arm

Formalising the Armv8 memory consistency model

OpenSHMEM workshop, Baltimore MD

Will Deacon <will.deacon@arm.com>

August, 2018

Hello!

- MEng in Computing: interested in low-level systems programming and computer architecture
- Senior Principal Software Engineer in the Open-Source Software group at Arm
- Upstream Linux kernel maintainer for `arm64` and others
- Close working relationship with Architecture and Technology Group
- Co-author of Armv8 memory model

Hello!

- MEng in Computing: interested in low-level systems programming and computer architecture
- Senior Principal Software Engineer in the Open-Source Software group at Arm
- Upstream Linux kernel maintainer for `arm64` and others
- Close working relationship with Architecture and Technology Group
- Co-author of Armv8 memory model

I have never used `OpenSHMEM`!

Why am I here?

- Earlier this year, I gave a version of this talk to a small group of people at Oak Ridge about memory consistency models and the recent revision of the Armv8 memory model…

Why am I here?

- Earlier this year, I gave a version of this talk to a small group of people at Oak Ridge about memory consistency models and the recent revision of the Armv8 memory model…
- Afterwards, Manju told me about something called OpenSHMEM and the desire for a concrete specification of its memory model.

Why am I here?

- Earlier this year, I gave a version of this talk to a small group of people at Oak Ridge about memory consistency models and the recent revision of the Armv8 memory model…
- Afterwards, Manju told me about something called OpenSHMEM and the desire for a concrete specification of its memory model.
- It took less than 5 minutes of discussion around a whiteboard before we were both totally confused, so he suggested I came here and spoke to the community. Hopefully the same principles for the CPU can be applied to OpenSHMEM.

Why am I here?

- Earlier this year, I gave a version of this talk to a small group of people at Oak Ridge about memory consistency models and the recent revision of the Armv8 memory model…
- Afterwards, Manju told me about something called OpenSHMEM and the desire for a concrete specification of its memory model.
- It took less than 5 minutes of discussion around a whiteboard before we were both totally confused, so he suggested I came here and spoke to the community. Hopefully the same principles for the CPU can be applied to OpenSHMEM.

I have never used `OpenSHMEM`!

Background and Terminology

I’m used to dealing with CPUs running assembly programs:

- Coherent shared memory directly addressable by the instruction set with `ns` access times
- Physically small machines with <1000 h/w threads
- Single-copy and multi-copy atomicity

Background and Terminology

I'm used to dealing with CPUs running assembly programs:

- Coherent shared memory directly addressable by the instruction set with `ns` access times
- Physically small machines with <1000 h/w threads
- Single-copy and multi-copy atomicity

When I say…

load you say get

store you say put

barrier you say fence

Background and Terminology

I'm used to dealing with CPUs running assembly programs:

- Coherent shared memory directly addressable by the instruction set with `ns` access times
- Physically small machines with <1000 h/w threads
- Single-copy and multi-copy atomicity

When I say…

load	you say get
store	you say put
barrier	you say fence

When you say…

barrier	I say synchronize
non-blocking	I look blank
quiet	I shut up

Background and Terminology

I'm used to dealing with CPUs running assembly programs:

- Coherent shared memory directly addressable by the instruction set with `ns` access times
- Physically small machines with <1000 h/w threads
- Single-copy and multi-copy atomicity

When I say…

load	you say get
store	you say put
barrier	you say fence

When you say…

barrier	I say synchronize
non-blocking	I look blank
quiet	I shut up

We both need a memory consistency model to reason about the behaviour of concurrent interactions on a globally mutable state for a large body of existing code and implementations

What is a memory consistency model?

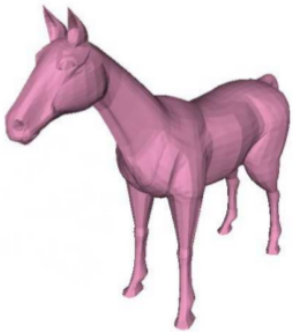
Given a potentially concurrent program, a memory (consistency) model defines the possible values returned for each read in the program as well as the final values of each location in memory.

- Illusion of *program order* for a uniprocessor system
- Barrier/fence instructions
- Dependencies
- Write propagation between threads
- The ‘order’ of loads and stores

Can be used to identify unwanted results in an under-constrained environment.

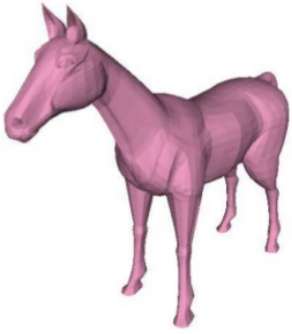
But why should I care?

Ponies



Pony courtesy of Jade Alglave

Ponies



Ponies courtesy of Jade Alglave

Failure to enforce ordering between concurrent producer and consumer.

Hardware and software are far less conservative than they used to be!

File writeback

`msync()` flushes changes made to the in-core copy of a file that was mapped into memory using `mmap(2)` back to the filesystem. – `man msync`

File writeback

`msync()` flushes changes made to the in-core copy of a file that was mapped into memory using `mmap(2)` back to the filesystem. – `man msync`

ARM64: kernel panics in DABT in `sys_msync` path – Yury Norov, LKML, Sept 2017

File writeback

`msync()` flushes changes made to the in-core copy of a file that was mapped into memory using `mmap(2)` back to the filesystem. – `man msync`

ARM64: kernel panics in DABT in `sys_msync` path – Yury Norov, LKML, Sept 2017

```
Unable to handle kernel paging request at virtual address ffffffffc0000d68
swapper pgtable: 4k pages, 48-bit VAs, pgd = ffff00000901f000
[ffffffffc0000d68] *pgd=0000000000000000
Internal error: Oops: 96000004 [#1] PREEMPT SMP
Modules linked in:
CPU: 0 PID: 9861 Comm: doio Not tainted 4.13.0-00027-g2fdc18baa2ae #196
Hardware name: linux,dummy-virt (DT)
task: ffff80000300d400 task.stack: ffff80003d28c000
PC is at check_pte+0x8/0x130
LR is at page_vma_mapped_walk+0x240/0x498
...
```


arm

Ok, so what do I need to know?

Architects, CPU vendors and programming languages have helpfully documented their memory models, so we just need to read their specifications…

Ok, so what do I need to know?

Architects, CPU vendors and programming languages have helpfully documented their memory models, so we just need to read their specifications…

C++	good intentions and well written, but flawed (thin-air, unsound wrt h/w)
x86	TSO, except where it isn't (IRIW)
Armv7/PPC	Mind-bending recursion attempts to place accesses into 'groups'
JMM	defined empirically in terms of a cryptic set of tests
Perl6(!)	can't tell if it's a joke. I hope that it is.

Ok, so what do I need to know?

Architects, CPU vendors and programming languages have helpfully documented their memory models, so we just need to read their specifications…

C++	good intentions and well written, but flawed (thin-air, unsound wrt h/w)
x86	TSO, except where it isn't (IRIW)
Armv7/PPC	Mind-bending recursion attempts to place accesses into 'groups'
JMM	defined empirically in terms of a cryptic set of tests
Perl6(!)	can't tell if it's a joke. I hope that it is.

No formal wording. No common nomenclature. No common abstraction. No official tooling. No accountability.

arm

Ok, so what do I need to know?

Architects, CPU vendors and programming languages have helpfully documented their memory models, so we just need to read their specifications…

C++	good intentions and well written, but flawed (thin-air, unsound wrt h/w)
x86	TSO, except where it isn't (IRIW)
Armv7/PPC	Mind-bending recursion attempts to place accesses into 'groups'
JMM	defined empirically in terms of a cryptic set of tests
Perl6(!)	can't tell if it's a joke. I hope that it is.

No formal wording. No common nomenclature. No common abstraction. No official tooling. No accountability.

It mostly works by magic…

Ok, so what do I need to know?

Architects, CPU vendors and programming languages have helpfully documented their memory models, so we just need to read their specifications…

C++	good intentions and well written, but flawed (thin-air, unsound wrt h/w)
x86	TSO, except where it isn't (IRIW)
Armv7/PPC	Mind-bending recursion attempts to place accesses into 'groups'
JMM	defined empirically in terms of a cryptic set of tests
Perl6(!)	can't tell if it's a joke. I hope that it is.

No formal wording. No common nomenclature. No common abstraction. No official tooling. No accountability.

It mostly works by magic…

…Engineering shouldn't be magic.

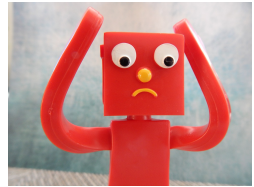
arm

The trouble with prose

Unsurprisingly, people are confused by memory models.

- “Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations.”
- “…This means that ACQUIRE acts as a minimal ‘acquire’ operation and RELEASE acts as a minimal ‘release’ operation.”
- “A DMB creates two groups of memory accesses, Group A and Group B:…”

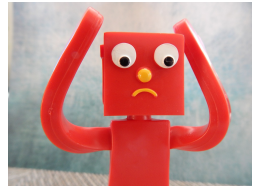
A cacophony of confusion



- “AFAIK, on x86 cpu fence is no-op. My understanding that on ARM I have to use ISB?”
- “Currently this is implemented using a full barrier. Is it still OK to use acquire/release ordering in this case?”
- “IOW, is ‘full barrier’ a more strong version of ‘fully ordered’ or not?”
- “`my head is here: o and memory barriers are over there mb`”
- “I, for one, understand nothing about memory barriers…

arm

A cacophony of confusion



- “AFAIK, on x86 cpu fence is no-op. My understanding that on ARM I have to use ISB?”
- “Currently this is implemented using a full barrier. Is it still OK to use acquire/release ordering in this case?”
- “IOW, is ‘full barrier’ a more strong version of ‘fully ordered’ or not?”
- “`my head is here: o and memory barriers are over there mb`”
- “I, for one, understand nothing about memory barriers…so I use them religiously”

Example: store buffering

Initially, `X` and `Y` are 0 in memory; `foo` and `bar` are local (register) variables:

p0

```
a: X = 1;
b: foo = Y;
```

p1

```
c: Y = 1;
d: bar = 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:

p0

```
a: X = 1;
b: foo = Y;
```

p1

```
c: Y = 1;
d: bar = X;
```

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

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

Lies, damned lies and sequential consistency

p0	p1	Interleavings
`a: X = 1;`	`c: Y = 1;`	`{a,b,c,d}`
`b: foo = Y;`	`d: bar = 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 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

p0	p1	Interleavings
`a: X = 1;`	`c: Y = 1;`	`{a,b,c,d}`
`b: foo = Y;`	`d: bar = 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 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.

Real hardware and architectures

	RR	RW	WW	WR	RA	WA	DR	IC
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
ARMv7-A/R	Y	Y	Y	Y	Y	Y		Y
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

Variety of CPU behaviours within the scope of a single architecture.

arm

Table courtesy of Paul McKenney ('perfbook')

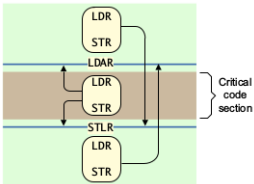
Overview of Armv8

Armv8 is weakly-ordered and requires special instructions to restore SC. Explicitly designed with C/C++11 (SC-DRF) in mind.

Dependencies from a Load to a subsequent instruction. Control, Data and Address.

Barrier instructions with access-type restrictions (`DMB`, `DSB`)

Acquire/release instructions (`LDAR`, `STLR`) are RCsc with roach motel semantics



Revised to be multi-copy atomic…

arm

Litmus tests

Memory ordering problems can be expressed as litmus tests with funny names:

```
AArch64 S
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
P0           | P1           ;
MOV W0,#2    | LDR W0,[X1] ;
STR W0,[X1] | MOV W2,#1    ;
MOV W2,#1    | STR W2,[X3] ;
STR W2,[X3] |              ;
exists
(x=2 /\ 1:X0=1)
```

These tests are constructed from a program and a constraint.

Litmus tests

Memory ordering problems can be expressed as litmus tests with funny names:

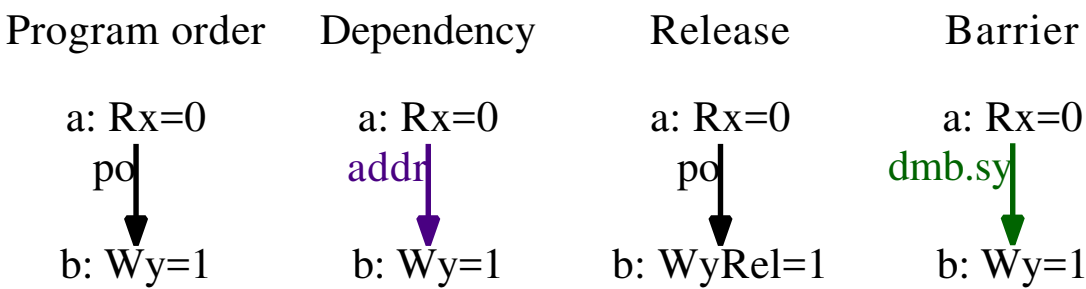
```
AArch64 S
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
P0           | P1           ;
MOV W0,#2    | LDR W0,[X1] ;
STR W0,[X1] | MOV W2,#1    ;
MOV W2,#1    | STR W2,[X3] ;
STR W2,[X3] |              ;
exists
(x=2 /\ 1:X0=1)
```

These tests are constructed from a program and a constraint.

Search Google for ‘test6.pdf’

Programs

A program describes the memory-related (read, write and barrier) instructions in each thread, and any dependencies they may have:



Notice that the program relations are strictly intra-thread.

arm

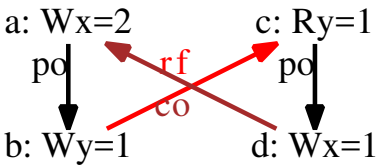
Candidate executions

For a given litmus test, a candidate execution binds the values of reads and the final values in memory by providing two relations:

Reads-from (rf) pairs each read with a write to the same location

Coherence-order (co) pairs stores to the same location in a total order (the overwrite order)

```
  P0             |   P1           ;
  MOV W0,#2      | c:LDR W0,[X1] ;
a:STR W0,[X1]    |   MOV W2,#1    ;
  MOV W2,#1      | d:STR W2,[X3] ;
b:STR W2,[X3]    |                ;
```



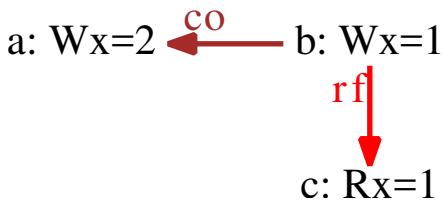
$rf = \{(b, c)\}$

$co = \{x \mapsto \{(x_0, d), (d, a), (x_0, a)\}, y \mapsto \{(y_0, b)\}\}$

We can generate all possible candidate executions and feed them to a particular memory model.

arm

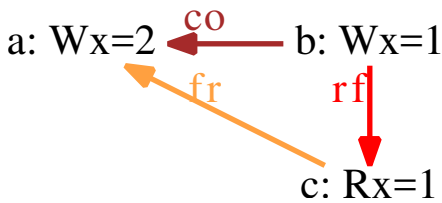
Deriving from-reads (fr)



The from-reads relation links a read to all writes appearing later in the coherence order than the write from which the read reads-from.

co, rf and fr encapsulate the three forms of inter-thread communication.

Deriving from-reads (fr)



The from-reads relation links a read to all writes appearing later in the coherence order than the write from which the read reads-from.

co, rf and fr encapsulate the three forms of inter-thread communication.

Formal modelling

Evaluating litmus tests by hand is cumbersome and error-prone…

Formal modelling

Evaluating litmus tests by hand is cumbersome and error-prone…

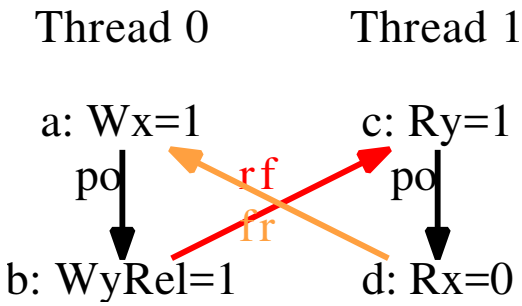
Formal models to the rescue!

- Exhaustively generate all possible outcomes for a test
- Generate more tests of interest
- Run tests on real hardware
- Generate tests that differ between memory models

Develop an intuition for the limitations of a memory model whilst ensuring correctness of code.

Example litmus test: MP+popl+po

```
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)
```



arm

Example litmus test: MP+popl+po

```
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)
```

```
Test MP+popl+po Allowed
States 4
1:X0=0; 1:X2=0;
1:X0=0; 1:X2=1;
1:X0=1; 1:X2=0;
1:X0=1; 1:X2=1;
Ok
Witnesses
Positive: 1 Negative: 3
Condition exists (1:X0=1 /\ 1:X2=0)
Observation MP+popl+po Sometimes 1 3
Time MP+popl+po 0.01
Hash=75d804cb38f3f607de6ab3cc9925140e
```


arm

Operational models

rmem

http://www.cl.cam.ac.uk/~sf502/regressions/rmem/

Operational models

An operational model is a complex transition system that incrementally explores possible states of an abstract machine.

- Close to an ‘abstract’, maximally permissive CPU design
- Split each instruction into multiple steps
- Allows the user to build an intuition based upon a trace of events
- Emergent behaviour, but relatively slow execution speed

Incrementally generates all possible outcomes for a given program.

Example: Armv8 `STR` instruction

Pseudocode for an instruction can be broken down into sub-transitions. For a store (roughly):

1. Fetch instruction
2. Read input registers
3. Initiate writes with memory footprint
4. Instantiate write values
5. Commit store
6. Propagate to all other threads
7. Complete instruction
8. Finish instruction

Operational semantics for `SHMEM_PUT_NBI`?

The routines return after posting the operation. The operation is considered complete after a subsequent call to `shmem_quiet`. At the completion of `shmem_quiet`, the data has been copied into the dest array on the destination PE. The delivery of data words into the data object on the destination PE may occur in any order. Furthermore, two successive put routines may deliver data out of order unless a call to `shmem_fence` is introduced between the two calls.

Axiomatic models

herdtools7

https://github.com/herd/herdtools7

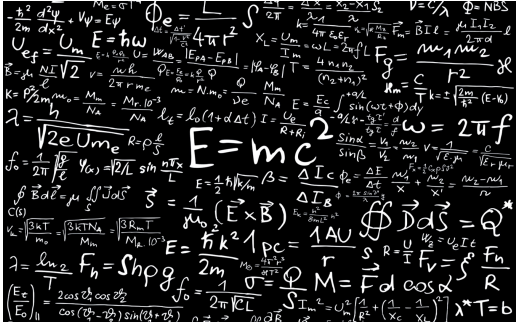
Axiomatic models

An axiomatic model places mathematical constraints on relations over events derived from the execution of a program.

For each candidate execution:

- Construct the *rf* and *co* relations
- Construct the derived *fr* relation
- Construct high-level derived relations
- Validate against acyclicity constraints

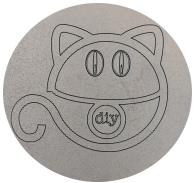
In other words, define relations that are strict partial orders over the set of events.



Generate all possible executions and test against a predicate.

arm

Operation of `herd`



For a given litmus test, `herd` helpfully constructs the event sets and basic relations for you:

- Reads (`R`), Writes (`W`), Fences …
- *rf*, *fr*, *co*, *po*, …
- A `.cat` file defines the derived relations and their constraints for a given memory model
- All candidate executions are generated by `herd` and evaluated in-turn by the `.cat` file
- A litmus test will be permitted always, sometimes or never

Candidate generation

```
AArch64
{
0:X1=x; 1:X1=x
}
 P0           | P1           ;
 MOV W0,#1    | LDR W0,[X1] ;
 STR W0,[X1] |              ;
```

arm

Syntax of the `cat` DSL

A bit like `ML`, but focussed on defining constrained relations:

Event sets and selectors `W`, `[W]`, `L`, `R`, `A`

Set theory Composition (`;`), closure (`+`), Kleene star (`*`), intersection (`&`), union (`|`), difference (`\`)

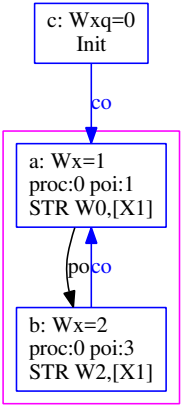
Definition `let myrelation = ...`

Built-in relations `int`, `ext`, `loc`

Constraints `acyclic`, `irreflexive`, `empty`

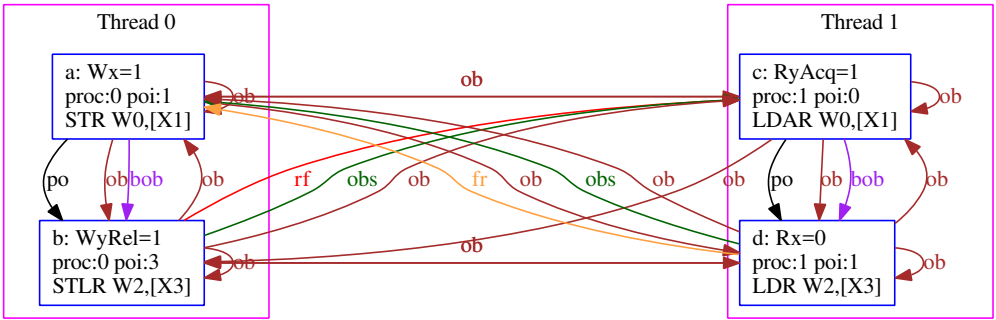
```
let po-loc = po & loc
let rfe = rf & ext
```

Examples for Arm (CoWW)




```
acyclic po-loc | fr | co | rf as coherence
```

Examples for Arm (MP+popl+poap)




```
let bob = ([A | Q]; po) | (po; [L])
let obs = rfe | fre | coe
let rec ob = obs | bob | ob; ob
irreflexive ob
```

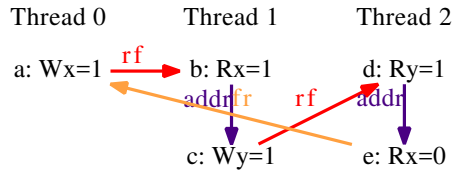

arm

The revised Armv8 memory model

Multi-copy atomicity

Multi-copy atomicity forbids a number of litmus tests, but is difficult to characterise:

- A store from a thread becomes visible to all other threads at the same time
- i.e. the cache coherency protocol hides write-propagation delays from software
- Reduces memory ordering to intra-thread re-ordering
- Compositional reasoning without overhead of SC
- The following is forbidden by MCA:



Test WRC+addrs, Generic(ARMv8 AArch64)

arm

Revised Armv8 memory model

Evolution to multi-copy atomicity means all ordering rules are created equal and largely restricted to intra-thread accesses:

- Architected in an ‘axiomatic’ style
- Each line of English corresponds directly to an axiomatic definition
- Requires acyclicity of the `ordered-before` relation
- Abstracts away implementation details, but what about incremental debugging?

Revised Armv8 memory model

Evolution to multi-copy atomicity means all ordering rules are created equal and largely restricted to intra-thread accesses:

- Architected in an ‘axiomatic’ style
- Each line of English corresponds directly to an axiomatic definition
- Requires acyclicity of the `ordered-before` relation
- Abstracts away implementation details, but what about incremental debugging?

Proof of equivalance between axiomatic and operational models:

`http://www.cl.cam.ac.uk/~pes20/armv8-mca/armv8-mca-draft.pdf`

arm

Example: `barrier-ordered-before`

A read or a write RW_1 is *Barrier-ordered-before* a read or a write RW_2 from the same Observer if and only if RW_1 appears in program order before RW_2 and any of the following cases apply:

…

- RW_1 is a write W_1 generated by an instruction with Release semantics and RW_2 is a read R_2 generated by an instruction with Acquire semantics.
- RW_1 is a read R_1 and either:
  - R_1 appears in program order before a `DMB LD` that appears in program order before RW_2
  - R_1 is generated by an instruction with Acquire or AcquirePC semantics

…

Example: barrier-ordered-before

A read or a write RW_1 is *Barrier-ordered-before* a read or a write RW_2 from the same Observer if and only if RW_1 appears in program order before RW_2 and any of the following cases apply:

…

- RW_1 is a write W_1 generated by an instruction with Release semantics and RW_2 is a read R_2 generated by an instruction with Acquire semantics.
- RW_1 is a read R_1 and either:
  - R_1 appears in program order before a `DMB LD` that appears in program order before RW_2
  - R_1 is generated by an instruction with Acquire or AcquirePC semantics

…

```
let bob = ...
  | [L]; po; [A]
  | [R]; po; [dmb.ld]; po
  | [A | Q]; po
  | ...
```

Litmus tests in OpenSHMEM

(my first attempt)

Event syntax

- Memory initialised to zero
- Local variables named `rN`
- All non-local (symmetric) variables accessed by implicit pointer dereference
- Simple C assignment: `x=42`, `r0=y`
- Basic tests around `put`, `get` etc
- `put(x=1,5) => char _x = 1; openshmem_put(&x, &_x, 1, 5)`
- `r0 = get(x,5) => openshmem_get(&r0, &x, 1, 5)`
- Warning! Based on quick reading of OpenSHMEM spec: tests may be nonsensical

Coherence

*…two successive **put** routines may deliver data out of order unless a call to `shmem_fence` is introduced between the two calls.*

We can adapt `CoWW` in a couple of ways:

```
 P0            | P1 ;
 put(x=1, 1)   |    ;
 put(x=2, 1)   |    ;
 r0=get(x, 1)  |    ;

exists 0:r0=1
```


arm

Coherence

*…two successive **put** routines may deliver data out of order unless a call to `shmem_fence` is introduced between the two calls.*

We can adapt `CoWW` in a couple of ways:

```
P0            | P1 ;
put(x=1, 1)   |    ;
put(x=2, 1)   |    ;
r0=get(x, 1)  |    ;

exists 0:r0=1
```

```
P0            | P1                ;
put(x=1, 1)   | wait_until(x=2)   ;
put(x=2, 1)   | r0=x              ;

exists 1:r0=1
```


arm

MP

```
 P0            | P1               ;
 put(x=1, 1)   | wait_until(y=1)  ;
 fence()       | r0=x             ;
 put(y=1, 1)   |                  ;

exists 1:r0=0
```

MP

```
P0            | P1               ;
put(x=1, 1)   | wait_until(y=1)  ;
fence()       | r0=x             ;
put(y=1, 1)   |                  ;

exists 1:r0=0
```

What about interaction with C11 (data races) and dependencies?

```
P0                  | P1                    ;
atomic_store(x, 1)  | r0=get(y, 0)          ;
atomic_store(y, 1)  | if (r0 == 1)          ;
                    |   r1=get_nbi(x, 0)   ;

exists 1:r0=1 /\ 1:r1=0
```

SB

Need `quiet()` for Store->Load ordering:

```
 P0              | P1             ;
 x=1             | y=1            ;
 quiet()         | quiet()        ;
 r0=get(y, 1)    | r0=get(x, 0)   ;

exists 0:r0=0 /\ 1:r0=0
```

ISA2

Three threads:

```
 P0              | P1               | P2               ;
 x=1             | wait_until(y=1)  | wait_until(z=1)  ;
 quiet()         | put(z=1, 2)      | r0=get(x, 0)     ;
 put(y=1, 1)     |                  |                  ;

exists 2:r0=0
```

Can we avoid the `quiet()` on P0?

IRIW

Four threads:

```
 P0   | P1             | P2             | P3  ;
 x=1  | r0=get(x, 0)   | r0=get(x, 3)   | x=1 ;
      | r1=get(x, 3)   | r1=get(x, 0)   |     ;

exists 1:r0=1 /\ 1:r1=0 /\ 2:r0=1 /\ 2:r1=0
```

`barrier()` required?

Questions?