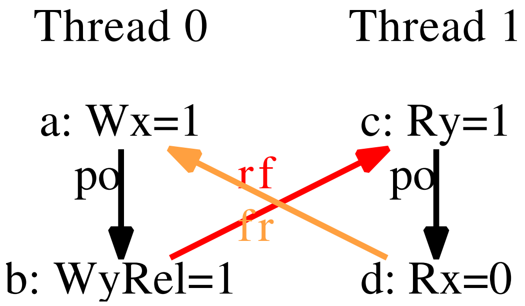
Litmus tests

```
AArch64 MP+popl+po
"PodWWPL RfeLP PodRR Fre"
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0             | P1          ;
 MOV W0,#1      | LDR W0,[X1] ;
 STR W0,[X1]    | LDR W2,[X3] ;
 MOV W2,#1      |             ;
 STLR W2,[X3]   |             ;
exists
(1:X0=1 /\ 1:X2=0)
```



Litmus tests

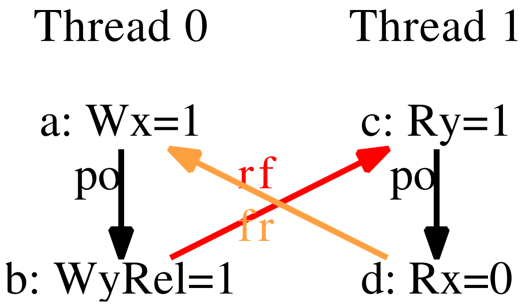
```
AArch64 MP+popl+po
"PodWWPL RfeLP PodRR Fre"
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0             | P1          ;
 MOV W0,#1      | LDR W0,[X1] ;
 STR W0,[X1]    | LDR W2,[X3] ;
 MOV W2,#1      |             ;
 STLR W2,[X3]   |             ;
exists
(1:X0=1 /\ 1:X2=0)
```



Remember: cycles are bad!

Litmus tests

```
AArch64 MP+popl+po
"PodWWPL RfeLP PodRR Fre"
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0              | P1          ;
 MOV W0,#1       | LDR W0,[X1] ;
 STR W0,[X1]     | LDR W2,[X3] ;
 MOV W2,#1       |             ;
 STLR W2,[X3]    |             ;
exists
(1:X0=1 /\ 1:X2=0)
```



Remember: cycles are bad!

A memory model tells you which ones to worry about.

Beyond shared memory communication

Out-of-band communication and side-effects

Not all communication between observers is via explicit accesses to shared memory:

- IPI using interrupt controller
- DMA using a peripheral
- Page-table modifications
- Clocks and regulators
- Passing of time

$$\begin{tikzcd}
\text{Thread 0} & \text{Thread 1} \\
\text{a: Wx=1} \arrow[d, "po"'] & \langle\text{interrupt}\rangle \arrow[d, "po"'] \\
\text{b: WirqcRel=1} & \text{c: Rx=0} \arrow[ul, "fr"']
\end{tikzcd}$$

These interactions are generally considered out-of-scope by memory models and rely on implementation-specific details!

Generalise to multiple endpoints

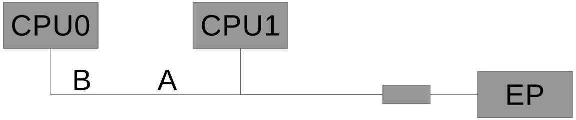
Redefine inter-processor communication by considering accesses to endpoints:

An access	is an event targetting a specific endpoint which can cause it to change state
An endpoint	is a piece of hardware with mutable state that can respond to accesses, or generate accesses targetting other endpoints

- For us, endpoints are either memory or an MMIO interface (i.e. `__iomem *`)
- Accesses are load/store operations, using appropriate accessor functions

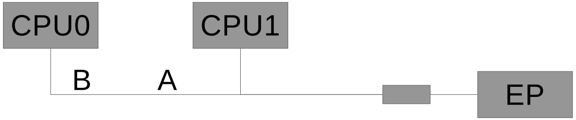
Ordering vs completion

Ordering requires that two accesses to the same endpoint will be remain in-order on their way to that endpoint.



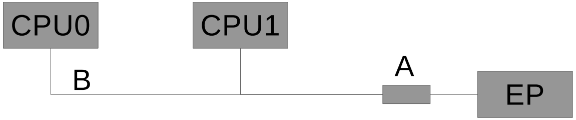
Ordering vs completion

Ordering requires that two accesses to the same endpoint will be remain in-order on their way to that endpoint.



Completion requires that a prior access reaches a certain point before initiating a later access:

Reads	complete when they have their data, so they appear to complete at the endpoint
Writes	can be buffered/merged and therefore may complete early at the *point of serialisation* (e.g. posted write)



The practice: I/O ordering in Linux

Caveat: assumptions

I/O ordering is like a melting pot of other memory models:

- The CPU architecture provides software mechanisms for ordering
- A bus/interconnect has its own ordering rules (e.g. AXI, PCI)
- These worlds are bridged together until they hit an endpoint
- Endpoints can have their own constraints too

Linux assumes some basic sanity such as a point of coherence and the ability to enforce ordering in the ISA (i.e. not IMP DEF magic). Correct bridging is crucial!

DMA buffers are allocated via `dma_alloc_coherent` or mapped using the streaming API. Devices are either coherent or they aren't.

MMIO regions are mapped using `ioremap()`, which requires aligned access and guarantees atomicity, access size and lack of speculation. `ioremap_wc()` is weaker (more like memory) and `ioremap_nocache()` is stronger (no buffering).

Default I/O accessors

Dereferencing an `__iomem *` must use a suitable I/O accessor:

`inX/outX` Legacy x86 port I/O access instructions

`readX/writeX` MMIO accessors

`ioreadX/iowriteX` Expand to appropriate underlying accessors

- Little-endian by default
- Ordered against other accesses to the same endpoint: reads can ‘push’ writes
- Write accessor initiates after completing prior memory writes
- Read accessor completes before initiating later memory reads and `delay()` loops

If you’re crazy, can inter-operate with `spinlock_t` using `mmiowb()`.

Very expensive on non-x86 architectures!

Relaxed accessors

Not all (most?) MMIO accesses are related to DMA:

`readX_relaxed` MMIO read access

`writeX_relaxed` MMIO write access

`readsX/writesX, ioreadX_rep/iowriteX_rep, insX/outsX` String accessors

Do not provide completion guarantees wrt accesses to memory!

Like the default accessors, `_relaxed` accesses remain ordered to the same endpoint.

Practically, they will also work with `spinlock_t`.

Mandatory barriers

Fine-grained control over completion guarantees using expensive barrier macros:

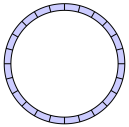
Barrier	Completes prior	Before initiating later
`mb()`	Reads/writes	Reads/writes
`rmb()`	Reads	Reads
`wmb()`	Writes	Writes

Can even be used in conjunction with `_relaxed` I/O accessors:

```
writel() => wmb(); writel_relaxed()
writel_relaxed(); mb(); READ_ONCE()
```

Generally don't need these if you're using the default accessors for regular DMA

DMA barriers



Provide ordering guarantees for CPU accesses to DMA buffers (i.e. `dma_alloc_coherent()` allocations):

`dma_rmb()` Order reads from a DMA buffer

`dma_wmb()` Order writes to a DMA buffer

- Useful for coherent descriptor rings, where the descriptor payload must be read or written in a specific order relative to its header.
- Relatively cheap, even if the underlying device isn't cache coherent.
- No effect on `__iomem` accesses

Examples

Trigger DMA read

`drivers/iommu/arm-smmu-v3.c`: Submitting a command to the SMMU

```
// queue_write()
for (i = 0; i < n_dwords; ++i)
    *dst++ = cpu_to_le64(*src++);

// queue_inc_prod()
u32 prod = (Q_WRP(q, q->prod) | Q_IDX(q, q->prod)) + 1;
q->prod = Q_OVF(q, q->prod) | Q_WRP(q, prod) | Q_IDX(q, prod);
writel(q->prod, q->prod_reg);
```

Process DMA write

`drivers/net/ethernet/marvell/mvneta.c`: Reading RX data

```
// mvneta_rxq_busy_desc_num_get()
u32 val = mvreg_read(pp, MVNETA_RXQ_STATUS_REG(rxq->id)); // readl
return val & MVNETA_RXQ_OCCUPIED_ALL_MASK;

// mvneta_rx_swbm
int rx_todo = mvneta_rxq_busy_desc_num_get(pp, rxq);
while ((rcvd_pkts < budget) && (rx_proc < rx_todo)) {
    struct mvneta_rx_desc *rx_desc = mvneta_rxq_next_desc_get(rxq);
    index = rx_desc - rxq->descs;
    page = (struct page *)rxq->buf_virt_addr[index];
    data = page_address(page);
    memcpy(rxq->skb->data, data + MVNETA_MH_SIZE, copy_size);
```

Batch device configuration

`drivers/gpu/drm/mediatek/mtk_disp_rdma.c`: Configure DMA parameters

```
// mtk_rdma_layer_config()
writel_relaxed(con, comp->regs + DISP_RDMA_MEM_CON);
writel_relaxed(addr, comp->regs + DISP_RDMA_MEM_START_ADDR);
writel_relaxed(pitch, comp->regs + DISP_RDMA_MEM_SRC_PITCH);
writel(RDMA_MEM_GMC, comp->regs + DISP_RDMA_MEM_GMC_SETTING_0);
```

People tend to get this wrong and add `wmb()`s!

Delay-based device configuration

`drivers/soc/qcom/cpu_ops.c:` Bringing up L2 and SCU…

Take a deep breath…

Delay-based device configuration

`drivers/soc/qcom/cpu_ops.c`: Bringing up L2 and SCU…

Take a deep breath…

```
/* De-assert L2/SCU Logic reset */
writel_relaxed(0x100203, l2_base + L2_PWR_CTL);
mb();
udelay(54);

/* Turn on the PMIC_APC */
writel_relaxed(0x10100203, l2_base + L2_PWR_CTL);
```

How would you fix this code? (don’t worry, it’s not in mainline)

DMA descriptor rings

`drivers/infiniband/hw/bnxt_re/qplib_fp.c:` Polling in-memory notification queue

```
// bnxt_qplib_service_nq()     [tasklet]
while (budget--) {
    nqe = &nq_ptr[NQE_PG(sw_cons)][NQE_IDX(sw_cons)];
    if (!NQE_CMP_VALID(nqe, raw_cons, hwq->max_elements))
        break;

    /* The valid test of the entry must be done first before
     * reading any further. */
    dma_rmb();

    type = le16_to_cpu(nqe->info10_type) & NQ_BASE_TYPE_MASK;
```

PIO

`drivers/net/ethernet/smsc/smc911x.c`: Reading from/writing to MMIO FIFO

```
#define SMC_insl(lp, r, p, l)    \
        ioread32_rep((int*)((lp)->base + (r)), p, l)
#define SMC_PULL_DATA(lp, p, l) \
        SMC_insl ( lp, RX_DATA_FIFO, p, (l) >> 2 )

#define SMC_outsl(lp, r, p, l)   \
        iowrite32_rep((int*)((lp)->base + (r)), p, l)
#define SMC_PUSH_DATA(lp, p, l) \
        SMC_outsl( lp, TX_DATA_FIFO, p, (l) >> 2 )

SMC_PULL_DATA(lp, data, pkt_len+2+3); // smc911x_rcv()
SMC_PUSH_DATA(lp, buf, len); // smc911x_hardware_send_pkt()
```

The read-triggered DMA challenge!

- Some adaptec card rumoured to do this
- Makes little sense from h/w perspective (reads are slow)
- I couldn't find anything in the tree
- Would require explicit `mb()` before the MMIO read

Please let me know if you find any examples!

Questions?

References

- Desert island – By Timo Newton-Syms from Helsinki, Finland and Chalfont St Giles, Bucks, UK - Desert Island, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=26292873
- Alpha CPU – CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=91624
- Server racks – By CSIRO, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=35458082
- Magnetic core – By Bubba73 (Jud McCranie) - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=39746489
- PCI cards – By Hannes Grobe (talk) - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=21932132
- Couple relaxing on beach – By Hector Alejandroderivative work: Danapit - This file was derived from: An old couple relaxing on the beach.jpg:, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=26487578
- Circular buffer – By I, Cburnett, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2302964
- Reward poster – By Archives New Zealand from New Zealand - Reward Poster, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=51250708