Multicore Semantics: Making Sense of Relaxed Memory

Peter Sewell[1], Christopher Pulte[1], Shaked Flur[1,2]

with contributions from Mark Batty[3], Luc Maranget[4], Alasdair Armstrong[1]

[1] University of Cambridge, [2] Google, [3] University of Kent, [4] INRIA Paris

February – March, 2025

Slides for Part 2 of the Multicore Semantics and Programming course, version of 2026-02-12

Part 1 is by Tim Harris, with separate slides

These Slides

These are the slides for the Multicore Semantics part of the University of Cambridge *Multicore Semantics and Programming* course (MPhil ACS, Part III, Part II), 2024–2025.

They cover multicore semantics: the concurrency of multiprocessors and programming languages, focussing on the concurrency behaviour one can rely on from mainstream machines and languages, how this can be investigated, and how it can be specified precisely, all linked to usage, microarchitecture, experiment, and proof.

We focus largely on x86; on Arm-A, IBM POWER, and RISC-V; and on C/C++. We use the x86 part also to introduce some of the basic phenomena and the approaches to modelling and testing, and give operational and axiomatic models in detail. For Armv8-A, POWER, and RISC-V we introduce many but not all of the phenomena and again give operational and axiomatic models, but omitting some aspects. For C/C++11 we introduce the programming-language concurrency design space, including the thin-air problem, the C/C++11 constructs, and the basics of its axiomatic model, but omit full explanation of the model.

These lectures are by Peter Sewell, with Christopher Pulte for the Armv8/RISC-V model section. The slides are for around 10 hours of lectures, and include additional material for reference.

The other part of the course, by Tim Harris, covers concurrent programming: simple algorithms, correctness criteria, advanced synchronisation patterns, transactional memory.

These Slides

The slides include citations to some of the most directly relevant related work, but this is primarily a lecture course focussed on understanding the concurrency semantics of mainstream architectures and languages as we currently see them, for those that want to program above or otherwise use those models, not a comprehensive literature review. There is lots of other relevant research that we do not discuss.

Acknowledgements

Contributors to these slides: Shaked Flur, Christopher Pulte, Mark Batty, Luc Maranget, Alasdair Armstrong. Ori Lahav and Viktor Vafeiadis for discussion of the current models for C/C++. Paul Durbaba for his 2021 Part III dissertation mechanising the x86-TSO axiomatic/operational correspondence proof.

Our main industry collaborators: Derek Williams (IBM); Richard Grisenthwaite and Will Deacon (Arm); Hans Boehm, Paul McKenney, and other members of the C++ concurrency group; Daniel Lustig and other members of the RISC-V concurrency group

All the co-authors of the directly underlying research [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] [16, ?, 17, ?, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29], especially all the above, Susmit Sarkar, Jade Alglave, Scott Owens, Kathryn E. Gray, Jean Pichon-Pharabod, and Francesco Zappa Nardelli, and the authors of the language-level research cited later.

The students of this and previous versions of the course, from 2010–2011 to date.

Research funding: ERC Advanced Grant 789108 (ELVER, Sewell); EPSRC grants EP/K008528/1 (Programme Grant REMS: Rigorous Engineering for Mainstream Systems), EP/F036345 (Reasoning with Relaxed Memory Models), EP/H005633 (Leadership Fellowship, Sewell), and EP/H027351 (Postdoc Research Fellowship, Sarkar); the Scottish Funding Council (SICSA Early Career Industry Fellowship, Sarkar); an ARM iCASE award (Pulte); ANR grant WMC (ANR-11-JS02-011, Zappa Nardelli, Maranget); EPSRC IAA KTF funding; Arm donation funding; IBM donation funding; ANR project ParSec (ANR-06-SETIN-010); and INRIA associated team MM. This work is part of the CIFV project sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-18-C-7809. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Memory

The abstraction of a *memory* goes back some time...

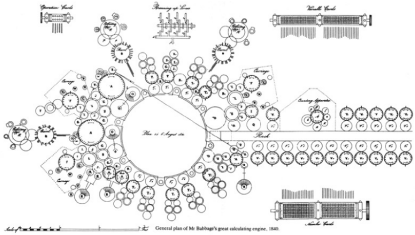
Memory

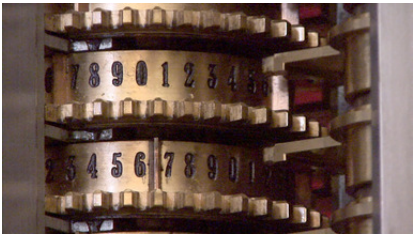
The calculating part of the engine may be divided into two portions

1st *The Mill in which all operations are performed*

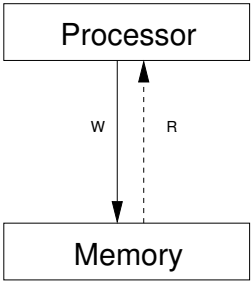
2nd *The Store in which all the numbers are originally placed and to which the numbers computed by the engine are returned.*

[Dec 1837, *On the Mathematical Powers of the Calculating Engine*, Charles Babbage]





The Golden Age, (1837–) 1945–1962



1962: First(?) Multiprocessor

BURROUGHS D825, 1962

“Outstanding features include truly modular hardware with parallel processing throughout”
FUTURE PLANS The complement of compiling languages is to be expanded.”

Multiprocessors, 1962–now

Niche multiprocessors since 1962

IBM System 370/158MP in 1972

Mass-market since 2005 (Intel Core 2 Duo).

Multiprocessors, 2019

Intel Xeon E7-8895 v3
36 hardware threads

ARM®

Commonly 8 hardware threads.

IBM®

IBM Power 8 server
(up to 1536 hardware threads)

Why now?

Exponential increases in transistor counts continued — but not per-core performance

- energy efficiency (computation per Watt)
- limits of instruction-level parallelism

Concurrency finally mainstream — but how to understand, design, and program concurrent systems? Still very hard.

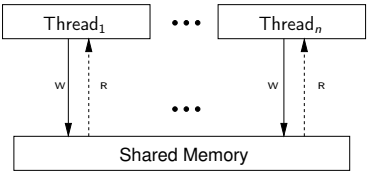
Concurrency everywhere

At many scales:

- intra-core
- multicore processors ← our focus
- ...and programming languages ← our focus
- GPU
- datacenter-scale
- internet-scale

explicit message-passing vs shared memory abstractions

The most obvious semantics: Sequential Consistency



Multiple threads acting on a *sequentially consistent* (SC) shared memory:

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program *[Lamport, 1979]*

A naive two-thread mutual-exclusion algorithm

Initial state: `x=0; y=0;`	
Thread 0	Thread 1
`x=1;` `if (y==0) {...critical section...}`	`y=1;` `if (x==0) {...critical section...}`

Can both be in their critical sections at the same time, in SC?

A naive two-thread mutual-exclusion algorithm

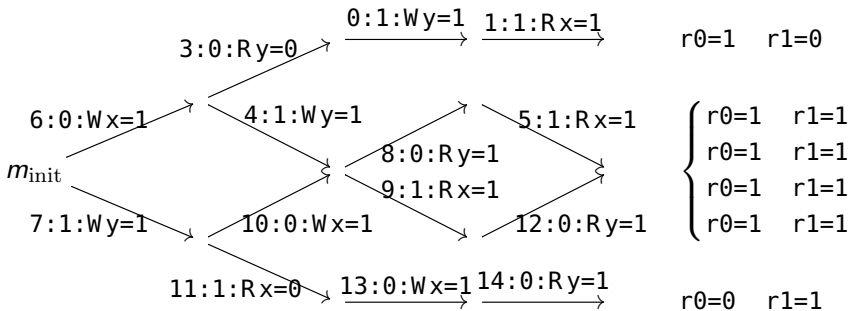
Initial state: `x=0; y=0;`	
Thread 0	Thread 1
`x=1;` `r0=y`	`y=1;` `r1=x`

Is a final state with `r0=0` and `r1=0` possible in SC?

A naive two-thread mutual-exclusion algorithm

Initial state: `x=0; y=0;`	
Thread 0	Thread 1
`x=1;` `r0=y`	`y=1;` `r1=x`

Is a final state with `r0=0` and `r1=0` possible in SC?
Try all six interleavings of SC model:



Let's try...

We'll use the `litmus7` tool (`diy.inria.fr`, Alglave, Maranget, et al. [27])

Write the test in litmus format, in a file `SB.litmus`:

```
1  X86_64 SB
2  "PodWR Fre PodWR Fre"
3  Cycle=Fre PodWR Fre PodWR
4  Relax=
5  Safe=Fre PodWR
6  Generator=diy7 (version 7.55+01(dev))
7  Prefetch=0:x=F,0:y=T,1:y=F,1:x=T
8  Com=Fr Fr
9  Orig=PodWR Fre PodWR Fre
10 Align=
11 {
12 uint64_t y; uint64_t x; uint64_t 1:rax; uint64_t 0:rax;
13
14 }
15  P0             | P1             ;
16  movq $1,(y)    | movq $1,(x)    ;
```

Let's try...

To install `litmus7`:

1. install the `opam` package manager for OCaml: `https://opam.ocaml.org/`
2. `opam install herdtools7` (docs at `diy.inria.fr`)

Let's try...

```
[...]
Generated assembler
#START _litmus_P1
        movq $1,(%r9,%rcx)
        movq (%r8,%rcx),%rax
#START _litmus_P0
        movq $1,(%r8,%rcx)
        movq (%r9,%rcx),%rax
[...]
```

Let's try...

```
$ litmus7 SB.litmus
[...]
Histogram (4 states)
14    *>0:rax=0; 1:rax=0;
499983:>0:rax=1; 1:rax=0;
499949:>0:rax=0; 1:rax=1;
54    :>0:rax=1; 1:rax=1;
[...]
Observation SB Sometimes 14 999986
[...]
```

14 in 1e6, on an Intel Core i7-7500U

(beware: 1e6 is a small number; rare behaviours might need 1e9+, and litmus tuning)

Let's try...

```
Histogram (4 states)
7136481  *> 0:X2=0; 1:X2=0;
596513783:> 0:X2=0; 1:X2=1;
596513170:> 0:X2=1; 1:X2=0;
36566    :> 0:X2=1; 1:X2=1;
[...]
Observation SB Sometimes 7136481 1193063519
```

7e6 in 1.2e9, on an Apple-designed ARMv8-A SoC (Apple A10 Fusion) in an iPhone 7

Let's try...

Why could that be?

1. error in the test
2. error in the `litmus7`-generated test harness
3. error in the OS
4. error in the hardware processor design
5. manufacturing defect in the particular silicon we're running on
6. error in our calculation of what the SC model allows
7. error in the model

Let's try...

Why could that be?

1. error in the test
2. error in the `litmus7`-generated test harness
3. error in the OS
4. error in the hardware processor design
5. manufacturing defect in the particular silicon we're running on
6. error in our calculation of what the SC model allows
7. error in the model ← this time

Sequential Consistency is not a correct model for x86 or Arm processors.

Let's try...

Why could that be?

1. error in the test
2. error in the `litmus7`-generated test harness
3. error in the OS
4. error in the hardware processor design
5. manufacturing defect in the particular silicon we're running on
6. error in our calculation of what the SC model allows
7. error in the model ← this time

Sequential Consistency is not a correct model for x86 or Arm processors.

...or for IBM Power, RISC-V, C, C++, Java, etc.

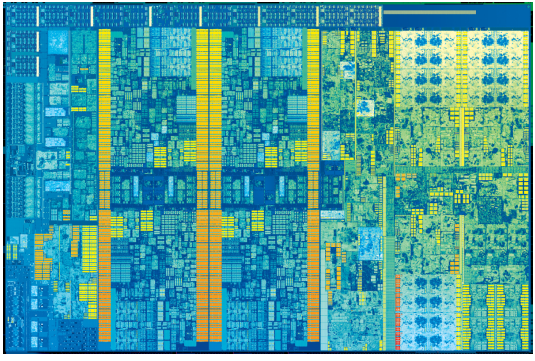
Instead, all these have some form of ***relaxed memory model*** (or *weak memory model*), allowing some non-SC behaviour

What does it mean to be a good model?

Processor implementations

Intel i7-8700K, AMD Ryzen 7 1800X, Qualcomm Snapdragon 865, Samsung Exynos 990, IBM Power 9 Nimbus, ...

Each has fantastically complex internal structure:



[Die shot of quad-core Intel i7-7700K (Kaby Lake) processor, en.wikichip.org]

Processor implementations

We can't use that as our *programmer's model* – it's:

- too complex
- too confidential
- too *specific*:

  software should run correctly on a wide range of hardware implementations, current and future

Architecture specifications

An *architecture specification* aims to define an envelope of the *programmer-observable behaviour* of all members of a processor family:

the set of all behaviour that a programmer might see by executing multithreaded programs on any implementation of that family.

The hardware/software interface, serving both as the

1. criterion for correctness of hardware implementations, and the
2. specification of what programmers can depend on.

Architecture specifications

Thick books:

- Intel 64 and IA-32 Architectures Software Developer's Manual [30], 5052 pages
- AMD64 Architecture Programmer's Manual [31], 3165 pages
- Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile [32], 8248 pages
- Power ISA Version 3.0B [33], 1258 pages
- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [34] and Volume II: Privileged Architecture [35], 238+135 pages

Architecture specifications

Thick books:

- Intel 64 and IA-32 Architectures Software Developer's Manual [30], 5052 pages
- AMD64 Architecture Programmer's Manual [31], 3165 pages
- Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile [32], 8248 pages
- Power ISA Version 3.0B [33], 1258 pages
- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [34] and Volume II: Privileged Architecture [35], 238+135 pages

Each aims to define the:

- *architected state* (programmer-visible registers etc.)
- *instruction-set architecture* (ISA): instruction encodings and sequential behaviour
- *concurrency architecture* – how those interact
- ...

Architecture specifications

Architectures have to be *loose* specifications:

- accommodating the range of behaviour from runtime nondeterminism of a single implementation (e.g. from timing variations, cache pressure, ...)
- ...and from multiple implementations, with different microarchitecture

Desirable properties of an architecture specification

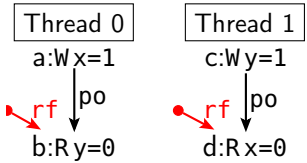
1. Sound with respect to current hardware
2. Sound with respect to future hardware
3. Opaque with respect to hardware microarchitecture implementation detail
4. Complete with respect to hardware?
5. Strong enough for software
6. Unambiguous / precise
7. Executable as a test oracle
8. Incrementally executable
9. Clear
10. Authoritative?

Litmus tests and candidate executions

SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	Thread 1
movq $1, (x) //a movq (y), %rax //b	movq $1, (y) //c movq (x), %rax //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000



Candidate executions consist of:

- a choice of a control-flow unfolding of the test source
- a choice, for each memory read, of which write it reads from, or the initial state
- ...more later

Represented as graphs, with nodes the memory events and various relations, including:

- *program order* po
- *reads-from* rf

The final-state condition of the test often identifies a unique candidate execution
...which might be observable or not on h/w, and allowed or not by a model.

Why is this an academic subject?

Why not just read the manuals?

Those desirable properties turn out to be very hard to achieve, esp. for subtle real-world concurrency

In 2007, many architecture prose texts were too vague to interpret reliably

Research from then to date has clarified much, and several architectures now incorporate precise models based on it (historical survey later)

...and this enables many kinds of research above these models

Much still to do!

x86

x86 basic phenomena

Observable relaxed-memory behaviour arises from hardware optimisations

(and compiler optimisations for language-level relaxed behaviour)

Observable relaxed-memory behaviour arises from hardware optimisations

(and compiler optimisations for language-level relaxed behaviour)

so we should be able to understand and explain them in those terms

Scope: “user” concurrency

Focus for now on the behaviour of memory accesses and barriers, as used in most concurrent algorithms (in user or system modes, but without systems features).

Coherent write-back memory, assuming:

- no misaligned or mixed-size accesses
- no exceptions
- no self-modifying code
- no page-table changes
- no ‘non-temporal’ operations
- no device memory

Most of those are active research areas. We also ignore fairness properties, considering finite executions only

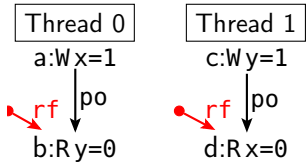
SB x86

Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//c`
`movq (y), %rax`	`//b`	`movq (x), %rax`	`//d`

Final: `0:rax=0; 1:rax=0;`

Observation: 171/100000000

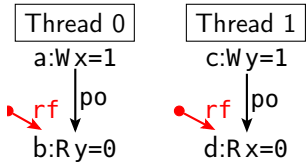


SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	Thread 1
movq $1, (x) //a movq (y), %rax //b	movq $1, (y) //c movq (x), %rax //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000

- experimentally: observed

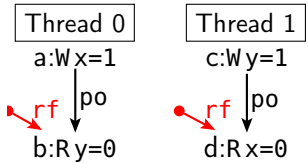


SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	Thread 1
movq $1, (x) //a movq (y), %rax //b	movq $1, (y) //c movq (x), %rax //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000

- experimentally: observed
- possible microarchitectural explanation?



SB x86

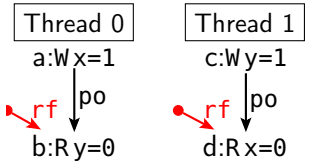
Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

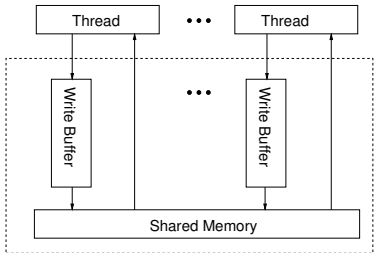
Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//c`
`movq (y), %rax`	`//b`	`movq (x), %rax`	`//d`

Final: `0:rax=0; 1:rax=0;`

`Observation: 171/100000000`

- experimentally: observed
- possible microarchitectural explanation?
  buffer stores? out-of-order execution?





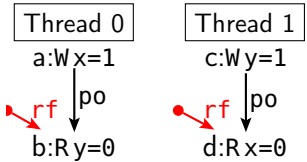
SB x86

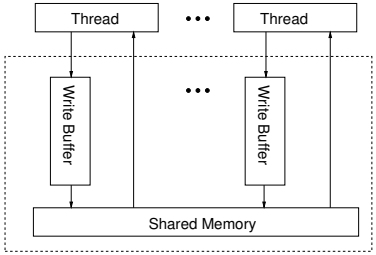
Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//c`
`movq (y), %rax`	`//b`	`movq (x), %rax`	`//d`

Final: `0:rax=0; 1:rax=0;`

`Observation: 171/100000000`





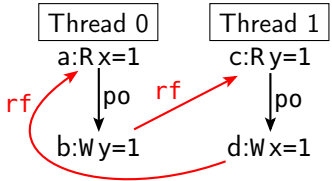
- experimentally: observed
- possible microarchitectural explanation?
  buffer stores? out-of-order execution?
- architecture prose and intent?

  Reads may be reordered with older writes to different locations but not with older writes to the same location. [Intel SDM,§8.2.2, and Example 8-3]

LB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	Thread 1
`movq (x), %rax //a` `movq $1, (y) //b`	`movq (y), %rax //c` `movq $1, (x) //d`
Final: 0:rax=1; 1:rax=1;	

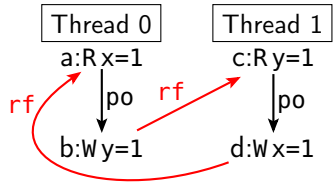
Observation: 0/0



LB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	Thread 1
`movq (x), %rax //a` `movq $1, (y) //b`	`movq (y), %rax //c` `movq $1, (x) //d`
Final: 0:rax=1; 1:rax=1;	

`Observation: 0/0`

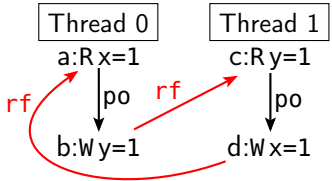


- experimentally: not observed

LB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	Thread 1
`movq (x), %rax //a` `movq $1, (y) //b`	`movq (y), %rax //c` `movq $1, (x) //d`
Final: 0:rax=1; 1:rax=1;	

`Observation: 0/0`



- experimentally: not observed
- possible microarchitectural explanation?

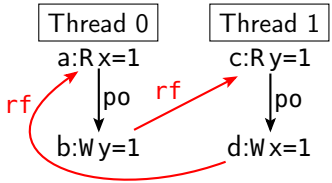
LB x86

Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

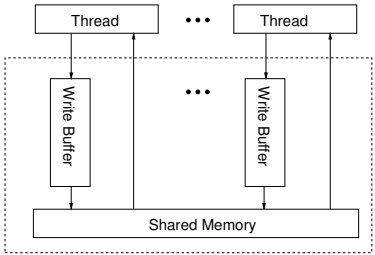
Thread 0	Thread 1
`movq (x), %rax //a`	`movq (y), %rax //c`
`movq $1, (y) //b`	`movq $1, (x) //d`

Final: `0:rax=1; 1:rax=1;`

`Observation: 0/0`



- experimentally: not observed
- possible microarchitectural explanation?
  Buffer load requests?
  Out-of-order execution?



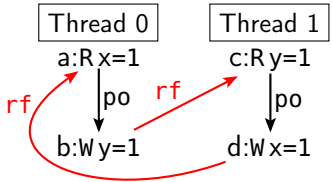
LB x86

Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0	Thread 1
`movq (x), %rax //a`	`movq (y), %rax //c`
`movq $1, (y) //b`	`movq $1, (x) //d`

Final: `0:rax=1; 1:rax=1;`

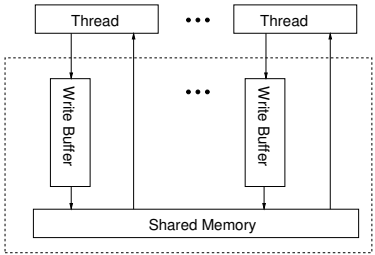
`Observation: 0/0`



- experimentally: not observed
- possible microarchitectural explanation?
  Buffer load requests?
  Out-of-order execution?
- architecture prose and intent?

  Reads may be reordered with older writes to different locations but not with older writes to the same location. [Intel SDM,§8.2.2]

  So?



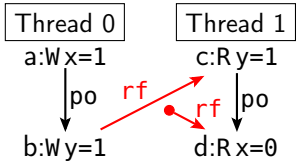
MP x86

Initial state: `1:rax=0; 1:rbx=0; y=0; x=0;`

Thread 0	Thread 1
`movq $1, (x) //a`	`movq (y), %rax //c`
`movq $1, (y) //b`	`movq (x), %rbx //d`

Final: `1:rax=1; 1:rbx=0;`

`Observation: 0/100000000`



MP x86

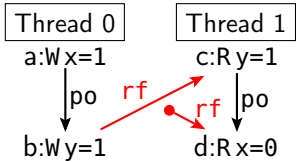
Initial state: `1:rax=0; 1:rbx=0; y=0; x=0;`

Thread 0	Thread 1
`movq $1, (x) //a`	`movq (y), %rax //c`
`movq $1, (y) //b`	`movq (x), %rbx //d`

Final: `1:rax=1; 1:rbx=0;`

`Observation: 0/100000000`

- experimentally: not observed
  (but it is on Armv8-A and IBM Power)



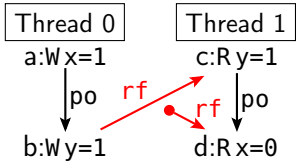
MP x86

Initial state: `1:rax=0; 1:rbx=0; y=0; x=0;`

Thread 0	Thread 1
`movq $1, (x) //a`	`movq (y), %rax //c`
`movq $1, (y) //b`	`movq (x), %rbx //d`

Final: `1:rax=1; 1:rbx=0;`

`Observation: 0/100000000`



- experimentally: not observed
  (but it is on Armv8-A and IBM Power)
- possible microarchitectural explanation?

  Out-of-order pipeline execution is another important hardware optimisation – but not *programmer-visible* here

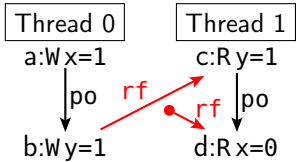
MP x86

Initial state: `1:rax=0; 1:rbx=0; y=0; x=0;`

Thread 0	Thread 1
`movq $1, (x) //a`	`movq (y), %rax //c`
`movq $1, (y) //b`	`movq (x), %rbx //d`

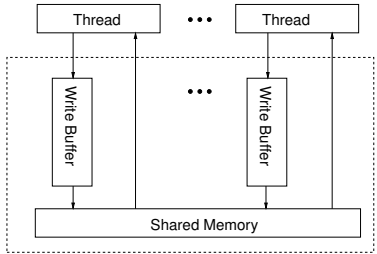
Final: `1:rax=1; 1:rbx=0;`

`Observation: 0/100000000`



- experimentally: not observed
  (but it is on Armv8-A and IBM Power)
- possible microarchitectural explanation?

  Out-of-order pipeline execution is another important hardware optimisation – but not *programmer-visible* here
- consistent with model sketch?



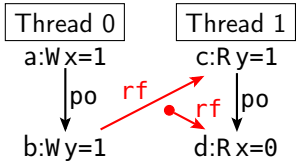
MP x86

Initial state: `1:rax=0; 1:rbx=0; y=0; x=0;`

Thread 0	Thread 1
`movq $1, (x) //a`	`movq (y), %rax //c`
`movq $1, (y) //b`	`movq (x), %rbx //d`

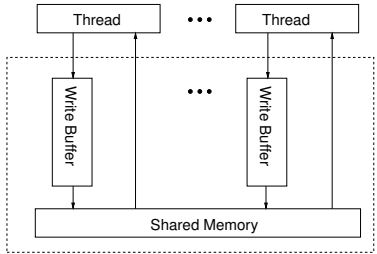
Final: `1:rax=1; 1:rbx=0;`

`Observation: 0/100000000`



- experimentally: not observed
  (but it is on Armv8-A and IBM Power)
- possible microarchitectural explanation?

  Out-of-order pipeline execution is another important hardware optimisation – but not *programmer-visible* here
- consistent with model sketch?
- architecture prose and intent?



> *Reads are not reordered with other reads. Writes to memory are not reordered with other writes, except non-temporal moves and string operations. Example 8-1*

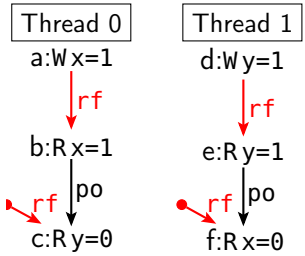
SB+rfi-pos x86

Initial state: `0:rax=0; 0:rbx=0; 1:rax=0; 1:rbx=0; y=0; x=0;`

Thread 0	Thread 1
`movq $1, (x) //a`	`movq $1, (y) //d`
`movq (x), %rax //b`	`movq (y), %rax //e`
`movq (y), %rbx //c`	`movq (x), %rbx //f`

Final: `0:rax=1; 0:rbx=0; 1:rax=1; 1:rbx=0;`

Observation: 320/100000000



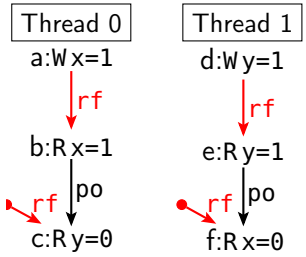
SB+rfi-pos x86

Initial state: `0:rax=0; 0:rbx=0; 1:rax=0; 1:rbx=0; y=0; x=0;`

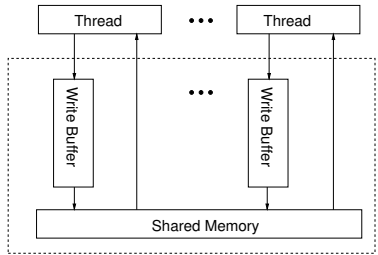
Thread 0	Thread 1
`movq $1, (x) //a`	`movq $1, (y) //d`
`movq (x), %rax //b`	`movq (y), %rax //e`
`movq (y), %rbx //c`	`movq (x), %rbx //f`

Final: `0:rax=1; 0:rbx=0; 1:rax=1; 1:rbx=0;`

`Observation: 320/100000000`



- is that allowed in the previous model sketch?



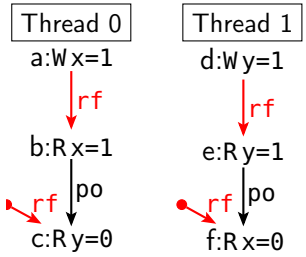
SB+rfi-pos x86

Initial state: `0:rax=0; 0:rbx=0; 1:rax=0; 1:rbx=0; y=0; x=0;`

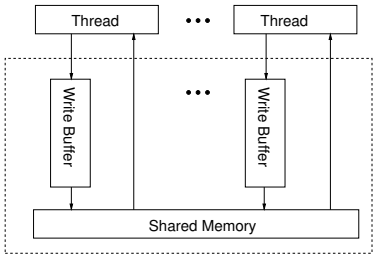
Thread 0	Thread 1
`movq $1, (x) //a`	`movq $1, (y) //d`
`movq (x), %rax //b`	`movq (y), %rax //e`
`movq (y), %rbx //c`	`movq (x), %rbx //f`

Final: `0:rax=1; 0:rbx=0; 1:rax=1; 1:rbx=0;`

`Observation: 320/100000000`



- is that allowed in the previous model sketch?
- we think the pairs of reads are not reordered – so no



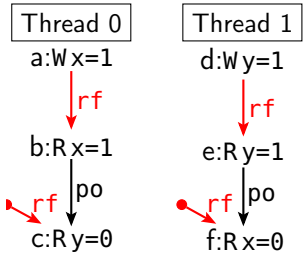
SB+rfi-pos x86

Initial state: `0:rax=0; 0:rbx=0; 1:rax=0; 1:rbx=0; y=0; x=0;`

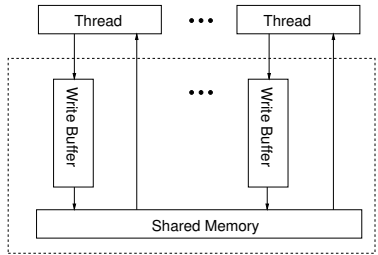
Thread 0	Thread 1
`movq $1, (x) //a`	`movq $1, (y) //d`
`movq (x), %rax //b`	`movq (y), %rax //e`
`movq (y), %rbx //c`	`movq (x), %rbx //f`

Final: `0:rax=1; 0:rbx=0; 1:rax=1; 1:rbx=0;`

`Observation: 320/100000000`



- is that allowed in the previous model sketch?
- we think the pairs of reads are not reordered – so no
- experimentally: observed



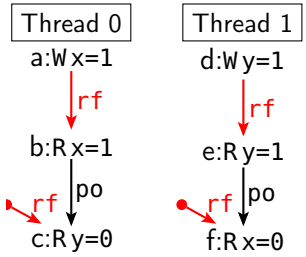
SB+rfi-pos x86

Initial state: `0:rax=0; 0:rbx=0; 1:rax=0; 1:rbx=0; y=0; x=0;`

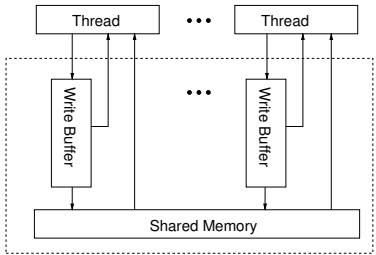
Thread 0	Thread 1
`movq $1, (x) //a`	`movq $1, (y) //d`
`movq (x), %rax //b`	`movq (y), %rax //e`
`movq (y), %rbx //c`	`movq (x), %rbx //f`

Final: `0:rax=1; 0:rbx=0; 1:rax=1; 1:rbx=0;`

`Observation: 320/100000000`



- is that allowed in the previous model sketch?
- we think the pairs of reads are not reordered – so no
- experimentally: observed
- microarchitectural refinement: allow – actually, *require* – reading from the store buffer



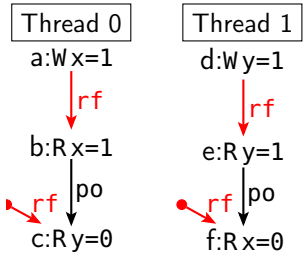
SB+rfi-pos x86

Initial state: `0:rax=0; 0:rbx=0; 1:rax=0; 1:rbx=0; y=0; x=0;`

Thread 0	Thread 1
`movq $1, (x) //a`	`movq $1, (y) //d`
`movq (x), %rax //b`	`movq (y), %rax //e`
`movq (y), %rbx //c`	`movq (x), %rbx //f`

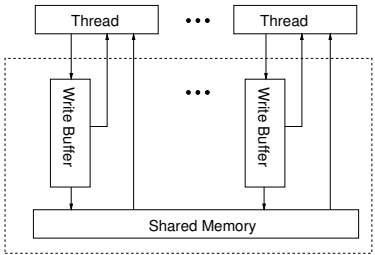
Final: `0:rax=1; 0:rbx=0; 1:rax=1; 1:rbx=0;`

`Observation: 320/100000000`



- is that allowed in the previous model sketch?
- we think the pairs of reads are not reordered – so no
- experimentally: observed
- microarchitectural refinement: allow – actually, *require* – reading from the store buffer
- architecture prose and intent?

Principles? But Example 8-5



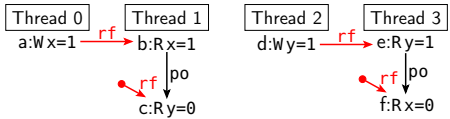
IRIW x86

Initial state: `1:rax=0; 1:rbx=0; 3:rax=0; 3:rbx=0; y=0; x=0;`

Thread 0	Thread 1	Thread 2	Thread 3
`movq $1, (x) //a`	`movq (x), %rax //b` `movq (y), %rbx //c`	`movq $1, (y) //d`	`movq (y), %rax //e` `movq (x), %rbx //f`

Final: `1:rax=1; 1:rbx=0; 3:rax=1; 3:rbx=0;`

`Observation: 0/100000000`



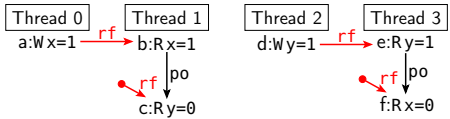
IRIW x86

Initial state: `1:rax=0; 1:rbx=0; 3:rax=0; 3:rbx=0; y=0; x=0;`

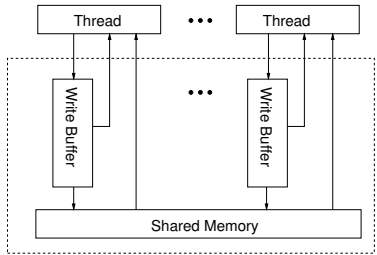
Thread 0	Thread 1	Thread 2	Thread 3
`movq $1, (x) //a`	`movq (x), %rax //b` `movq (y), %rbx //c`	`movq $1, (y) //d`	`movq (y), %rax //e` `movq (x), %rbx //f`

Final: `1:rax=1; 1:rbx=0; 3:rax=1; 3:rbx=0;`

`Observation: 0/100000000`



- is that allowed in the previous model sketch?



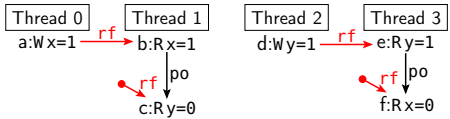
IRIW x86

Initial state: `1:rax=0; 1:rbx=0; 3:rax=0; 3:rbx=0; y=0; x=0;`

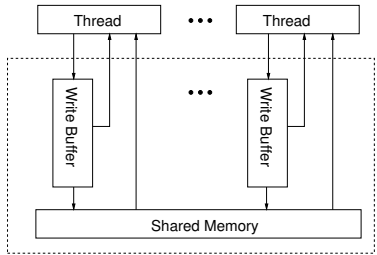
Thread 0	Thread 1	Thread 2	Thread 3
`movq $1, (x) //a`	`movq (x), %rax //b` `movq (y), %rbx //c`	`movq $1, (y) //d`	`movq (y), %rax //e` `movq (x), %rbx //f`

Final: `1:rax=1; 1:rbx=0; 3:rax=1; 3:rbx=0;`

`Observation: 0/100000000`



- is that allowed in the previous model sketch?
- we think the T2,3 read pairs are not reorderable – so no



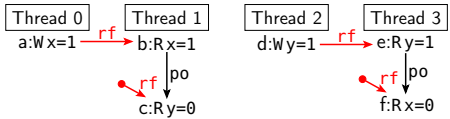
IRIW x86

Initial state: `1:rax=0; 1:rbx=0; 3:rax=0; 3:rbx=0; y=0; x=0;`

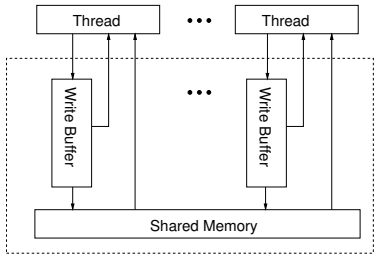
Thread 0	Thread 1	Thread 2	Thread 3
`movq $1, (x) //a`	`movq (x), %rax //b` `movq (y), %rbx //c`	`movq $1, (y) //d`	`movq (y), %rax //e` `movq (x), %rbx //f`

Final: `1:rax=1; 1:rbx=0; 3:rax=1; 3:rbx=0;`

`Observation: 0/100000000`



- is that allowed in the previous model sketch?
- we think the T2,3 read pairs are not reorderable – so no
- is it microarchitecturally plausible?



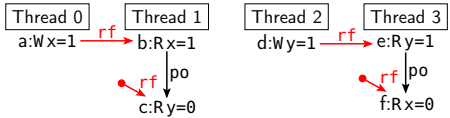
IRIW x86

Initial state: `1:rax=0; 1:rbx=0; 3:rax=0; 3:rbx=0; y=0; x=0;`

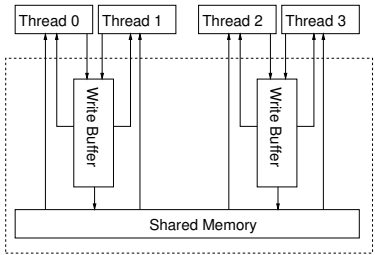
Thread 0	Thread 1	Thread 2	Thread 3
`movq $1, (x) //a`	`movq (x), %rax //b` `movq (y), %rbx //c`	`movq $1, (y) //d`	`movq (y), %rax //e` `movq (x), %rbx //f`

Final: `1:rax=1; 1:rbx=0; 3:rax=1; 3:rbx=0;`

`Observation: 0/100000000`



- is that allowed in the previous model sketch?
- we think the T2,3 read pairs are not reorderable – so no
- is it microarchitecturally plausible? yes, e.g. with shared store buffers or fancy cache protocols



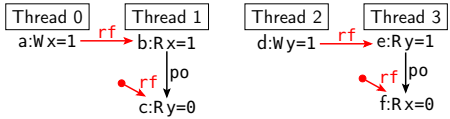
IRIW x86

Initial state: 1:rax=0; 1:rbx=0; 3:rax=0; 3:rbx=0; y=0; x=0;

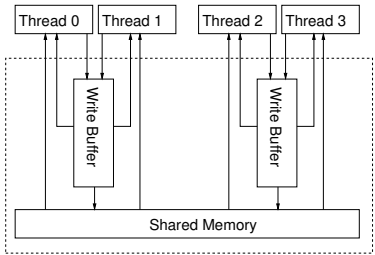
Thread 0	Thread 1	Thread 2	Thread 3
movq $1, (x) //a	movq (x), %rax //b movq (y), %rbx //c	movq $1, (y) //d	movq (y), %rax //e movq (x), %rbx //f

Final: 1:rax=1; 1:rbx=0; 3:rax=1; 3:rbx=0;

Observation: 0/100000000



- is that allowed in the previous model sketch?
- we think the T2,3 read pairs are not reorderable – so no
- is it microarchitecturally plausible? yes, e.g. with shared store buffers or fancy cache protocols
- experimentally: not observed



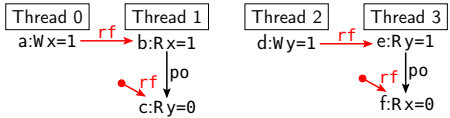
IRIW x86

Initial state: 1:rax=0; 1:rbx=0; 3:rax=0; 3:rbx=0; y=0; x=0;

Thread 0	Thread 1	Thread 2	Thread 3
movq $1, (x) //a	movq (x), %rax //b movq (y), %rbx //c	movq $1, (y) //d	movq (y), %rax //e movq (x), %rbx //f

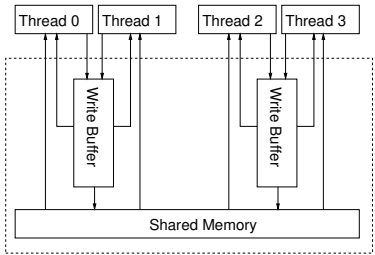
Final: 1:rax=1; 1:rbx=0; 3:rax=1; 3:rbx=0;

Observation: 0/100000000



- is that allowed in the previous model sketch?
- we think the T2,3 read pairs are not reorderable – so no
- is it microarchitecturally plausible? yes, e.g. with shared store buffers or fancy cache protocols
- experimentally: not observed
- architecture prose and intent?

  Any two stores are seen in a consistent order by processors other than those performing the stores; Example 8-7



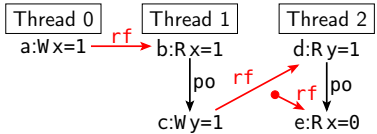
WRC x86

Initial state: `1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;`

Thread 0	Thread 1	Thread 2
`movq $1, (x) //a`	`movq (x), %rax //b` `movq $1, (y) //c`	`movq (y), %rax //d` `movq (x), %rbx //e`

Final: `1:rax=1; 2:rax=1; 2:rbx=0;`

`Observation: 0/100000000`

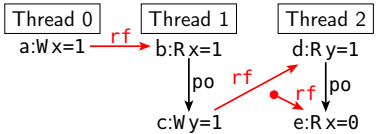


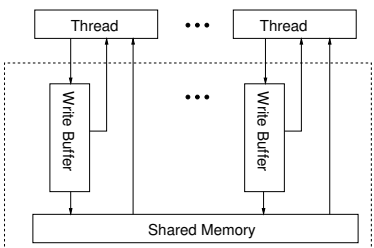
WRC x86

Initial state: 1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;		
Thread 0	Thread 1	Thread 2
`movq $1, (x) //a`	`movq (x), %rax //b` `movq $1, (y) //c`	`movq (y), %rax //d` `movq (x), %rbx //e`
Final: 1:rax=1; 2:rax=1; 2:rbx=0;		

`Observation: 0/100000000`

- is that allowed in the previous model sketch?

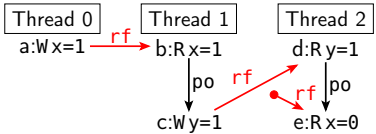




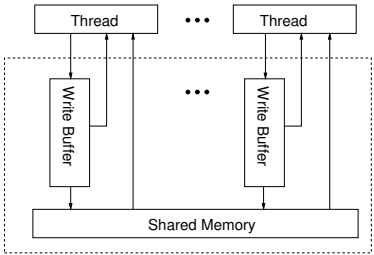
WRC x86

Initial state: 1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;		
Thread 0	Thread 1	Thread 2
movq $1, (x) //a	movq (x), %rax //b movq $1, (y) //c	movq (y), %rax //d movq (x), %rbx //e
Final: 1:rax=1; 2:rax=1; 2:rbx=0;		

Observation: 0/100000000



- is that allowed in the previous model sketch?
- we think the T1 read-write pair and T2 read pair are not reorderable – so no



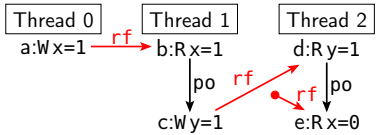
WRC x86

Initial state: `1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;`

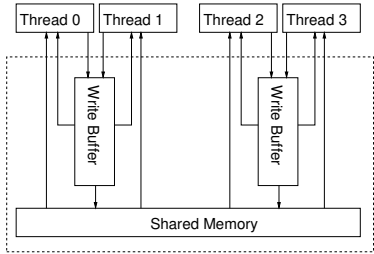
Thread 0	Thread 1	Thread 2
`movq $1, (x) //a`	`movq (x), %rax //b` `movq $1, (y) //c`	`movq (y), %rax //d` `movq (x), %rbx //e`

Final: `1:rax=1; 2:rax=1; 2:rbx=0;`

`Observation: 0/100000000`



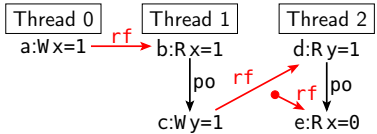
- is that allowed in the previous model sketch?
- we think the T1 read-write pair and T2 read pair are not reorderable – so no
  or in this one?



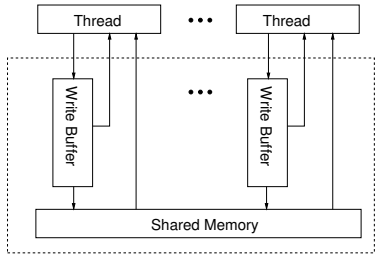
WRC x86

Initial state: 1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;		
Thread 0	Thread 1	Thread 2
movq $1, (x) //a	movq (x), %rax //b movq $1, (y) //c	movq (y), %rax //d movq (x), %rbx //e
Final: 1:rax=1; 2:rax=1; 2:rbx=0;		

Observation: 0/100000000



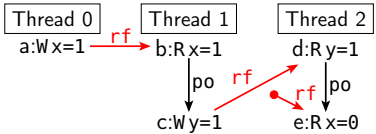
- is that allowed in the previous model sketch?
- we think the T1 read-write pair and T2 read pair are not reorderable – so no
- experimentally: not observed



WRC x86

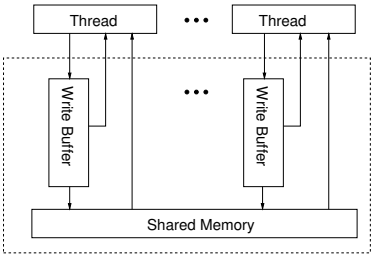
Initial state: 1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;		
Thread 0	Thread 1	Thread 2
`movq $1, (x) //a`	`movq (x), %rax //b` `movq $1, (y) //c`	`movq (y), %rax //d` `movq (x), %rbx //e`
Final: 1:rax=1; 2:rax=1; 2:rbx=0;		

`Observation: 0/100000000`



- is that allowed in the previous model sketch?
- we think the T1 read-write pair and T2 read pair are not reorderable – so no
- experimentally: not observed
- architecture prose and intent?

  Memory ordering obeys causality (memory ordering respects transitive visibility). Example 8-5



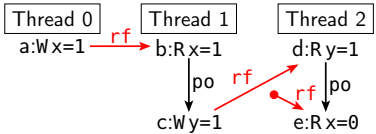
WRC x86

Initial state: `1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;`

Thread 0	Thread 1	Thread 2
`movq $1, (x) //a`	`movq (x), %rax //b` `movq $1, (y) //c`	`movq (y), %rax //d` `movq (x), %rbx //e`

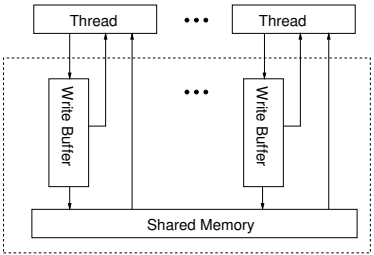
Final: `1:rax=1; 2:rax=1; 2:rbx=0;`

`Observation: 0/100000000`



- is that allowed in the previous model sketch?
- we think the T1 read-write pair and T2 read pair are not reorderable – so no
- experimentally: not observed
- architecture prose and intent?

  Memory ordering obeys causality (memory ordering respects transitive visibility). Example 8-5
- model sketch remains experimentally plausible, but interpretation of vendor prose unclear



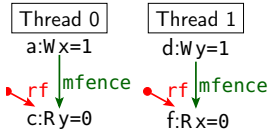
SB+mfences x86

Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//d`
`mfence`	`//b`	**`mfence`**	`//e`
`movq (y), %rax`	`//c`	`movq (x), %rax`	`//f`

Final: `0:rax=0; 1:rax=0;`

`Observation: 0/100000000`



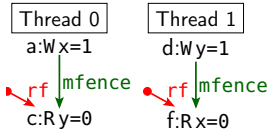
SB+mfences x86

Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//d`
`mfence`	`//b`	**`mfence`**	`//e`
`movq (y), %rax`	`//c`	`movq (x), %rax`	`//f`

Final: `0:rax=0; 1:rax=0;`

`Observation: 0/100000000`



- experimentally: not observed

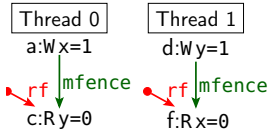
SB+mfences x86

Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//d`
`mfence`	`//b`	**`mfence`**	`//e`
`movq (y), %rax`	`//c`	`movq (x), %rax`	`//f`

Final: `0:rax=0; 1:rax=0;`

`Observation: 0/100000000`



- experimentally: not observed
- architecture prose and intent?

> *Reads and writes cannot pass earlier MFENCE instructions. MFENCE instructions cannot pass earlier reads or writes.*
> *MFENCE serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.*

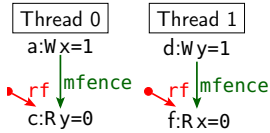
SB+mfences x86

Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//d`
`mfence`	`//b`	**`mfence`**	`//e`
`movq (y), %rax`	`//c`	`movq (x), %rax`	`//f`

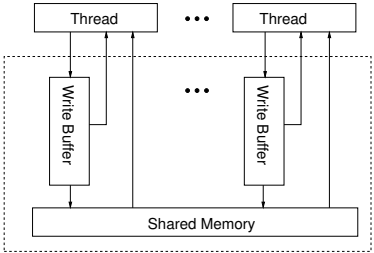
Final: `0:rax=0; 1:rax=0;`

`Observation: 0/100000000`



- experimentally: not observed
- architecture prose and intent?

  Reads and writes cannot pass earlier MFENCE instructions. MFENCE instructions cannot pass earlier reads or writes.
  MFENCE serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.



- in the model sketch: ...waits for local write buffer to drain? (or forces it to – it that observable?)
  NB: no inter-thread synchronisation

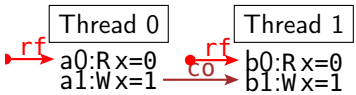
Adding Read-Modify-Write instructions

x86 is not RISC – there are many instructions that read and write memory, e.g.

INC x86

Initial state: x=0;	
Thread 0	Thread 1
incq (x)//a0,a1	incq (x)//b0,b1
Final: x=1;	

Observation: 1441/1000000



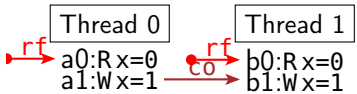
Adding Read-Modify-Write instructions

x86 is not RISC – there are many instructions that read and write memory, e.g.

INC x86

Initial state: x=0;	
Thread 0	Thread 1
incq (x)//a0,a1	incq (x)//b0,b1
Final: x=1;	

Observation: 1441/1000000



Non-atomic (even in SC semantics)

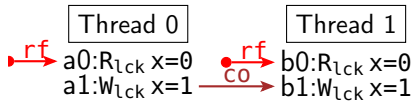
Adding Read-Modify-Write instructions

One can add the LOCK prefix (literally a one-byte opcode prefix) to make INC atomic

LOCKINC x86

Initial state: x=0;	
Thread 0	Thread 1
lock incq (x) //a0,a1	lock incq (x) //b0,b1
Final: x=1;	

Observation: 0/1000000



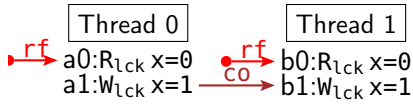
Adding Read-Modify-Write instructions

One can add the LOCK prefix (literally a one-byte opcode prefix) to make INC atomic

LOCKINC x86

Initial state: x=0;	
Thread 0	Thread 1
`lock incq (x) //a0,a1`	`lock incq (x) //b0,b1`
Final: x=1;	

`Observation: 0/1000000`



Also LOCK'd add, sub, xchg, etc., and cmpxchg

Being able to do that atomically is important for many low-level algorithms. On x86 can also do for other sizes, including for 8B and 16B adjacent-doublesize quantities

In early hardware implementations, this would literally lock the bus. Now, interconnects are much fancier.

CAS

Compare-and-swap (CAS):

```
lock cmpxchgq src, dest
```

compares `rax` with `dest`, then:

- if equal, set ZF=1 and load `src` into `dest`,
- otherwise, clear ZF=0 and load `dest` into `rax`

All this is one *atomic* step.

Can use to solve *consensus* problem...

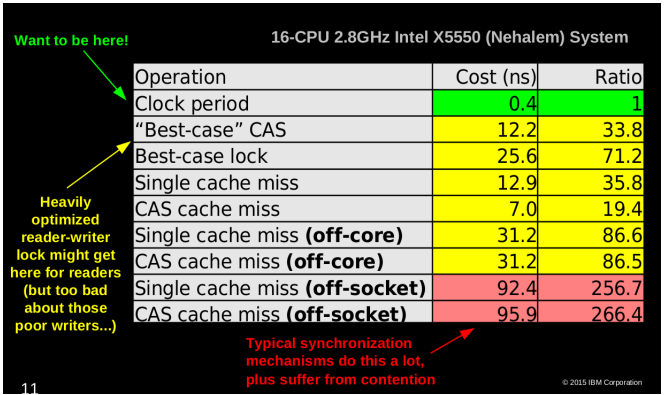
Synchronising power of locked instructions

"Loads and stores are not reordered with locked instructions"
Intel Example 8-9: SB with xchg for the stores, forbidden
Intel Example 8-10: MP with xchg for the first store, forbidden

"Locked instructions have a total order"
Intel Example 8-8: IRIW with xchg for the stores, forbidden

A rough guide to synchronisation costs

The costs of operations can vary widely between implementations and workloads, but for a very rough intuition, from Paul McKenney (`http://www2.rdrop.com/~paulmck/RCU/`):



Operation	Cost (ns)	Ratio
Clock period	0.4	1
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss **(off-core)**	31.2	86.6
CAS cache miss **(off-core)**	31.2	86.5
Single cache miss **(off-socket)**	92.4	256.7
CAS cache miss **(off-socket)**	95.9	266.4

See Tim Harris's lectures for more serious treatment of performance

Creating a usable model

History of x86 concurrency specs

- Before Aug. 2007 (Era of Vagueness): A Cautionary Tale

History of x86 concurrency specs

- Before Aug. 2007 (Era of Vagueness): A Cautionary Tale

Intel 'Processor Ordering' model, informal prose

Example: Linux Kernel mailing list, Nov–Dec 1999 (143 posts)

Keywords: speculation, ordering, cache, retire, causality

A one-instruction programming question; a microarchitectural debate!

1. spin_unlock() Optimization On Intel

20 Nov 1999 - 7 Dec 1999 (143 posts) Archive Link: "spin_unlock optimization(i386)"

Topics: BSD: FreeBSD, SMP

People: Linus Torvalds, Jeff V. Merkey, Erich Boleyn, Manfred Spraul, Peter Samuelson, Ingo Molnar

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "`lock; btrl $0,%0`" asm code, to 1 tick for a simple "`movl $0,%0`" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

> It does NOT WORK!
> Let the FreeBSD people use it, and let them get faster timings. They will crash, eventually.
> The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.
> The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).
> The issue is that you _have_ to have a serializing instruction in order to make sure that the processor doesn't re-order things around the unlock.
> For example, with a simple write, the CPU can legally delay a read that happened inside the critical region (maybe it missed a cache line), and get a stale value for any of the reads that _should_ have been serialized by the spinlock.
> Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

History of x86 concurrency

- Before Aug. 2007

Resolved only by appeal to an oracle:

pins, reads (cache miss) and writes appear in-order." He concluded from this that the second CPU would never see the spin_unlock() before the "b=a" line. Linus agreed that on a Pentium, Manfred was right. However, he quoted in turn from the Pentium Pro manual, "The only enhancement in the PentiumPro processor is the added support for speculative reads and store-buffer forwarding." He explained:

> A Pentium is a in-order machine, without any of the interesting speculation wrt reads etc. So on a Pentium you'll never see the problem.
>
> But a Pentium is also very uninteresting from a SMP standpoint these days. It's just too weak with too little per-CPU cache etc..
>
> This is why the PPro has the MTRR's - exactly to let the core do speculation (a Pentium doesn't need MTRR's, as it won't re-order anything external to the CPU anyway, and in fact won't even re-order things internally).

Jeff V. Merkey added:

> What Linus says here is correct for PPro and above. Using a mov instruction to unlock does work fine on a 486 or Pentium SMP system, but as of the PPro, this was no longer the case, though the window is so infintesimally small, most kernels don't hit it (Netware 4/5 uses this method but it's spinlocks understand this and the code is writtne to handle it. The most obvious aberrant behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the piplines are no longer invalid and the buffers should be blown out.
>
> I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD people must still be on older Pentium hardware and that's why they don't know this can bite in some cases.

Erich Boleyn, an Architect in an IA32 development group at Intel, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding the code Linus posted, Erich replied:

> It will always return 0. You don't need "`spin_unlock()`" to be serializing.
>
> The only thing you need is to make sure there is a store in "`spin_unlock()`", and that is kind of true by the fact that you're changing something to be observable on other processors.

History of x86 concurrency specs

- Before Aug. 2007 (Era of Vagueness): A Cautionary Tale

History of x86 concurrency specs

- Before Aug. 2007 (Era of Vagueness): A Cautionary Tale
- IWP and AMD64, Aug. 2007/Oct. 2008 (Era of Causality)

  Intel published a white paper (IWP) defining 8 informal-prose principles, e.g.

  P1 Loads are not reordered with older loads

  P2 Stores are not reordered with older stores

  P5 Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation

  supported by 10 litmus tests illustrating allowed or forbidden behaviours.

History of x86 concurrency specs

- Before Aug. 2007 (Era of Vagueness): A Cautionary Tale
- IWP and AMD64, Aug. 2007/Oct. 2008 (Era of Causality)
  Intel published a white paper (IWP) defining 8 informal-prose principles, e.g.
  - P1 Loads are not reordered with older loads
  - P2 Stores are not reordered with older stores
  - P5 Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation

  supported by 10 litmus tests illustrating allowed or forbidden behaviours.
- We codify these principles in an axiomatic model, x86-CC [1, POPL 2009]

History of x86 concurrency specs

- Before Aug. 2007 (Era of Vagueness): A Cautionary Tale
- IWP and AMD64, Aug. 2007/Oct. 2008 (Era of Causality)
  Intel published a white paper (IWP) defining 8 informal-prose principles, e.g.
  - P1 Loads are not reordered with older loads
  - P2 Stores are not reordered with older stores
  - P5 Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation

  supported by 10 litmus tests illustrating allowed or forbidden behaviours.
- We codify these principles in an axiomatic model, x86-CC [1, POPL 2009]

But there are problems:

1. the principles are ambiguous (we interpret them as w.r.t. a single causal order)
2. the principles (and our model) leave IRIW allowed, even with `mfences`, but the Sun implementation of the Java Memory Model assumes that `mfences` recovers SC
3. the model is unsound w.r.t. observable behaviour, as noted by Paul Loewenstein, with an example that is allowed in the store-buffer model

History of x86 concurrency specs

- Intel SDM rev.27– and AMD 3.17–, Nov. 2008–

  Now explicitly excludes IRIW:
  - Any two stores are seen in a consistent order by processors other than those performing the stores

  But, still ambiguous w.r.t. causality, and the view by those processors is left unspecified

Creating a good x86 concurrency model

We had to *create* a good concurrency model for x86 – "good" meaning the desirable properties listed before

Key facts:

- Store buffering (with forwarding) is observable
- These store buffers appear to be FIFO
- We don't see observable buffering of read requests
- We don't see other observable out-of-order or speculative execution
- IRIW and WRC not observable, and now forbidden by the docs – so *multicopy atomic*
- `mfence` appears to wait for the local store buffer to drain
- as do LOCK'd instructions, before they execute
- Various other reorderings are not observable and are forbidden

These suggested that x86 is, in practice, like SPARC TSO: the observable effects of store buffers are the only observable relaxed-memory behaviour

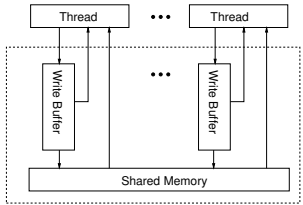
Our **x86-TSO** model codifies this, adapting SPARC TSO
Owens, Sarkar, Sewell [4, TPHOLs 2009] [5, CACM 2010]

Operational and axiomatic concurrency model definitions

Two styles:

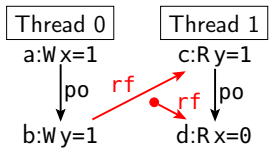
Operational

- an *abstract machine*
- incrementally executable
- often *abstract-microarchitectural operational models*



Axiomatic

- a *predicate on candidate executions*
- usually (but not always) further from microarchitecture (more concise, but less hardware intuition)
- not straightforwardly incrementally executable

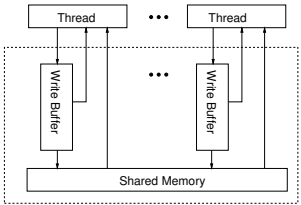


Operational and axiomatic concurrency model definitions

Two styles:

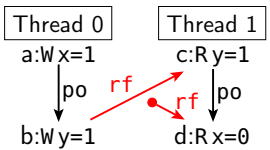
Operational

- an *abstract machine*
- incrementally executable
- often *abstract-microarchitectural operational models*



Axiomatic

- a *predicate on candidate executions*
- usually (but not always) further from microarchitecture (more concise, but less hardware intuition)
- not straightforwardly incrementally executable

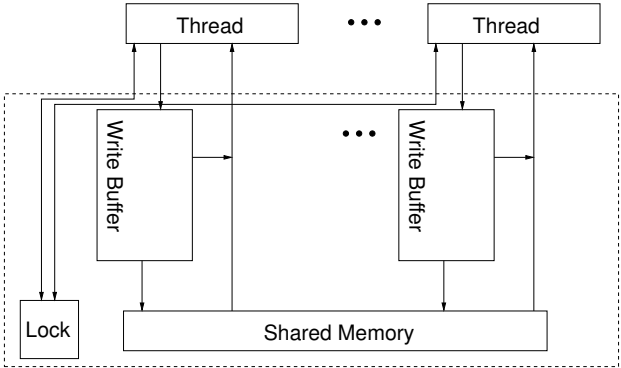


Ideally both, proven equivalent

x86-TSO operational model

x86-TSO Abstract Machine

Like the sketch except with state recording which (if any) thread has the machine lock



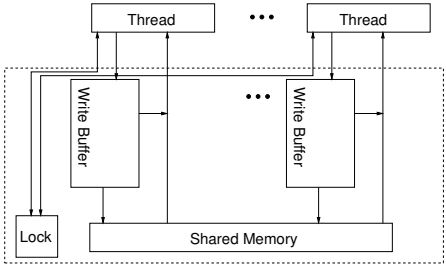
x86-TSO Abstract Machine

We factor the model into the *thread semantics* and the *memory model*.

The x86-TSO thread semantics just executes each instruction in program order

The whole machine is modelled as a parallel composition of the thread semantics (for each thread) and the x86-TSO memory-model abstract machine...

...exchanging messages for reads, writes, barriers, and machine lock/unlock events



x86-TSO Abstract Machine: Memory Behaviour

We formalise the x86-TSO memory-model abstract machine as a transition system

$$m \xrightarrow{e} m'$$

Read as: memory in state m can do a transition with event e to memory state m'

x86-TSO Abstract Machine: threads/memory interface

Events e	::=	$a{:}t{:}\mathrm{W}\,x{=}v$	a write of value v to address x by thread t, ID a
	|	$a{:}t{:}\mathrm{R}\,x{=}v$	a read of v from x by t
	|	$a{:}t{:}\mathrm{D}_w\,x{=}v$	an internal action of the abstract machine, dequeuing $w = (a'{:}t{:}\mathrm{W}\,x{=}v)$ from thread t's write buffer to shared memory
	|	$a{:}t{:}\mathrm{F}$	an `MFENCE` memory barrier by t
	|	$a{:}t{:}\mathrm{L}$	start of an instruction with `LOCK` prefix by t
	|	$a{:}t{:}\mathrm{U}$	end of an instruction with `LOCK` prefix by t

where

- a is a unique event ID, of type *eid*
- t is a hardware thread id, of type *tid*
- x and y are memory addresses, of type *addr*
- v and w are memory values, of type *value*
- w is a write event $a{:}t{:}\mathrm{W}\,x{=}v$, of type *write_event*

x86-TSO Abstract Machine: Memory States

An x86-TSO abstract-machine memory state m is a record with fields M, B, and L:

$$m : \langle M : addr \to value;\\ B : tid \to write\_event\ list;\\ L : tid\ option \rangle$$

Here:

- $m.M$ is the shared memory, mapping addresses to values
- $m.B$ gives the store buffer for each thread, a list of write events, most recent first (we use a list of write events for simplicity in proofs, but the event and thread IDs are erasable)
- $m.L$ is the global machine lock, indicating when some thread has exclusive access to memory. It is a *tid option*, either None, or Some t for some thread ID t

The initial state m_{init} has $m_{\mathrm{init}}.M$ zero for each address, $m_{\mathrm{init}}.B$ empty for all threads, and $m_{\mathrm{init}}.L =$ None (lock not taken).

Notation

Some and None construct optional values

$(\cdot, \cdot)$ builds tuples

[] builds lists

@ appends lists

$\cdot \oplus \langle \cdot := \cdot \rangle$ updates records

$\cdot \oplus (\cdot \mapsto \cdot)$ updates functions.

$\mathrm{id}(e)$, $\mathrm{thread}(e)$, $\mathrm{addr}(e)$, $\mathrm{value}(e)$ extract the respective components of event e

$\mathrm{isread}(e)$, $\mathrm{iswrite}(e)$, $\mathrm{isdequeue}(e)$, $\mathrm{ismfence}(e)$ identify the corresponding kinds

x86-TSO Abstract Machine: Auxiliary Definitions

Say there are *no pending* writes in t's buffer $m.B(t)$ for address x if there are no write events w in $m.B(t)$ with $\mathrm{addr}(w) = x$.

Say t is *not blocked* in machine state m if either it holds the lock ($m.L = \mathrm{Some}\, t$) or the lock is not held ($m.L = \mathrm{None}$).

x86-TSO Abstract Machine: Behaviour

RM: Read from memory

$$\frac{\begin{array}{l}\mathrm{not\_blocked}(m, t)\\ m.M(x) = v\\ \mathrm{no\_pending}(m.B(t), x)\end{array}}{m \xrightarrow{a{:}t{:}\mathrm{R}\,x=v} m}$$

Thread t can read v from memory at address x if t is not blocked, the memory does contain v at x, and there are no writes to x in t's store buffer.

(the event ID a is left unconstrained by the rule)

x86-TSO Abstract Machine: Behaviour

RB: Read from write buffer

$$\frac{\begin{array}{l}\mathrm{not\_blocked}(m, t)\\ \exists a'\ b_1\ b_2.\ m.B(t) = b_1 \mathbin{@} [a'{:}t{:}\mathrm{W}\ x{=}v] \mathbin{@} b_2\\ \mathrm{no\_pending}(b_1, x)\end{array}}{m \xrightarrow{a:t:\mathrm{R}\ x=v} m}$$

Thread t can read v from its store buffer for address x if t is not blocked and has v as the value of the most recent write to x in its buffer.

x86-TSO Abstract Machine: Behaviour

WB: Write to write buffer

$$m \xrightarrow{a{:}t{:}\mathrm{W}\,x{=}v} m \oplus \langle B := m.B \oplus (t \mapsto ([a{:}t{:}\mathrm{W}\,x{=}v] \mathbin{@} m.B(t)))\rangle$$

Thread t can write v to its store buffer for address x at any time.

x86-TSO Abstract Machine: Behaviour

DM: Dequeue write from write buffer to memory

$$\frac{\begin{array}{l}\mathrm{not\_blocked}(m, t)\\ m.B(t) = b \mathbin{@} [a'{:}t{:}\mathrm{W}\, x{=}v]\end{array}}{m \xrightarrow{a:t:\mathrm{D}_{a':t:\mathrm{W}\, x=v}\; x=v} m \oplus \langle M := m.M \oplus (x \mapsto v)\rangle \oplus \langle B := m.B \oplus (t \mapsto b)\rangle}$$

If Thread t is not blocked, it can silently dequeue the oldest write from its store buffer and update memory at that address with the new value, without coordinating with any hardware thread.

(we record the write in the dequeue event just to simplify proofs.)

x86-TSO Abstract Machine: Behaviour

M: MFENCE

$$\frac{m.B(t) = [\,]}{m \xrightarrow{a{:}t{:}\mathsf{F}} m}$$

If Thread t's store buffer is empty, it can execute an `MFENCE` (otherwise the `MFENCE` blocks until that becomes true).

Adding LOCK'd instructions to the model

We define the instruction semantics for locked instructions to bracket the transitions of their unlocked variant with $a{:}t{:}\mathrm{L}$ and $a'{:}t{:}\mathrm{U}$.
For example, a `lock inc x`, in thread t, will do

1. $a_1{:}t{:}\mathrm{L}$
2. $a_2{:}t{:}\mathrm{R}\, x{=}v$ for an arbitrary v
3. $a_3{:}t{:}\mathrm{W}\, x{=}(v+1)$
4. $a_4{:}t{:}\mathrm{U}$

This lets us reuse the `inc` semantics for `lock inc`, and to do so uniformly for all RMWs.

x86-TSO Abstract Machine: Behaviour

L: Lock

$$\frac{m.L = \text{None} \qquad m.B(t) = [\,]}{m \xrightarrow{a{:}t{:}\mathrm{L}} m \oplus \langle L := \text{Some}(t) \rangle}$$

If the lock is not held and its buffer is empty, thread t can begin a LOCK'd instruction.

Note that if a hardware thread t comes to a LOCK'd instruction when its store buffer is not empty, the machine can take one or more $a{:}t{:}\mathrm{D}_w\ x{=}v$ steps to empty the buffer and then proceed.

x86-TSO Abstract Machine: Behaviour

U: Unlock

$$\frac{\begin{array}{l} m.L = \mathrm{Some}(t) \\ m.B(t) = [\,] \end{array}}{m \xrightarrow{a{:}t{:}\mathrm{U}} m \oplus \langle L := \mathrm{None} \rangle}$$

If t holds the lock, and its store buffer is empty, it can end a LOCK'd instruction, resetting the lock.

First Example, Revisited

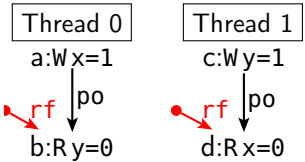
SB x86

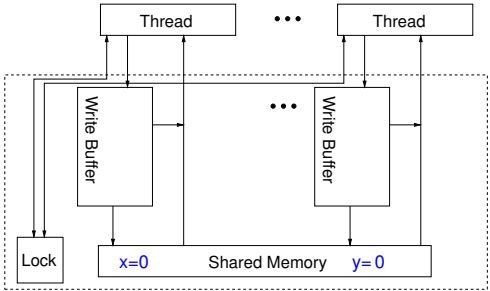
Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//c`
`movq (y), %rax`	`//b`	`movq (x), %rax`	`//d`

Final: `0:rax=0; 1:rax=0;`

Observation: 171/100000000





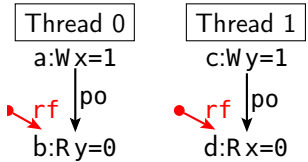
m_{init}

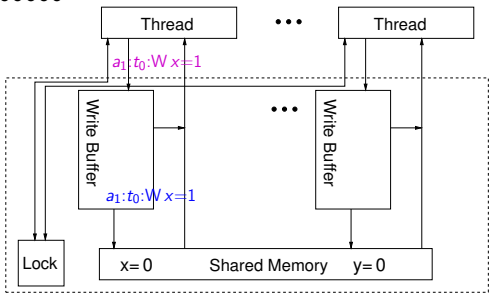
First Example, Revisited

SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	**Thread 1**
`movq $1, (x)` //a `movq (y), %rax` //b	`movq $1, (y)` //c `movq (x), %rax` //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000





$m_{\text{init}} \xrightarrow{a_1:t_0:\mathrm{W}\,x=1}$

First Example, Revisited

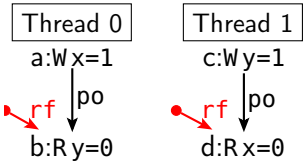
SB x86

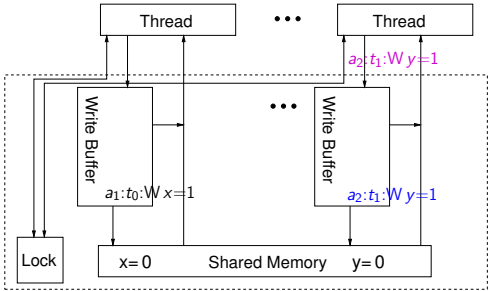
Initial state: `0:rax=0; 1:rax=0; y=0; x=0;`

Thread 0		Thread 1	
`movq $1, (x)`	`//a`	`movq $1, (y)`	`//c`
`movq (y), %rax`	`//b`	`movq (x), %rax`	`//d`

Final: `0:rax=0; 1:rax=0;`

`Observation: 171/100000000`





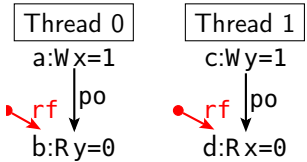
$$m_{\text{init}} \xrightarrow{a_1{:}t_0{:}\mathrm{W}\,x=1} \xrightarrow{a_2{:}t_1{:}\mathrm{W}\,y=1}$$

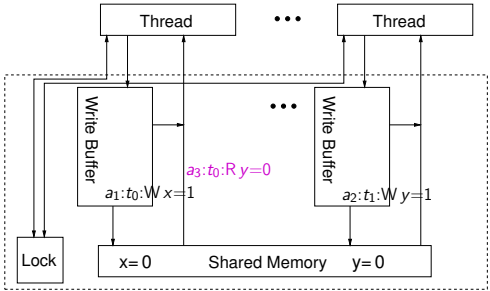
First Example, Revisited

SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	**Thread 1**
movq $1, (x) //a movq (y), %rax //b	movq $1, (y) //c movq (x), %rax //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000





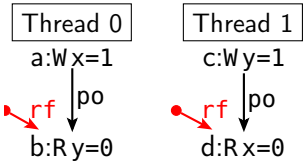
$$m_{\text{init}} \xrightarrow{a_1:t_0:\text{W}\,x=1} \xrightarrow{a_2:t_1:\text{W}\,y=1} \xrightarrow{a_3:t_0:\text{R}\,y=0}$$

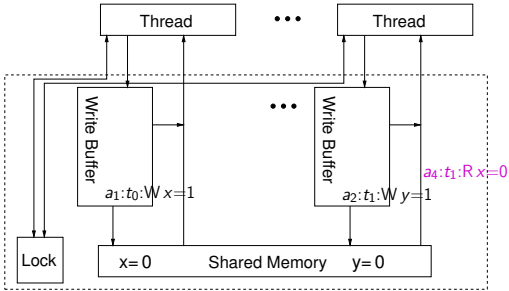
First Example, Revisited

SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	**Thread 1**
movq $1, (x) //a movq (y), %rax //b	movq $1, (y) //c movq (x), %rax //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000





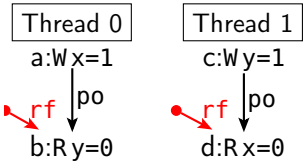
$$m_{\text{init}} \xrightarrow{a_1:t_0:\mathrm{W}\,x=1} \xrightarrow{a_2:t_1:\mathrm{W}\,y=1} \xrightarrow{a_3:t_0:\mathrm{R}\,y=0} \xrightarrow{a_4:t_1:\mathrm{R}\,x=0}$$

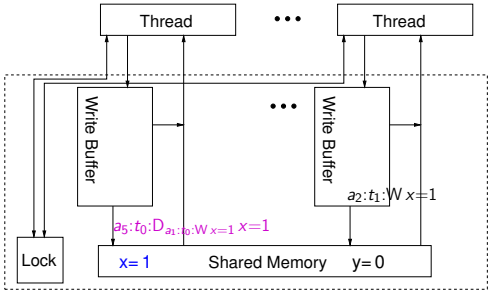
First Example, Revisited

SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	**Thread 1**
movq $1, (x) //a	movq $1, (y) //c
movq (y), %rax //b	movq (x), %rax //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000





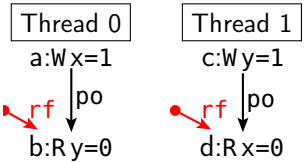
$m_{\text{init}} \xrightarrow{a_1:t_0:\text{W}\,x=1} \xrightarrow{a_2:t_1:\text{W}\,y=1} \xrightarrow{a_3:t_0:\text{R}\,y=0} \xrightarrow{a_4:t_1:\text{R}\,x=0} \xrightarrow{a_5:t_0:\text{D}_{a_1:t_0:\text{W}\,x=1}\,x=1}$

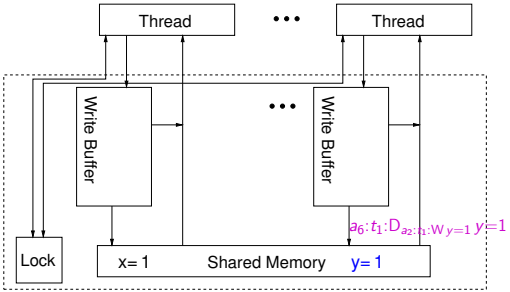
First Example, Revisited

SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	**Thread 1**
`movq $1, (x)` //a	`movq $1, (y)` //c
`movq (y), %rax` //b	`movq (x), %rax` //d
Final: 0:rax=0; 1:rax=0;	

Observation: 171/100000000





$m_{\text{init}} \xrightarrow{a_1{:}t_0{:}\mathrm{W}\,x=1} \xrightarrow{a_2{:}t_1{:}\mathrm{W}\,y=1} \xrightarrow{a_3{:}t_0{:}\mathrm{R}\,y=0} \xrightarrow{a_4{:}t_1{:}\mathrm{R}\,x=0} \xrightarrow{a_5{:}t_0{:}\mathrm{D}_{a_1{:}t_0{:}\mathrm{W}\,x=1}\,x=1} \xrightarrow{a_6{:}t_1{:}\mathrm{D}_{a_2{:}t_1{:}\mathrm{W}\,y=1}\,y=1}$

Does MFENCE restore SC?

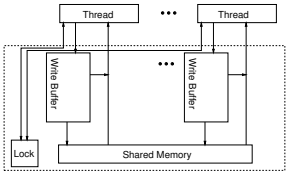
Intuitively, if the program executed by the thread semantics has an `mfence` between every pair of memory accesses, then any execution in x86-TSO will have essentially identical behaviour to the same program with `nop`s in place of `mfence`s in SC.

What does "essentially identical" mean? The same set of interface traces except with the $a{:}t{:}\mathrm{F}$ and $a{:}t{:}\mathrm{D}_w\, x{=}v$ events erased.

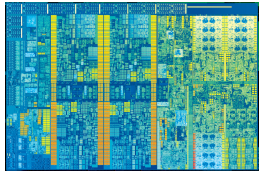
Restoring SC with RMWs

NB: This is an *Abstract* Machine

A tool to specify exactly and only the *programmer-visible behavior*, based on hardware intuition, but not a description of real implementation internals



$\supseteq_{\text{beh}}$
$\neq_{\text{hw}}$



Force: Of the internal optimizations of x86 processors, *only* per-thread FIFO write buffers are (ignoring timing) visible to programmers.

Still quite a loose spec: unbounded buffers, nondeterministic unbuffering, arbitrary interleaving

Remark: Processors, Hardware Threads, and Threads

Our ‘Threads’ are hardware threads.

Some processors have *simultaneous multithreading* (Intel: hyperthreading): multiple hardware threads/core sharing resources.

If the OS flushes store buffers on context switch (for x86 – or does whatever synchronisation is needed on other archs), software threads should have the same semantics as hardware threads.

x86-TSO vs SPARC TSO

x86-TSO based on SPARC TSO

SPARC defined

- TSO (Total Store Order)
- PSO (Partial Store Order)
- RMO (Relaxed Memory Order)

But as far as we know, only TSO has really been used (implementations have not been as weak as PSO/RMO or software has turned those off).

- The SPARC Architecture Manual, Version 8, Revision SAV080SI9308. 1992.
  http://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz App. K defines TSO and PSO.
- The SPARC Architecture Manual, Version 9, Revision SAV09R1459912. 1994
  http://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz Ch. 8 and App. D define TSO, PSO, RMO

Those were in an axiomatic style – see later. x86-TSO is extensionally similar to SPARC TSO except for x86 RMW operations

This model (like other operational models) is an interleaving semantics, just like SC – but with finer-grain transitions, as we've split each memory write into two transitions

Reasoning that a particular final state is allowed by an operational model is easy: just exhibit a trace with that final state

Reasoning that some final state is *not* allowed requires reasoning about *all* model-allowed traces – either exhaustively, as we did for SC at the start, or in some smarter way.

Making x86-TSO executable as a test oracle: the RMEM tool

RMEM is a tool letting one interactively or exhaustively explore the operational models for x86, Armv8-A, IBM Power, and RISC-V. (Flur, Pulte, Sarkar, Sewell, et al. [28]).

Either use the in-browser web interface: `http://www.cl.cam.ac.uk/users/pes20/rmem` or install locally and use the CLI interface (better performance), following: `https://github.com/rems-project/rmem`

Go to the web interface, load an x86 litmus test, set the "All eager" execution option, then click the allowed x86-TSO transitions to explore interactively

State

100%

Storage subsystem state (TSO):

```
0           1
|           |
|           |
|           |
-------------
      |
      |
      |
```

Memory = [(1000:0:0):W 0x1000 (y)/8=0, (1000:1:0):W 0x1100 (x)/8=0]
Lock = unlocked

Thread 0 state:

0:1 0x050000 fetched movq 1, (%rax) reg reads: RAX=0x_63'0000000000001100 (x) from initialstate
micro_op_state: MOS_pending_mem_write
(0:1:0):W 0x1100 (x)/8=1
0 0:1 propagate memory write to storage: (0:1:0):W 0x1100 (x)/8=1

Thread 1 state:

1:1 0x051000 fetched movq 1, (%rax) reg reads: RAX=0x_63'0000000000001000 (y) from initialstate
micro_op_state: MOS_pending_mem_write
(1:1:0):W 0x1000 (y)/8=1
1 1:1 propagate memory write to storage: (1:1:0):W 0x1000 (y)/8=1

Choices so far (0):
2 enabled transitions
No disabled transitions

Graph

Refresh | Download .dot | centre | 100%

Sources

Download | Edit | 100%

X86 SB

```
1  X86 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5  uint64_t x=0;
6  uint64_t y=0;
7  0:rax=x; 0:rbx=y; 1:rax=y; 1:rbx=x
8  }
9   P0                 | P1                 ;
10 mov $1,(%rax)      | mov $1,(%rax)  ;
11 mov (%rbx),%rcx    | mov (%rbx),%rcx ;
12 exists
13 (0:rcx=0 /\ 1:rcx=0)
```

Console

100%

```
Step 1 (0/2 finished, 0 trns) Choose [0]: 0
Step 1 (0/2 finished, 0 trns) Choose [0]:
```

Help

Open in new tab | 100%

RMEM

Help Contents

RMEM X86 SB | Load litmus | Load ELF | Model | Next | Back | Restart | Search | Execution | Interface | Graph | Link to this state | Help

State — 100%

Storage subsystem state (TSO):

```
0                               1
|  (0:1:0):W 0x1100 (x)/8=1  |
|                            |
|                            |
-----------------------------
              |
              |
              |
```

Memory = [(1000:0:0):W 0x1000 (y)/8=0, (1000:1:0):W 0x1100 (x)/8=0]
Lock = unlocked
2 propagate write to memory: (0:1:0):W 0x1100 (x)/8=1

Thread 0 state:

0:1 0x050000 **fetched movq 1, (%rax)** mem writes: (0:1:0):W 0x1100 (x)/8=1 reg reads:
RAX=0x_63'0000000000001100 (x) from initialstate
micro_op_state: MOS_plain
[_]
[_]
[()]
Env: addr=0b0000000000000000000000000000000000000000000000000000000000100000000, data=0b0000000000000000000000000000000000000000000000000000000000000001, len=8, locked=0
0:2 0x050004 fetched movq (%rbx), %rcx mem reads: (0:2:0):R 0x1000 (y)/8 from [] reg reads:
RBX=0x_63'0000000000001000 (y) from initialstate
micro_op_state: MOS_pending_mem_read
unsatisfied slices: [(0:2:0):R 0x1000 (y)/8: [0-7]]
writes read from: [(0:2:0):R 0x1000 (y)/8 from []]
requested slices: []
0 0:2 satisfy memory read from memory: (0:2:0):R 0x1000 (y)/8 = [0 from (1000:0:0):W 0x1000 (y)/8=0]

Thread 1 state:

1:1 0x051000 fetched movq 1, (%rax) reg reads: RAX=0x_63'0000000000001000 (y) from initialstate
micro_op_state: MOS_pending_mem_write
(1:1:0):W 0x1000 (y)/8=1
1 1:1 propagate memory write to storage: (1:1:0):W 0x1000 (y)/8=1

Choices so far (1): 0
3 enabled transitions
No disabled transitions

Console — 100%

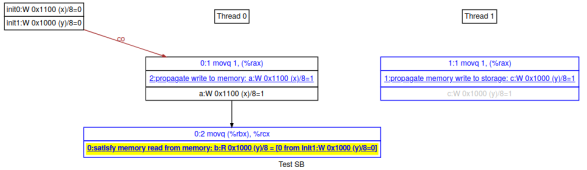
```
Step 1 (0/2 finished, 0 trns) Choose [0]: 0
Step 2 (1/3 finished, 30 trns) Choose [0]:
```

Help — Open in new tab — 100%

RMEM

Help Contents

Graph — Refresh | Download .dot | centre — 100% +



Sources — Download | Edit — 100%

X86 SB

```
1  X86 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5  uint64_t x=0;
6  uint64_t y=0;
7  0:rax=x; 0:rbx=y; 1:rax=y; 1:rbx=x
8  }
9   P0          | P1          ;
10 mov $1,(%rax)  | mov $1,(%rax)  ;
11 mov (%rbx),%rcx | mov (%rbx),%rcx ;
12 exists
13 (0:rcx=0 /\ 1:rcx=0)
```

State

Storage subsystem state (TSO):

```
0                                1
|  (0:1:0):W 0x1100 (x)/8=1  |  (1:1:0):W 0x1000 (y)/8=1
|                            |
|                            |
-------------------------------
               |
               |
               |
```

Memory = [(1000:0:0):W 0x1000 (y)/8=0, (1000:1:0):W 0x1100 (x)/8=0]
Lock = unlocked
2 propagate write to memory: (0:1:0):W 0x1100 (x)/8=1
3 propagate write to memory: (1:1:0):W 0x1000 (y)/8=1

Thread 0 state:
0:1 0x050000 **fetched movq 1, (%rax)** mem writes: (0:1:0):W 0x1100 (x)/8=1 reg reads:
RAX=0x_63'0000000000001100 (x) from initialstate
micro_op_state: MOS_plain
0:2 0x050004 fetched movq (%rbx), %rcx mem reads: (0:2:0):R 0x1000 (y)/8 from [] reg reads:
RBX=0x_63'0000000000001000 (y) from initialstate
micro_op_state: MOS_pending_mem_read
unsatisfied slices: [(0:2:0):R 0x1000 (y)/8: [0-7]]
writes read from: [(0:2:0):R 0x1000 (y)/8 from []]
requested slices: []
0 0:2 satisfy memory read from memory: (0:2:0):R 0x1000 (y)/8 = [0 from (1000:0:0):W 0x1000 (y)/8=0]

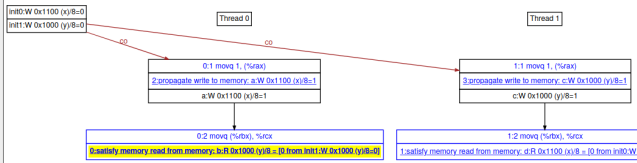
Thread 1 state:
1:1 0x051000 **fetched movq 1, (%rax)** mem writes: (1:1:0):W 0x1000 (y)/8=1 reg reads:
RAX=0x_63'0000000000001000 (y) from initialstate
micro_op_state: MOS_plain
| [_]
| ----------
| [_]
| ----------
| ()
Env: addr=0b0000000000000000000000000000000000000000000000000000000000001000000000000,
data=0b0000000000000000000000000000000000000000000000000000000000000001, len=8, locked=0
1:2 0x051004 fetched movq (%rbx), %rcx mem reads: (1:2:0):R 0x1100 (x)/8 from [] reg reads:
RBX=0x_63'0000000000001100 (x) from initialstate
micro_op_state: MOS_pending_mem_read
unsatisfied slices: [(1:2:0):R 0x1100 (x)/8: [0-7]]
writes read from: [(1:2:0):R 0x1100 (x)/8 from []]
requested slices: []
1 1:2 satisfy memory read from memory: (1:2:0):R 0x1100 (x)/8 = [0 from (1000:1:0):W 0x1100 (x)/8=0]

Choices so far (2): 0;1
4 enabled transitions
No disabled transitions

Console

Help — Open in new tab

Graph — Refresh | Download .dot | centre | 100%



Sources — Download | Edit | 100%

X86 SB

```
1  X86 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5  uint64_t x=0;
6  uint64_t y=0;
7  0:rax=x; 0:rbx=y; 1:rax=y; 1:rbx=x
8  }
9   P0          | P1          ;
10 mov $1,(%rax)  | mov $1,(%rax)  ;
11 mov (%rbx),%rcx | mov (%rbx),%rcx ;
12 exists
13 (0:rcx=0 /\ 1:rcx=0)
```

RMEM X86 SB | Load litmus | Load ELF | Model | Next | Back | Restart | Search | Execution | Interface | Graph | Link to this state | Help

State

100%

Storage subsystem state (TSO):

```
0                              1
|  (0:1:0):W 0x1100 (x)/8=1  |  (1:1:0):W 0x1000 (y)/8=1
|                            |
|                            |
-------------------------------
               |
               |
               |
```

Memory = [(1000:0:0):W 0x1000 (y)/8=0, (1000:1:0):W 0x1100 (x)/8=0]
Lock = unlocked
2 propagate write to memory: (0:1:0):W 0x1100 (x)/8=1
3 propagate write to memory: (1:1:0):W 0x1000 (y)/8=1

Thread 0 state:
0:1 0x050000 **fetched movq 1, (%rax)** mem writes: (0:1:0):W 0x1100 (x)/8=1 reg reads:
RAX=0x_63'0000000000001100 (x) from initialstate
micro_op_state: MOS_plain
0:2 0x050004 **fetched movq (%rbx), %rcx** mem reads: (0:2:0):R from (1000:0:0):W 0x1000 (y)/8=0 reg reads: RBX=0x_63'0000000000001000 (y) from initialstate reg writes: RCX=0x_63'0000000000000000
micro_op_state: MOS_plain
| [_]
| ----------
| ()
Env: e=Ea_r (Sz64 (),1), locked=0, r=1,
w=0b0000000000000000000000000000000000000000000000000000000000000000

Thread 1 state:
1:1 0x051000 **fetched movq 1, (%rax)** mem writes: (1:1:0):W 0x1000 (y)/8=1 reg reads:
RAX=0x_63'0000000000001000 (y) from initialstate
micro_op_state: MOS_plain
1:2 0x051004 fetched movq (%rbx), %rcx mem reads: (1:2:0):R 0x1100 (x)/8 from [] reg reads:
RBX=0x_63'0000000000001100 (x) from initialstate
micro_op_state: MOS_pending_mem_read
unsatisfied slices: [(1:2:0):R 0x1100 (x)/8: [0-7]]
writes read from: [(1:2:0):R 0x1100 (x)/8 from []]
requested slices: []
1 1:2 satisfy memory read from memory: (1:2:0):R 0x1100 (x)/8 = [0 from (1000:1:0):W 0x1100 (x)/8=0]

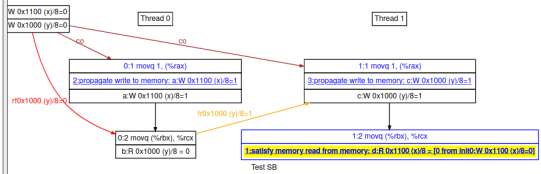
Choices so far (3): 0;1;0
3 enabled transitions
No disabled transitions

Console

Step 3 (2/4 finished, 60 trns) Choose [0]: 0

Graph

Refresh | Download .dot | centre | 100%



Sources

Download | Edit | 100%

X86 SB

```
1  X86 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5  uint64_t x=0;
6  uint64_t y=0;
7  0:rax=x; 0:rbx=y; 1:rax=y; 1:rbx=x
8  }
9   P0          | P1          ;
10 mov $1,(%rax)  | mov $1,(%rax)  ;
11 mov (%rbx),%rcx | mov (%rbx),%rcx ;
12 exists
13 (0:rcx=0 /\ 1:rcx=0)
```

Help

Open in new tab | 100%

RMEM X86 SB | Load litmus | Load ELF | Model | Next | Back | Restart | Search | Execution | Interface | Graph | Link to this state | Help

State

100%

Storage subsystem state (TSO):

```
0                              1
|  (0:1:0):W 0x1100 (x)/8=1  |  (1:1:0):W 0x1000 (y)/8=1
|                            |
|                            |
-------------------------------
                |
                |
                |
```

Memory = [(1000:0:0):W 0x1000 (y)/8=0, (1000:1:0):W 0x1100 (x)/8=0]
Lock = unlocked
2 propagate write to memory: (0:1:0):W 0x1100 (x)/8=1
3 propagate write to memory: (1:1:0):W 0x1000 (y)/8=1

Thread 0 state:
0:1 0x050000 **fetched movq 1, (%rax)** mem writes: (0:1:0):W 0x1100 (x)/8=1 reg reads:
RAX=0x_63'0000000000001100 (x) from initialstate
micro_op_state: MOS_plain
0:2 0x050004 **fetched movq (%rbx), %rcx** mem reads: (0:2:0):R from (1000:0:0):W 0x1000 (y)/8=0 reg reads: RBX=0x_63'0000000000001000 (y) from initialstate reg writes: RCX=0x_63'0000000000000000
micro_op_state: MOS_plain

Thread 1 state:
1:1 0x051000 **fetched movq 1, (%rax)** mem writes: (1:1:0):W 0x1000 (y)/8=1 reg reads:
RAX=0x_63'0000000000001000 (y) from initialstate
micro_op_state: MOS_plain
1:2 0x051004 **fetched movq (%rbx), %rcx** mem reads: (1:2:0):R from (1000:1:0):W 0x1100 (x)/8=0 reg reads: RBX=0x_63'0000000000001100 (x) from initialstate reg writes: RCX=0x_63'0000000000000000
micro_op_state: MOS_plain
| [_]
| --
| ()
Env: e=Ea_r (Sz64 (),1), locked=0, r=1,
w=0b0000000000000000000000000000000000000000000000000000000000000000

Choices so far (4): 0;1;0;1
2 enabled transitions
No disabled transitions

Console

Step 4 (3/4 finished, 66 trns) Choose [1]: 1

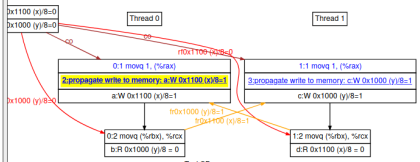
Help

Open in new tab

100%

Graph

Refresh | Download .dot | centre | 100%



Sources

Download | Edit | 100%

X86 SB

```
1  X86 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5  uint64_t x=0;
6  uint64_t y=0;
7  0:rax=x; 0:rbx=y; 1:rax=y; 1:rbx=x
8  }
9   P0          | P1          ;
10 mov $1,(%rax)  | mov $1,(%rax) ;
11 mov (%rbx),%rcx | mov (%rbx),%rcx ;
12 exists
13 (0:rcx=0 /\ 1:rcx=0)
```

RMEM X86 SB | Load litmus | Load ELF | Model | Next | Back | Restart | Search | Execution | Interface | Graph | Link to this state | Help

State

100%

Storage subsystem state (TSO):

```
0           1
|           |  (1:1:0):W 0x1000 (y)/8=1
|           |
|           |
------------
      |
      |
      |
```

Memory = [(1000:0:0):W 0x1000 (y)/8=0, (0:1:0):W 0x1100 (x)/8=1, (1000:1:0):W 0x1100 (x)/8=0]
Lock = unlocked
3 propagate write to memory: (1:1:0):W 0x1000 (y)/8=1

Thread 0 state:

```
0:1    0x050000 fetched movq 1, (%rax)  mem writes: (0:1:0):W 0x1100 (x)/8=1  reg reads:
RAX=0x_63'0000000000001100 (x) from initialstate
  micro_op_state: MOS_plain
0:2    0x050004 fetched movq (%rbx), %rcx  mem reads: (0:2:0):R from (1000:0:0):W 0x1000 (y)/8=0  reg
reads: RBX=0x_63'0000000000001000 (y) from initialstate  reg writes: RCX=0x_63'0000000000000000
  micro_op_state: MOS_plain
```

Thread 1 state:

```
1:1    0x051000 fetched movq 1, (%rax)  mem writes: (1:1:0):W 0x1000 (y)/8=1  reg reads:
RAX=0x_63'0000000000001000 (y) from initialstate
  micro_op_state: MOS_plain
1:2    0x051004 fetched movq (%rbx), %rcx  mem reads: (1:2:0):R from (1000:1:0):W 0x1100 (x)/8=0  reg
reads: RBX=0x_63'0000000000001100 (x) from initialstate  reg writes: RCX=0x_63'0000000000000000
  micro_op_state: MOS_plain
```

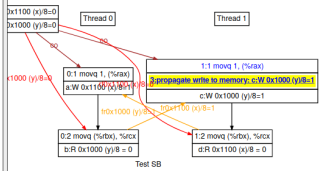
```
Choices so far (5): 0;1;0;1;2
1 enabled transitions
No disabled transitions
```

Console

```
Step 5 (4/4 finished, 72 trns) Choose [2]: 2
```

Help | Open in new tab | 100%

Graph | Refresh | Download .dot | centre | 100% | ‹› | ⇕



Sources | Download | Edit | 100% | ‹› | ⇕

X86 SB

```
1  X86 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5  uint64_t x=0;
6  uint64_t y=0;
7  0:rax=x; 0:rbx=y; 1:rax=y; 1:rbx=x
8  }
9   P0          | P1          ;
10 mov $1,(%rax)  | mov $1,(%rax) ;
11 mov (%rbx),%rcx | mov (%rbx),%rcx ;
12 exists
13 (0:rcx=0 /\ 1:rcx=0)
```

State

100%

Storage subsystem state (TSO):

```
0           1
|           |
|           |
|           |
------------
      |
      |
      |
```

Memory = [(1:1:0):W 0x1000 (y)/8=1, (0:1:0):W 0x1100 (x)/8=1, (1000:0:0):W 0x1000 (y)/8=0]
Lock = unlocked

Thread 0 state:

```
0:1    0x050000 fetched movq 1, (%rax)  mem writes: (0:1:0):W 0x1100 (x)/8=1  reg reads:
RAX=0x_63'0000000000001100 (x) from initialstate
  micro_op_state: MOS_plain
0:2    0x050004 fetched movq (%rbx), %rcx  mem reads: (0:2:0):R from (1000:0:0):W 0x1000 (y)/8=0  reg
reads: RBX=0x_63'0000000000001000 (y) from initialstate  reg writes: RCX=0x_63'0000000000000000
  micro_op_state: MOS_plain
```

Thread 1 state:

```
1:1    0x051000 fetched movq 1, (%rax)  mem writes: (1:1:0):W 0x1000 (y)/8=1  reg reads:
RAX=0x_63'0000000000001000 (y) from initialstate
  micro_op_state: MOS_plain
1:2    0x051004 fetched movq (%rbx), %rcx  mem reads: (1:2:0):R from (1000:1:0):W 0x1100 (x)/8=0  reg
reads: RBX=0x_63'0000000000001100 (x) from initialstate  reg writes: RCX=0x_63'0000000000000000
  micro_op_state: MOS_plain
```

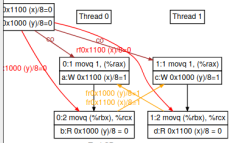
```
Choices so far (6): 0;1;0;1;2;3
No enabled transitions
No disabled transitions
```

Console

```
Step 6 (4/4 finished, /3 trns) Choose [3]: 3
```

Help | Open in new tab | 100%

Graph | Refresh | Download .dot | centre | 100%



Sources | Download | Edit | 100%

X86 SB

```
1  X86 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5  uint64_t x=0;
6  uint64_t y=0;
7  0:rax=x; 0:rbx=y; 1:rax=y; 1:rbx=x
8  }
9   P0           | P1           ;
10 mov $1,(%rax)  | mov $1,(%rax) ;
11 mov (%rbx),%rcx | mov (%rbx),%rcx ;
12 exists
13 (0:rcx=0 /\ 1:rcx=0)
```

Making x86-TSO executable as a test oracle: the RMEM tool

```
$ rmem -eager true -model tso SB.litmus
```

This provides a command-line version of the same gdb-like interface for exploring the possible transitions of the operational model, showing the current state and its possible transitions

```
help                          list commands
set always_print true         print the current state after every command
set always_graph true         generate a pdf graph in out.pdf after every step
<N>                           take transition labelled <N>, and eager successors
b                             step back one transition
search exhaustive             exhaustive search from the current state
[...]
```

```
--------------------------------------------------------------------------------
Storage subsystem state (TSO):
0           1
|           |
|           |
|           |
-------------
       |
       |
       |

  Memory = [(1000:0:0):W 0x0000000000001000 (y)/8=0, (1000:1:0):W 0x0000000000001100 (x)/8=0]
  Lock = unlocked


Thread 0 state:

ioid: 0:1     loc: 0x0000000000050000  fetched movq $1, (x)   instruction kind: write
  regs_in: {}  regs_out: {}  ioids_feeding_address: {}  nias: {succ} dia: none
  mem writes_read_from: []  committed store: true  propagated_writes: []  potential_write_addresses: []  atomic store success: not set  committed_barriers: []
  reg_reads: []
  reg_writes: []  finished: false  micro_op_state: MOS_pending_mem_write
  (0:1:0):W 0x0000000000001100 (x)/8=1
*** 0     0:1    propagate memory write to storage: (0:1:0):W 0x0000000000001100 (x)/8=1 ***

Thread 1 state:

ioid: 1:1     loc: 0x0000000000051000  fetched movq $1, (y)   instruction kind: write
  regs_in: {}  regs_out: {}  ioids_feeding_address: {}  nias: {succ} dia: none
  mem writes_read_from: []  committed store: true  propagated_writes: []  potential_write_addresses: []  atomic store success: not set  committed_barriers: []
  reg_reads: []
  reg_writes: []  finished: false  micro_op_state: MOS_pending_mem_write
  (1:1:0):W 0x0000000000001000 (y)/8=1
    1     1:1    propagate memory write to storage: (1:1:0):W 0x0000000000001000 (y)/8=1



--------------------------------------------------------------------------------
Choices so far (0):
Enabled transitions:
*** 0     0:1    propagate memory write to storage: (0:1:0):W 0x0000000000001100 (x)/8=1 ***
    1     1:1    propagate memory write to storage: (1:1:0):W 0x0000000000001000 (y)/8=1
No disabled transitions
Step 1 (0/2 finished, 0 trns) Choose [0]:
```

Making x86-TSO executable as a test oracle: the RMEM tool

And non-interactive exhaustive search:

```
$ rmem -interactive false -eager true -model tso SB.litmus
Test SB Allowed
Memory-writes=
States 4
2     *>0:RAX=0; 1:RAX=0;  via "0;0;1;0;2;1"
2     :>0:RAX=0; 1:RAX=1;  via "0;0;1;2;0;1"
2     :>0:RAX=1; 1:RAX=0;  via "0;1;1;2;3;0"
2     :>0:RAX=1; 1:RAX=1;  via "0;1;2;1;3;0"
Unhandled exceptions 0
Ok
Condition exists (0:RAX=0 /\ 1:RAX=0)
Hash=90079b984f817530bfea20c1d9c55431
Observation SB Sometimes 1 3
Runtime: 0.171546 sec
```

One can then step through a selected trace interactively using `-follow "0;0;1;0;2;1"`

x86-TSO spinlock example and TRF

Consider language-level mutexes

Statements $s ::= \ldots \mid \texttt{lock}\, x \mid \texttt{unlock}\, x$

Say lock free if it holds 0, taken otherwise.

For simplicity, don't mix locations used as locks and other locations.

Semantics (outline): $\texttt{lock}\, x$ has to *atomically* (a) check the mutex is currently free, (b) change its state to taken, and (c) let the thread proceed.
$\texttt{unlock}\, x$ has to change its state to free.

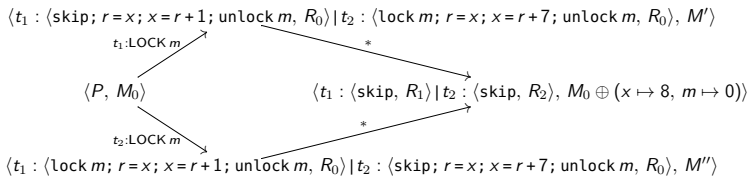
Record of which thread is holding a locked lock? Re-entrancy?

Using a Mutex

Consider

$$\begin{array}{rcl} P & = & t_1 : \langle \texttt{lock}\, m;\ r = x;\ x = r + 1;\ \texttt{unlock}\, m,\ R_0 \rangle \\ & \mid & t_2 : \langle \texttt{lock}\, m;\ r = x;\ x = r + 7;\ \texttt{unlock}\, m,\ R_0 \rangle \end{array}$$

in the initial store M_0:



where $M' = M_0 \oplus (m \mapsto 1)$

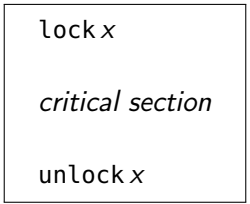
Deadlock

`lock` m can block (that's the point). Hence, you can *deadlock*.

$$
\begin{aligned}
P = \quad & t_1 : \langle \texttt{lock}\, m_1 \texttt{;}\ \texttt{lock}\, m_2 \texttt{;}\ x = 1 \texttt{;}\ \texttt{unlock}\, m_1 \texttt{;}\ \texttt{unlock}\, m_2,\ R_0 \rangle \\
\mid \quad & t_2 : \langle \texttt{lock}\, m_2 \texttt{;}\ \texttt{lock}\, m_1 \texttt{;}\ x = 2 \texttt{;}\ \texttt{unlock}\, m_1 \texttt{;}\ \texttt{unlock}\, m_2,\ R_0 \rangle
\end{aligned}
$$

Implementing mutexes with simple x86 spinlocks

Implementing the language-level mutex with x86-level simple spinlocks



Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    skip
}

critical section

unlock(x)
```

Invariant:
lock taken if $x \leq 0$
lock free if x=1
(NB: different internal representation from high-level semantics)

Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip }
}

critical section

unlock(x)
```

Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip }
}

critical section

x ←1      OR      atomic_write(x, 1)
```

Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip }
}

critical section

x ←1
```

Simple x86 Spinlock

The address of x is stored in register eax.

```
acquire:  LOCK DEC [eax]
          JNS enter
spin:     CMP [eax],0
          JLE spin
          JMP acquire
enter:

          critical section

release:  MOV [eax]←1
```

From Linux v2.6.24.7

NB: don't confuse levels — we're using x86 atomic (LOCK'd) instructions in a Linux spinlock implementation.

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	
x = 1		read x

Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	
x = 1		read x
x = 0		acquire

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x

Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x
x = 0		acquire

Spinlock SC Data Race

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x

[TODO:]

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

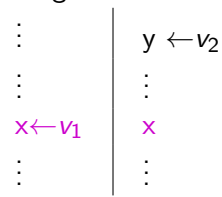
Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x
x = 0		acquire

Triangular Races

Owens [6, ECOOP 2010]

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

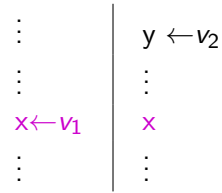


Triangular Races

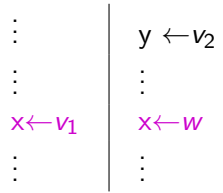
Owens [6, ECOOP 2010]

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



Not triangular race



Triangular Races

Owens [6, ECOOP 2010]

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	y $\leftarrow v_2$
⋮	⋮
x$\leftarrow v_1$	x
⋮	⋮

Not triangular race

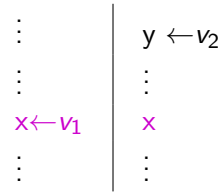
⋮	y $\leftarrow v_2$
⋮	**mfence**
x$\leftarrow v_1$	x
⋮	⋮

Triangular Races

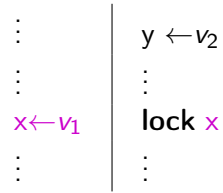
Owens [6, ECOOP 2010]

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



Not triangular race

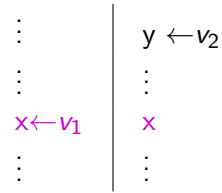


Triangular Races

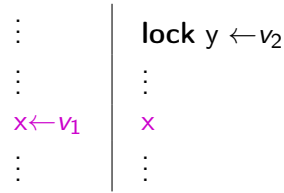
Owens [6, ECOOP 2010]

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race



Not triangular race



Triangular Races

Owens [6, ECOOP 2010]

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	y $\leftarrow v_2$
⋮	⋮
x$\leftarrow v_1$	x
⋮	⋮

Triangular race

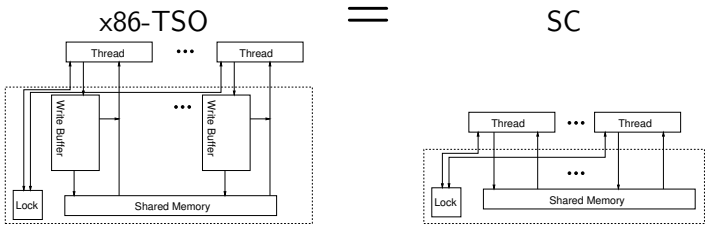
⋮	y $\leftarrow v_2$
⋮	⋮
lock x$\leftarrow v_1$	x
⋮	⋮

TRF Principle for x86-TSO

Say a program is *triangular race free (TRF)* if no SC execution has a triangular race.

Theorem 1 (TRF). If a program is TRF then any x86-TSO execution is equivalent to some SC execution.

If a program has no triangular races when run on a sequentially consistent memory, then



Spinlock Data Race

```
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip } }
critical section
x ←1
```

x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

- acquire's writes are locked

Program Correctness

Theorem 2. Any well-synchronized program that uses the spinlock correctly is TRF.

Theorem 3. Spinlock-enforced critical sections provide mutual exclusion.

Axiomatic models

Coherence

Conventional hardware architectures guarantee *coherence*:

- in any execution, for each location, there is a total order over all the writes to that location, and for each thread the order is consistent with the thread's program-order for its reads and writes to that location; or (equivalently)
- in any execution, for each location, the execution restricted to just the reads and writes to that location is SC.

Without this, you wouldn't even have correct sequential semantics, e.g. if different threads act on disjoint locations within a cache line.

In simple hardware implementations, the coherence order is that in which the processors gain write access to the cache line.

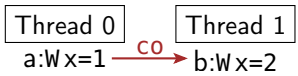
Coherence

We'll include the coherence order in the data of a candidate execution, e.g.

1+1W x86

Initial state: x=0;	
Thread 0	Thread 1
movq $1, (x) //a	movq $2, (x) //b
Final: x=2;	

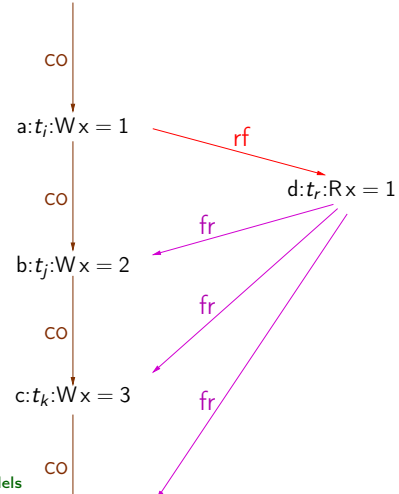
Observation: 0/0



For tests with at most two writes to each location, with values distinct from each other and from the initial state, the coherence order of a candidate execution is determined by the final state. Otherwise one might have to add "observer" threads to the test.

From-reads

Given coherence, there is a sense in which a read event is "before" the coherence-successors of the write it reads from, in the *from-reads* relation [36, 3]: $w \xrightarrow{\text{fr}} r$ iff r reads from a coherence-predecessor of w.

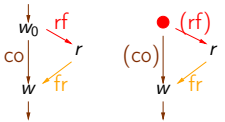


From-reads

Given coherence, there is a sense in which a read event is "before" the coherence-successors of the write it reads from, in the *from-reads* relation [36, 3]: $w \xrightarrow{\text{fr}} r$ iff r reads from a coherence-predecessor of w.

Given a candidate execution with a coherence order $\xrightarrow{\text{co}}$ (an irreflexive transitive relation over same-address writes), and a reads-from relation $\xrightarrow{\text{rf}}$ from writes to reads, define the *from-reads* relation $\xrightarrow{\text{fr}}$ to relate each read to all $\xrightarrow{\text{co}}$-successors of the write it reads from (or to all writes to its address if it reads from the initial state).

$$r \xrightarrow{\text{fr}} w \quad \text{iff} \quad (\exists w_0.\ w_0 \xrightarrow{\text{co}} w \ \wedge \ w_0 \xrightarrow{\text{rf}} r) \ \vee \ (\text{iswrite}(w) \wedge \text{addr}(w) = \text{addr}(r) \wedge \neg\exists w_0.\ w_0 \xrightarrow{\text{rf}} r)$$



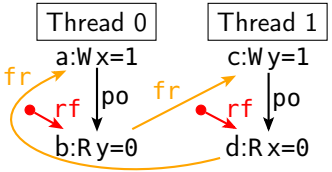
From-reads

Given coherence, there is a sense in which a read event is "before" the coherence-successors of the write it reads from, in the *from-reads* relation [36, 3]: $w \xrightarrow{\text{fr}} r$ iff r reads from a coherence-predecessor of w.

Lemma 1. In any well-formed candidate execution:

- For any distinct same-address writes w and w', either $w \xrightarrow{\text{co}} w'$ or $w' \xrightarrow{\text{co}} w$.
- For any same-address read r and write w, either $w(\xrightarrow{\text{co}} \cup \xrightarrow{\text{id}}) \xrightarrow{\text{rf}} r$, or $r \xrightarrow{\text{fr}} w$.
- For any same-address reads r and r', either they both read from the same write (or both from the initial state), or $r \left(\xrightarrow{\text{fr}}\xrightarrow{\text{rf}}\right) r'$, or $r' \left(\xrightarrow{\text{fr}}\xrightarrow{\text{rf}}\right) r$.

The SB cycle

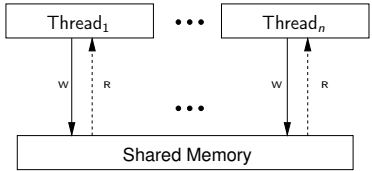


In this candidate execution the reads read from the initial state, which is coherence-before all writes, so there are fr edges from the reads to all the writes at the same address.

This suggests a more abstract characterisation of why this execution is non-SC, and hence a different "axiomatic" style of defining relaxed models:

If we regard the reads as in their $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{fr}}$ places in the per-location coherence orders, those are not consistent with the per-thread program orders.

SC again, operationally



Define an *SC abstract machine memory* $m \xrightarrow{e} m'$
(forgetting `MFENCE` and LOCK'd instructions for now)

Take each thread as executing in-order (again)

Events e ::= $a{:}t{:}\mathrm{W}\,x{=}v$ a write of value v to address x by thread t, ID a
| $a{:}t{:}\mathrm{R}\,x{=}v$ a read of v from x by t, ID a

States m are just memory states:

$$m : addr \to value$$

RM: Read from memory

$$\frac{m(x) = v}{m \xrightarrow{a:t:\mathrm{R}\,x=v} m}$$

WM: Write to memory

$$\frac{}{m \xrightarrow{a:t:\mathrm{W}\,x=v} m \oplus (x \mapsto v)}$$

SC again, operationally

See how this captures the essence of SC:

reads read from the most recent write to the same address, in some program-order-respecting interleaving of the threads.

SC again, operationally

Say a *trace* T is a list of events $[e_1, \ldots, e_n]$ that have unique IDs
$\forall i, j \in 1..n.\ i \neq j \implies \mathrm{id}(e_i) \neq \mathrm{id}(e_j)$

Write:

- $e < e'$ iff e is before e' in the trace $\quad e < e' \Leftrightarrow \exists i, j.\ e = e_i \wedge e' = e_j \wedge i < j$

Say the traces of the SC abstract machine memory are all traces $T = [e_1, \ldots, e_n]$ with unique IDs such that

$$m_{\mathrm{init}} \xrightarrow{e_1} m_1 \ldots \xrightarrow{e_n} m_n$$

for the initial memory state $m_{\mathrm{init}} = \lambda x : addr.\, 0$ and some $m_1, \ldots, m_n$

SC, axiomatically

Now we try to capture the same set of behaviours as a property of candidate executions

Candidate Executions, more precisely

Say a *candidate execution* consists of a *candidate pre-execution* $\langle E, \xrightarrow{\text{po}} \rangle$, where:

- E is a finite set of events, with unique IDs, ranged over by e etc.
- *program order* (po) is an irreflexive transitive relation over E, that only relates pairs of events from the same thread (In general this might not be an irreflexive total order for the events of each thread separately, but we assume that too fo

and a *candidate execution witness* $X = \langle \xrightarrow{\text{rf}}, \xrightarrow{\text{co}} \rangle$, consisting of:

- *reads-from* (rf), a binary relation over E, that only relates write/read pairs with the same address and value, with at most one write per read, and other reads reading from the initial state
  (note that this is intensional: it identifies *which write*, not just the value)

- *coherence* (co), an irreflexive transitive binary relation over E, that only relates write/write pairs with the same address, and that is an irreflexive total order when restricted to the writes of each address separately

Candidate Executions, more precisely

Say a *candidate execution* consists of a *candidate pre-execution* $\langle E, \xrightarrow{\text{po}} \rangle$, where:

- E is a finite set of events, with unique IDs, ranged over by e etc. $\forall e, e'.\ e \neq e' \implies \mathrm{id}(e) \neq \mathrm{id}(e')$
- *program order* (*po*) is an irreflexive transitive relation over E, that only relates pairs of events from the same thread (In general this might not be an irreflexive total order for the events of each thread separately, but we assume that too fo

$\forall e.\ \neg(e \xrightarrow{\text{po}} e) \qquad \forall e, e'.\ (\mathrm{thread}(e) = \mathrm{thread}(e') \wedge e \neq e') \implies e \xrightarrow{\text{po}} e' \vee e' \xrightarrow{\text{po}} e$
$\forall e, e', e''.\ (e \xrightarrow{\text{po}} e' \wedge e' \xrightarrow{\text{po}} e'') \implies e \xrightarrow{\text{po}} e''$
$\forall e, e'.\ e \xrightarrow{\text{po}} e' \implies \mathrm{thread}(e) = \mathrm{thread}(e')$

and a *candidate execution witness* $X = \langle \xrightarrow{\text{rf}}, \xrightarrow{\text{co}} \rangle$, consisting of:

- *reads-from* (*rf*), a binary relation over E, that only relates write/read pairs with the same address and value, with at most one write per read, and other reads reading from the initial state
  (note that this is intensional: it identifies *which write*, not just the value)

$\forall e, e', e''.\ (e \xrightarrow{\text{rf}} e'' \wedge e' \xrightarrow{\text{rf}} e'') \implies e = e'$
$\forall e, e'.\ e \xrightarrow{\text{rf}} e' \implies \mathrm{iswrite}(e) \wedge \mathrm{isread}(e') \wedge \mathrm{addr}(e) = \mathrm{addr}(e') \wedge \mathrm{value}(e) = \mathrm{value}(e')$
$\forall e.\ (\mathrm{isread}(e) \wedge \neg\exists e'.e' \xrightarrow{\text{rf}} e) \implies \mathrm{value}(e) = m_{\mathrm{init}}(\mathrm{addr}(e))$

- *coherence* (*co*), an irreflexive transitive binary relation over E, that only relates write/write pairs with the same address, and that is an irreflexive total order when restricted to the writes of each address separately

$\forall e.\ \neg(e \xrightarrow{\text{co}} e)$
$\forall e, e', e''.\ (e \xrightarrow{\text{co}} e' \wedge e' \xrightarrow{\text{co}} e'') \implies e \xrightarrow{\text{co}} e''$
$\forall e, e'.\ e \xrightarrow{\text{co}} e' \implies \mathrm{iswrite}(e) \wedge \mathrm{iswrite}(e') \wedge \mathrm{addr}(e) = \mathrm{addr}(e')$
$\forall a.\ \forall e, e'.\ (e \neq e' \wedge \mathrm{iswrite}(e) \wedge \mathrm{iswrite}(e') \wedge \mathrm{addr}(e) = a \wedge \mathrm{addr}(e') = a) \implies e \xrightarrow{\text{co}} e' \vee e' \xrightarrow{\text{co}} e$

SC, axiomatically

Say a trace $T = [e_1, \ldots, e_n]$ and a candidate pre-execution $\langle E, \xrightarrow{\text{po}} \rangle$ have the same thread-local behaviour if

- they have the same events $E = \{e_1, \ldots, e_n\}$
- they have the same program-order relations, i.e. $\xrightarrow{\text{po}} = \{(e, e') \mid e < e' \wedge \text{thread}(e) = \text{thread}(e')\}$

Then:

Theorem 4. If T and $\langle E, \xrightarrow{\text{po}} \rangle$ have the same thread-local behaviour, then the following are equivalent:

1. T is a trace of the SC abstract-machine memory
2. there exists an execution witness $X = \langle \xrightarrow{\text{rf}}, \xrightarrow{\text{co}} \rangle$ for $\langle E, \xrightarrow{\text{po}} \rangle$ such that $\text{acyclic}(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$.

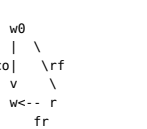
Proof. For left-to-right, given the trace order $<$, construct an execution witness:

$$
\begin{aligned}
e \xrightarrow{\mathsf{rf}} e' \quad &\Leftrightarrow \quad \mathrm{iswrite}(e) \wedge \mathrm{isread}(e') \wedge \mathrm{addr}(e) = \mathrm{addr}(e') \wedge e < e' \wedge \\
&\qquad \forall e''.\ (e < e'' \wedge e'' < e') \implies \neg(\mathrm{iswrite}(e'') \wedge \mathrm{addr}(e'') = \mathrm{addr}(e')) \\
e \xrightarrow{\mathsf{co}} e' \quad &\Leftrightarrow \quad \mathrm{iswrite}(e) \wedge \mathrm{iswrite}(e') \wedge \mathrm{addr}(e) = \mathrm{addr}(e') \wedge e < e'
\end{aligned}
$$

```
Now check the properties

Checking po properties: ...all follow from "have the same program-order relations"
Checking rf properties:
forall e,e',e''. (e rf e'' & e' rf e'') => e=e'
...Suppose wlog e<e' then that contradicts the no-intervening-write clause of the construction
forall e,e'. e rf e' => iswrite e & isread e' & addr e=addr e'
...by construction of rf
forall e,e'. e rf e' => value e = value e'
...because there are no intervening writes to the same address between e and e', m(addr e) remains constant (by induction on that part of the
   execution trace), and hence is read at e'
forall e (isread e & not exists e'. e' rf e) => value(e)=m0(addr(e))
...from the construction of rf, if there isn't an rf edge then there isn't a write to that address preceding in the trace (if there were
   one, there would be a <-maximal one), so by induction along that part of the trace the value in m for this address is unchanged from m0.
Checking co properties:
forall e. not (e co e)
...if e co e then e<e but that contradicts the definition of <
forall e,e',e'' (e co e' & e' co e'') => e co e''
...equivalence of iswrite and same-addr, and transitivity of <
forall e, e'. e co e' => iswrite e & iswrite e' & addr e = addr e'
...by construction of co
forall a. forall e,e'. (e<>e' & iswrite e & iswrite e' & addr e = a & addr e' = e) => e co e' || e' co e
...if e<>e' then either e<e' or e'<e; then in either case construct a co
```

Now check each of po, rf, co, and rf go forwards in the trace. This is just about the construction; it doesn't involve the machine.

po, rf, co: by construction
fr: suppose r fr w
case 1) for some w0, w0 co w & w0 rf r



If r < w we are done, so suppose for a contradiction that w < r.

By the definitions of co and rf, w0 is a write, w0 and w and r have the same address, w0 < w, and w0 < r. But then w0 < w < r, contradicting the no-intervening-write clause of the definition of rf

case 2) iswrite w & addr w = addr r & not exists w0. w0 rf r
Suppose for a contradiction that w < r.
Then there is at least one write (namely w) with the same address as r before it in <.
Take the last such write, w', then by the definition of rf, w' rf r.

Finally, as we have po, rf, co, and fr all embedded in <, which by definition is acyclic, their union must be acyclic.

For the right-to-left direction, given an execution witness $E = \langle \xrightarrow{\mathsf{rf}}, \xrightarrow{\mathsf{co}} \rangle$ such that $\mathrm{acyclic}(\xrightarrow{\mathsf{ob}})$, where $\xrightarrow{\mathsf{ob}} = (\xrightarrow{\mathsf{po}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})$, construct a trace $[e_1, .., e_n]$ as an arbitrary linearisation of $\xrightarrow{\mathsf{ob}}$.

```
By acyclic(ob), we know if ei ob ej then i<j  (but not the converse).

Construct memory states mi inductively along that trace, starting with m0,
  mutating the memory for each write event, and leaving it unchanged for each read.

To check that actually is a trace of the SC abstract machine memory, i.e. that m0 --e1--> m1 ... --en--> mn, it remains to check
  for each read, say rj at index j, that m_{j-1}(addr(rj)) = value(rj)

By the construction of the mi,

  m_{j-1}(addr(rj)) = value(ei)     where i is the largest i<j such that iswrite ei & addr ei=addr rj, if there is one
                    or m0(addr(rj))  otherwise

In the first case, write wi for ei. We know by the fr lemma that either wi co* rf rj  or rj fr wi.

   Case the latter (rj fr wi):  then rj ob wi so j<i, contradicting i<j.
   Case the former (wi co* wk rf rj for some k):
      We know i <= k < j, so unless i=k we contradict the "largest"
      So wi rf rj, so they have the same value

In the second case, there is no i<j such that iswrite ei & addr ei=addr rj
So there is no w ob rj such that addr w = addr rj
So there is no w rf rj
So by the candidate-execution initial-state condition, value(rj)=m0(addr(rj))
```

SC, axiomatically

This lets us take the predicate $\mathrm{acyclic}(\xrightarrow{\mathsf{po}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})$ as an equivalent characterisation of sequential consistency.

The executions of the SC axiomatic model are all candidate executions, i.e. all pairs of

- a candidate pre-execution $\langle E, \xrightarrow{\mathsf{po}} \rangle$, and
- a candidate execution witness $X = \langle \xrightarrow{\mathsf{rf}}, \xrightarrow{\mathsf{co}} \rangle$ for it,

that satisfy the condition $\mathrm{acyclic}(\xrightarrow{\mathsf{po}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})$.

Note that we've not yet constrained either the operational or axiomatic model to the correct *thread-local* semantics for any particular machine language – we'll come back to that. So far, this is just the memory behaviour.

SC, axiomatically

This characterisation suggests a good approach to *test generation*: construct interesting non-SC tests from non-SC cycles of relations – the idea of the `diy7` tool [27, Alglave, Maranget]. More later.

It also gives different ways of making the model *executable as a test oracle*:

- enumerating all conceivable candidate executions and checking the predicate, as in the `herd7` tool [27], and
- translating the predicate into SMT constraints, as the `isla-axiomatic` [29, Armstrong et al.] tool does.

More on these later too.

Note how the construction of an arbitrary linearisation of $\xrightarrow{\mathsf{ob}}$ illustrates some "irrelevant" interleaving in the SC operational model.

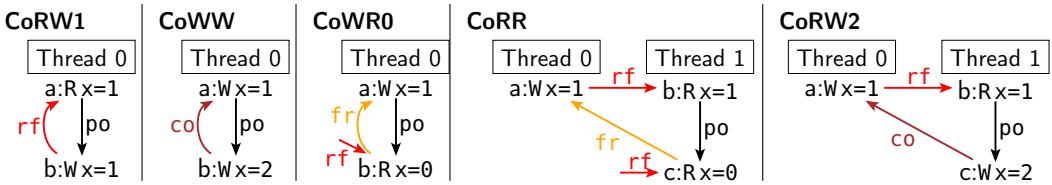
Expressing coherence axiomatically, on candidate executions

```
let pos = po & loc            (* same-address part of po, aka po-loc *)

acyclic pos | rf | co | fr    (* coherence check *)
```

Coherence is equivalent to per-location SC. Note that $\xrightarrow{\text{pos}}$, $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{co}}$, and $\xrightarrow{\text{fr}}$ only relate pairs of events with the same address, so this checks SC-like acyclicity for each address separately.

We already proved that any SC machine execution satisfies this, because $\xrightarrow{\text{pos}} \subseteq \xrightarrow{\text{po}}$

Basic coherence shapes

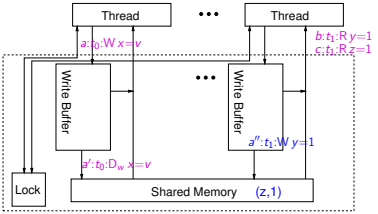


Theorem 5. If a candidate execution has a cycle in `pos | co | rf | fr`, it contains one of the above shapes (where the reads shown as from the initial state could be from any coherence predecessor of the writes) [23, 15, Alglave].

How does the SC machine prevent each of these?

x86-TSO axiomatic model

Axiomatic model style: single vs multi-event per access



In the x86-TSO operational model (unlike SC):

- each store has two events, $w = (a{:}t_0{:}\mathrm{W}\ x{=}v)$ and $a'{:}t_0{:}\mathrm{D}_w\ x{=}v$
- each load has one event, but it can arise in two ways

but that is not explicit in the candidate executions we've used.

We could conceivably:

1. add some or all of that data to candidate executions, and give an axiomatic characterisation of the abstract-machine execution, or
2. stick with one-event-per-access candidate executions, expressing the conditions that define allowed behaviour just on those

Perhaps surprisingly, 2 turns out to be possible

Two x86-TSO axiomatic models

1. one in TPHOLs09 [4, Owens, Sarkar, Sewell], in SparcV8 style
2. one simplified from a current `cat` model, in the "herd" style of [15, Alglave et al.]
   `https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso-mixed.cat`

Both proved equivalent to the operational model and tested against hardware
(on small and large test suites for the two models respectively)

forget LOCK'd instructions and MFENCEs for a bit

Notation

Axiomatic models define predicates on candidate execution using various binary relations over events

Binary relations are just sets of pairs.

We write

- $(e, e') \in \mathsf{r}$
- $e \xrightarrow{\mathsf{r}} e'$
- $e \ \mathsf{r} \ e'$

interchangeably.

Notation: relational algebra

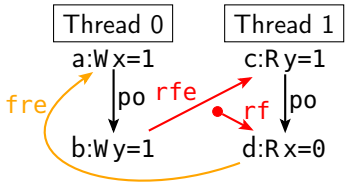
As models become more complex, it's convenient to use relational algebra instead of pointwise definitions, as in the "cat" language of `herd7` (and also `isla-axiomatic`):

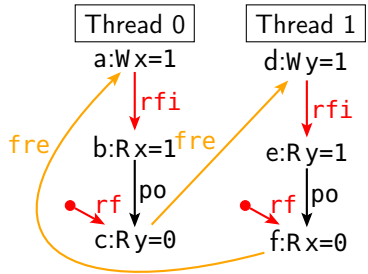
`r | s`	the union of `r` and `s`	$\{(e, e') \mid e \ \mathtt{r}\ e' \vee e \ \mathtt{s}\ e'\}$
`r & s`	the intersection of `r` and `s`	$\{(e, e') \mid e \ \mathtt{r}\ e' \wedge e \ \mathtt{s}\ e'\}$
`r ; s`	the composition of `r` and `s`	$\{(e, e'') \mid \exists e'.\ e \ \mathtt{r}\ e' \ \mathtt{s}\ e''\}$
`r \ s`	`r` minus `s`	$\{(e, e') \mid e \ \mathtt{r}\ e' \wedge \neg(e \ \mathtt{s}\ e')\}$
`[S]`	the identity on some set `S` of events	$\{(e, e) \mid e \in \mathtt{S}\}$
`S*S'`	the product of sets `S` and `S'`	$\{(e, e') \mid e \in \mathtt{S} \wedge e' \in \mathtt{S'}\}$
`loc`	same-location, events at the same address	$\{(e, e') \mid \mathrm{addr}(e) = \mathrm{addr}(e')\}$
`int`	internal, events of the same thread	$\{(e, e') \mid \mathrm{thread}(e) = \mathrm{thread}(e')\}$
`ext`	external, events of different thread	$\{(e, e') \mid \mathrm{thread}(e) \neq \mathrm{thread}(e')\}$

`R`, `W`, `MFENCE`: the sets of all read, write, and mfence events $\{e \mid \mathrm{isread}(e)\}$, etc.

Internal vs external relations

In TSO, and in the more relaxed Armv8-A, IBM Power, and RISC-V that we come to later, the same-thread and different-thread parts of `rf`, `co`, and `fr` behave quite differently.





Write `rfe` and `rfi` for the *external* (different-thread) and *internal* (same-thread) parts of `rf`, and similarly `coe`, `coi`, and `fre`, `fri`.

$$\texttt{rfe} \;=\; \texttt{rf\&ext} \;=\; \{(e,e') \mid e \;\texttt{rf}\; e' \wedge \mathrm{thread}(e) \neq \mathrm{thread}(e')\}$$
$$\texttt{rfi} \;=\; \texttt{rf\&int} \;=\; \{(e,e') \mid e \;\texttt{rf}\; e' \wedge \mathrm{thread}(e) = \mathrm{thread}(e')\}$$

Internal vs external relations for x86-TSO

In the abstract machine (ignoring LOCK'd instructions), threads interact only via the common memory

Any external (inter-thread) reads-from, coherence, or from-reads edge is, in operational terms, about write dequeue events:

- if w `rfe` e in the machine, then w must have been dequeued before e reads from it
- if w `coe` w' in the machine, then w must have been dequeued before w' is dequeued
- if r `fre` w in the machine, then r reads before w is dequeued

Does the x86-TSO abstract machine maintain coherence? How?

The coherence order over writes is determined by the order that they reach memory: the trace order of $a{:}t{:}\mathrm{D}_w\, x{=}v$ dequeue events (might not match the enqueue order)

Does the x86-TSO abstract machine maintain coherence? How?

The coherence order over writes is determined by the order that they reach memory: the trace order of $a{:}t{:}\mathrm{D}_w\, x{=}v$ dequeue events (might not match the enqueue order)

Read events that read from memory are in the right place in the trace w.r.t. that (after the dequeue of their `rf`-predecessor and before the dequeues of their `fr`-successors)

Does the x86-TSO abstract machine maintain coherence? How?

The coherence order over writes is determined by the order that they reach memory: the trace order of $a{:}t{:}\mathrm{D}_w\, x{=}v$ dequeue events (might not match the enqueue order)

Read events that read from memory are in the right place in the trace w.r.t. that (after the dequeue of their `rf`-predecessor and before the dequeues of their `fr`-successors)

But read events that read from buffers will be *before* the corresponding dequeue event in the trace

- they will be after the $a{:}t{:}\mathrm{W}\, x{=}v$ enqueue event they read from, and before any po-later enqueue event
- the ordering among same-thread write enqueues ends up included in the coherence order by the FIFO nature of the buffer: two po-related writes are dequeued in the same order

Does the x86-TSO abstract machine maintain coherence? How?

The coherence order over writes is determined by the order that they reach memory: the trace order of $a{:}t{:}\mathrm{D}_w\, x{=}v$ dequeue events (might not match the enqueue order)

Read events that read from memory are in the right place in the trace w.r.t. that (after the dequeue of their `rf`-predecessor and before the dequeues of their `fr`-successors)

But read events that read from buffers will be *before* the corresponding dequeue event in the trace

- they will be after the $a{:}t{:}\mathrm{W}\, x{=}v$ enqueue event they read from, and before any po-later enqueue event
- the ordering among same-thread write enqueues ends up included in the coherence order by the FIFO nature of the buffer: two po-related writes are dequeued in the same order

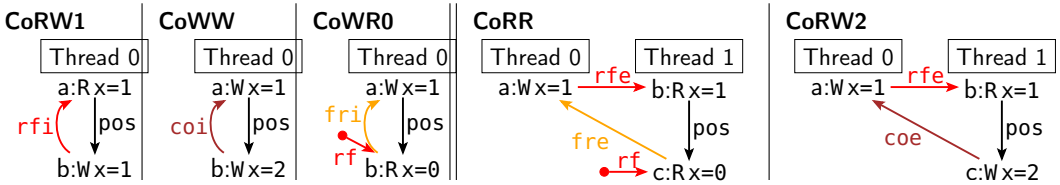
For reading from memory, if there's a write to this address in the local buffer, it will end up coherence-after all writes that have already reached memory, so it would be a coherence violation to read from memory – hence the buffer-empty condition in RM

Back to coherence, axiomatically

Recall we expressed coherence axiomatically as:

```
acyclic pos | rf | co | fr   (* coherence check, where pos = po & loc *)
```

Basic coherence shapes again



How does the machine prevent each of these?

CoRW1: a read can only see a same-thread write that is pos-before it (via buffer or via memory)
CoWW: the buffers are FIFO, so two pos writes are dequeued in pos-order
CoWR0: b reads from a coherence-predecessor $c{:}t{:}\mathrm{W}\,x{=}0$ (which could be on any thread) of a

- Case c is on the same thread as b. c must be po-before a, as writes are enqueued in po and, because the buffers are FIFO, dequeued (establishing their coherence order) in the same order.
  - Case b reads from memory, by RM. Then c must have been dequeued.
    - Case a has been dequeued before the read. Then that must have been after c was, so b would have read from a.
    - Case a is still buffered at the read. That violates the $\mathrm{no\_pending}(m.B(t), x)$ condition of RM.
  - Case b reads from buffer, by RB. Then a must still precede c in the buffer. This violates the $\mathrm{no\_pending}(b_1, x)$ condition of RB.
- Case c is on a different thread to b. Then b reads from memory, by RM
  - Case c was dequeued before a. Then b would have read from a.
  - Case c was dequeued after a. Then a must still be in the buffer, violating the $\mathrm{no\_pending}(m.B(t), x)$ condition of RM.

CoRR: The dequeue of a must be before b reads, and b reads before c does. c reads from a coherence-predecessor $d{:}t{:}\mathrm{W}\,x{=}0$ (which could be on any thread) of a, so d must be dequeued before a. But then c would have read from a.
CoRW2: The dequeue of a must be before b reads, and b reads before c is enqueued, which is before c is dequeued. Then c is coherence-before a, so c must be dequeued before a is. But this would be a cycle in machine execution time.

Locally ordered before w.r.t. external relations

Now what about thread-local ordering of events that might be to different locations, as seen by other threads?

Say a machine trace T is *complete* if it has no non-dequeued write, and for any write enqueue event w in such, write $D(w)$ for the unique corresponding dequeue event

For same-thread events in a complete machine trace:

- If w po w' then w is dequeued before w' (write $D(w) < D(w')$)
- If r po r' then r reads before r' reads
- If r po w then r reads before w is enqueued, and hence before w is dequeued
- If w po r, then w is enqueued before r reads, but the dequeue of w and the read are unordered

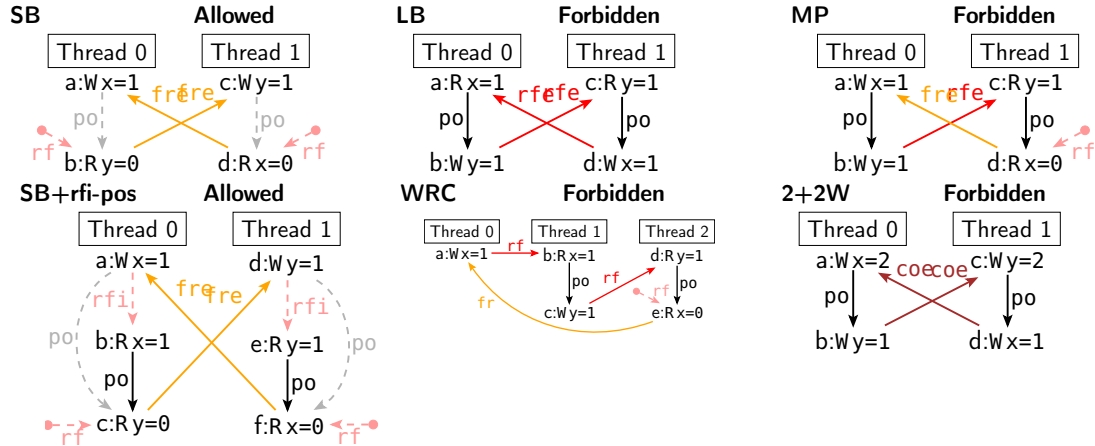
So, as far as external observations go (i.e. via `rfe`, `coe`, `fre`), `po\([W];po;[R])` is preserved.

x86-TSO axiomatic

That leads us to:

```
let pos = po & loc                (* same-address part of po (aka po-loc)*)
acyclic pos | rf | co | fr        (* coherence check *)


let obs = rfe | coe | fre         (* observed-by *)
let lob = po \ ([W];po;[R])       (* locally-ordered-before *)
let ob = obs | lob                (* ordered-before *)

 (* ob = po \ ([W];po;[R]) | rfe | coe | fre     just expanding out *)

acyclic ob                        (* 'external' check *)
```

x86-TSO axiomatic: some examples again



Coherence: `acyclic pos|rf|co|fr` ...the only pos here are the `rfi` edges

External observation: `acyclic po\([W];po;[R]) | rfe | coe | fre` ...solid edges

[TODO:use the rfi/rfe etc versions, as in the notes]

x86-TSO axiomatic: more formally

Say an *x86-TSO trace* T is a list of x86-TSO machine events $[e_1, \ldots, e_n]$ with unique IDs

Given such a trace, we write $<$ for the trace order $\quad e < e' \Leftrightarrow \exists i, j.\ e = e_i \wedge e' = e_j \wedge i < j$

Say an x86-TSO candidate pre-execution is $\langle E, \mathrm{po} \rangle$ where

- E is exactly as for SC, a set of write and read events from the x86-TSO machine event grammar, without D events
- po is a relation over E satisfying the same conditions as for SC

and a candidate execution witness is $\langle \mathtt{rf}, \mathtt{co} \rangle$ satisfying the same conditions as for SC.

Say a trace $T = [e_1, \ldots, e_n]$ and a candidate pre-execution $\langle E, \mathrm{po} \rangle$ have the same thread-local behaviour if

- they have the same thread-interface access events (no dequeue or fence events)
  $E = \{e \mid e \in \{e_1, \ldots, e_n\} \wedge (\mathrm{iswrite}(e) \vee \mathrm{isread}(e))\}$
- they have the same program-order relations over those, i.e.
  $\mathrm{po} = \{(e, e') \mid e \in E \wedge e' \in E \wedge e < e' \wedge \mathrm{thread}(e) = \mathrm{thread}(e')\}$

x86-TSO operational/axiomatic correspondence

Then:

Theorem 6. For any candidate pre-execution $\langle E, \mathtt{po} \rangle$, the following are equivalent:

1. there exists a complete trace T of the x86-TSO abstract-machine memory with the same thread-local behaviour as that candidate pre-execution
2. there exists an x86-TSO execution witness $X = \langle \mathtt{rf}, \mathtt{co} \rangle$ for $\langle E, \mathtt{po} \rangle$ such that $\mathrm{acyclic}(\mathtt{pos} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})$ and $\mathrm{acyclic}\ \mathtt{ob}$.

x86-TSO operational/axiomatic correspondence

Proof idea:

1. Given an operational execution, construct an axiomatic candidate in roughly the same way as we did for SC, mapping dequeue transitions to write events, then check the acyclicity properties.
2. Given an axiomatic execution, construct an operational trace by sequentialising ob, mapping write events onto dequeue transitions and adding write enqueue transitions as early as possible, then check the operational machine admits it.

Proof sketch: x86-TSO operational implies axiomatic

Given such a trace T, construct a candidate execution.
$E = \{e \mid e \in \{e_1, \ldots, e_n\} \wedge (\mathrm{iswrite}(e) \vee \mathrm{isread}(e)\}$

For `rf`, we recharacterise the machine behaviour in terms of the labels of the trace alone.
Say the potential writes for a read r are $\mathrm{PW}(r) = \{w \mid w \in E \wedge \mathrm{iswrite}(w) \wedge \mathrm{addr}(w) = \mathrm{addr}(r)\}$

w `rf` r	$\iff$	$\mathrm{isread}(r) \wedge w \in \mathrm{PW}(r) \wedge ($
(from-buffer, same-thread *)*		
(w in buffer *)*		$(\mathrm{thread}(w) = \mathrm{thread}(r) \wedge w < r < D(w)$
(no intervening in buffer *)*		$\wedge \neg \exists w' \in \mathrm{PW}(r).\mathrm{thread}(w') = \mathrm{thread}(r) \wedge w < w' < r)$
(from-memory, any-thread *)*		$\vee$
(w in memory *)*		$(D(w) < r$
(no intervening in buffer *)*		$\wedge \neg \exists w' \in \mathrm{PW}(r).\ \mathrm{thread}(w') = \mathrm{thread}(r) \wedge w' < r < D(w')$
(no intervening in memory *)*		$\wedge \neg \exists w' \in \mathrm{PW}(r).\ D(w) < D(w') < r))$

For `co`, say w `co` w' if $\mathrm{iswrite}(w) \wedge \mathrm{iswrite}(w') \wedge \mathrm{addr}(w) = \mathrm{addr}(w') \wedge D(w) < D(w')$

Check the candidate execution well-formedness properties hold
...the w `rf` r implies $\mathrm{value}(r) = \mathrm{value}(w)$ condition essentially checks correctness of the `rf` characterisation

For `acyclic ob`, check each (e, e') in `po\([W];po;[R]) | rfe | coe | fre` is embedded in the trace order w.r.t. read and dequeue-write points
i.e., that $\hat{D}(e) < \hat{D}(e')$, where $\hat{D}(w) = D(w)$ and $\hat{D}(r) = r$

For `acyclic pos|rf|co|fr`, construct a modified total order $<_C$, the machine coherence order augmented with reads in the coherence-correct places, and check each (e, e') is embedded in that.
$<_C$ is constructed from the trace order $<$ by:

w	$\mapsto$	[]
r	$\mapsto$	$[r]$ if r reads from memory
		[] if r reads from its thread's buffer
$a{:}t{:}\mathrm{D}_w\ x{=}v$	$\mapsto$	$[w]$ @[r | r reads from w via buffer, ordered by $<$]

Note how this preserves trace order among all D events and reads from memory (mapping the D's to W's), and reshuffles reads from buffers to correct places in coherence, preserving `pos` but not other `po`.

Proof sketch: x86-TSO axiomatic implies operational

Consider a candidate execution satisfying `acyclic(ob)` and `acyclic(pos|rf|co|fr)`

Take some arbitrary linearisation S of `ob`, and define a trace by recursion on S.

```
g [] T              = T
g ((e::S') as S) T  =

  (* eagerly enqueue all possible writes *)
  let next_writes = [ w | w IN S & w NOTIN T & w not S-after any non-write thread(w) event ]
  let T' = T @ next_writes

  match e with
  | w -> g S' (T' @ [D(w)])          (* dequeue the write when we get to its W event in S *)
  | r -> g S' (T' @ [r])             (* perform reads when we get to them *)
  | ...likewise for mfence except that we're ignoring those for now.
```

Check that that is a machine trace, using the acyclicity properties.

Mechanised proof

Mechanised formalisation and proof, in Isabelle, by Paul Durbaba (Part III, 2020–21)

x86-TSO axiomatic: adding MFENCEs and RMWs

```
include "x86fences.cat"
include "cos.cat"
let pos = po & loc                      (* same-address part of po, aka po-loc *)

(* Observed-by *)
let obs = rfe | fre | coe

(* Locally-ordered-before *)
let lob = po \ ([W]; po; [R])
        | [W]; po; [MFENCE]; po; [R] (* W/R pairs separated by an MFENCE *)
        | [W]; po; [R & X]           (* W/R pairs with at least one from an  *)
        | [W & X]; po; [R]           (*  atomic RMW, where X identifies such *)

(* Ordered-before *)
let ob = obs | lob

(* Coherence check *)
acyclic pos | rf | co | fr

(* Atomicity requirement *)
empty rmw & (fre;coe)          (* nothing between the R and W of atomic RMWs *)

(* External check *)
acyclic ob
```

Summary of axiomatic-model sets and relations

The data of a candidate pre-execution:

- a set E of events
- $\texttt{po} \subseteq E \times E$, program-order

The data of a candidate execution witness:

- $\texttt{rf} \subseteq W \times R$, reads-from
- $\texttt{co} \subseteq W \times W$, coherence

Subsets of E:

R	all read events
W	all write events
MFENCE	all mfence events
X	all locked-instruction accesses

Derived relations, generic:

`loc`	same-location, events at the same address	$\{(e, e') \mid \mathrm{addr}(e) = \mathrm{addr}(e')\}$
`ext`	external, events of different thread	$\{(e, e') \mid \mathrm{thread}(e) \neq \mathrm{thread}(e')\}$
`int`	internal, events of the same thread	$\{(e, e') \mid \mathrm{thread}(e) = \mathrm{thread}(e')\}$
`pos`	same-location `po`	`po & loc` (aka `po-loc`)
`pod`	different-location `po`	`po \ loc`
`fr`	from-reads	r `fr` w iff $(\exists w_0.\ w_0$ `co` $w \wedge w_0$ `rf` $r) \vee (\mathrm{iswrite}(w) \wedge \mathrm{addr}(w) = \mathrm{addr}(r) \wedge \neg\exists w_0.\ w_0$ `rf`
`rfe`, `coe`, `fre`	different-thread (external) parts of `rf`, `co`, `fr`	`rfe`=`rf & ext` etc.
`rfi`, `coi`, `fri`	same-thread (internal) parts of `rf`, `co`, `fr`	`rfi`=`rf & int` etc.

Derived relations, specific to x86 model:

`obs`	observed-by	`obs = rfe | coe | fre`
`lob`	locally-ordered-before	`lob = po \([W];po;[R]) | ...`
`ob`	ordered before	`ob = obs | lob`

Validating models

Validating the models?

We invented a new abstraction; we didn't just formalise an existing clear-but-non-mathematical spec. So why should we, or anyone else, believe it?

- some aspects of the vendor arch specs *are* clear (especially the examples)
- experimental comparison of model-allowed and h/w-observed behaviour on tests
  - models should be *sound* w.r.t. experimentally observable behaviour of existing h/w (modulo h/w bugs)
  - but the architectural intent may be (often is) looser
- discussion with vendor architects – does it capture their intended envelope of behaviour? Do they *a priori* know what that is in all cases?
- discussion with expert programmers – does it match their practical knowledge?
- proofs of metatheory
  - operational / axiomatic correspondence
  - implementability of C/C++11 model above x86-TSO [7, POPL 2011]
  - TRF-SC result [6, ECOOP 2010]

Re-read x86 vendor prose specifications with x86-TSO op/ax in mind

Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3 Ch.8, page 3056 (note that the initial contents page only covers Vol.1; Vol.3 starts on page 2783)
8.2.2 Memory Ordering in P6 and More Recent Processor Families The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

1. Reads are not reordered with other reads.
2. Writes are not reordered with older reads.
3. Writes to memory are not reordered with other writes [...]
4. Reads may be reordered with older writes to different locations but not with older writes to the same location.
5. Reads or writes cannot be reordered with locked instructions
6. Reads cannot pass earlier MFENCE instructions.
7. Writes cannot pass earlier MFENCE instructions.
8. MFENCE instructions cannot pass earlier reads or writes.

In a multiple-processor system, the following ordering principles apply:

1. Writes by a single processor are observed in the same order by all processors.
2. Writes from an individual processor are NOT ordered with respect to the writes from other processors.
3. Memory ordering obeys causality (memory ordering respects transitive visibility).
4. Any two stores are seen in a consistent order by processors other than those performing the stores
5. Locked instructions have a total order.

MFENCE – Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Re-read x86 vendor prose specifications with x86-TSO op/ax in mind

Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3 Ch.8, page 3056 (note that the initial contents page only covers Vol.1; Vol.3 starts on page 2783)
8.2.2 Memory Ordering in P6 and More Recent Processor Families The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

1. Reads are not reordered with other reads.x86-TSO-op: instructions are not reordered, but the buffering has a similar effect for `[W];pod;[R]`
2. Writes are not reordered with older reads.
3. Writes to memory are not reordered with other writes [...]
4. Reads may be reordered with older writes to different locations but not with older writes to the same location.
5. Reads or writes cannot be reordered with locked instructions
6. Reads cannot pass earlier MFENCE instructions.
7. Writes cannot pass earlier MFENCE instructions.
8. MFENCE instructions cannot pass earlier reads or writes.

In a multiple-processor system, the following ordering principles apply:

1. Writes by a single processor are observed in the same order by all processors.
2. Writes from an individual processor are NOT ordered with respect to the writes from other processors.
3. Memory ordering obeys causality (memory ordering respects transitive visibility).
4. Any two stores are seen in a consistent order by processors other than those performing the stores
5. Locked instructions have a total order.

MFENCE – Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Re-read x86 vendor prose specifications with x86-TSO op/ax in mind

Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3 Ch.8, page 3056 (note that the initial contents page only covers Vol.1; Vol.3 starts on page 2783)
8.2.2 Memory Ordering in P6 and More Recent Processor Families The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

1. Reads are not reordered with other reads.x86-TSO-op: instructions are not reordered, but the buffering has a similar effect for `[W];pod;[R]`
2. Writes are not reordered with older reads.x86-TSO-ax: does the order of "reordered" match `ob`?
3. Writes to memory are not reordered with other writes [...]
4. Reads may be reordered with older writes to different locations but not with older writes to the same location.
5. Reads or writes cannot be reordered with locked instructions
6. Reads cannot pass earlier MFENCE instructions.
7. Writes cannot pass earlier MFENCE instructions.
8. MFENCE instructions cannot pass earlier reads or writes.

In a multiple-processor system, the following ordering principles apply:

1. Writes by a single processor are observed in the same order by all processors.
2. Writes from an individual processor are NOT ordered with respect to the writes from other processors.
3. Memory ordering obeys causality (memory ordering respects transitive visibility).
4. Any two stores are seen in a consistent order by processors other than those performing the stores
5. Locked instructions have a total order.

MFENCE – Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Re-read x86 vendor prose specifications with x86-TSO op/ax in mind

Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3 Ch.8, page 3056 (note that the initial contents page only covers Vol.1; Vol.3 starts on page 2783)
8.2.2 Memory Ordering in P6 and More Recent Processor Families The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

1. Reads are not reordered with other reads.x86-TSO-op: instructions are not reordered, but the buffering has a similar effect for `[W];pod;[R]`
2. Writes are not reordered with older reads.x86-TSO-ax: does the order of "reordered" match `ob`?
3. Writes to memory are not reordered with other writes [...]
4. Reads may be reordered with older writes to different locations but not with older writes to the same location.
5. Reads or writes cannot be reordered with locked instructions
6. Reads cannot pass earlier is "cannot pass" the same as "cannot be reordered with"? MFENCE instructions.
7. Writes cannot pass earlier MFENCE instructions.
8. MFENCE instructions cannot pass earlier reads or writes.

In a multiple-processor system, the following ordering principles apply:

1. Writes by a single processor are observed in the same order by all processors.
2. Writes from an individual processor are NOT ordered with respect to the writes from other processors.
3. Memory ordering obeys causality (memory ordering respects transitive visibility).
4. Any two stores are seen in a consistent order by processors other than those performing the stores
5. Locked instructions have a total order.

MFENCE – Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Re-read x86 vendor prose specifications with x86-TSO op/ax in mind

Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3 Ch.8, page 3056 (note that the initial contents page only covers Vol.1; Vol.3 starts on page 2783)
8.2.2 Memory Ordering in P6 and More Recent Processor Families The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

1. Reads are not reordered with other reads.x86-TSO-op: instructions are not reordered, but the buffering has a similar effect for `[W];pod;[R]`
2. Writes are not reordered with older reads.x86-TSO-ax: does the order of "reordered" match `ob`?
3. Writes to memory are not reordered with other writes [...]
4. Reads may be reordered with older writes to different locations but not with older writes to the same location.
5. Reads or writes cannot be reordered with locked instructions
6. Reads cannot pass earlier is "cannot pass" the same as "cannot be reordered with"? MFENCE instructions.
7. Writes cannot pass earlier MFENCE instructions.
8. MFENCE instructions cannot pass earlier reads or writes.

In a multiple-processor system, the following ordering principles apply:

1. Writes by a single processor are observed in the same order by all processors.
2. Writes from an individual processor are NOT ordered with respect to the writes from other processors.
3. Memory ordering obeys causality (memory ordering respects transitive visibility).of what order? Is "memory ordering" `ob`? Is it the order of R and D events?
4. Any two stores are seen in a consistent order by processors other than those performing the stores
5. Locked instructions have a total order.

MFENCE – Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Re-read x86 vendor prose specifications with x86-TSO op/ax in mind

Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3 Ch.8, page 3056 (note that the initial contents page only covers Vol.1; Vol.3 starts on page 2783)
8.2.2 Memory Ordering in P6 and More Recent Processor Families The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

1. Reads are not reordered with other reads.x86-TSO-op: instructions are not reordered, but the buffering has a similar effect for `[W];pod;[R]`
2. Writes are not reordered with older reads.x86-TSO-ax: does the order of "reordered" match `ob`?
3. Writes to memory are not reordered with other writes [...]
4. Reads may be reordered with older writes to different locations but not with older writes to the same location.
5. Reads or writes cannot be reordered with locked instructions
6. Reads cannot pass earlier is "cannot pass" the same as "cannot be reordered with"? MFENCE instructions.
7. Writes cannot pass earlier MFENCE instructions.
8. MFENCE instructions cannot pass earlier reads or writes.

In a multiple-processor system, the following ordering principles apply:

1. Writes by a single processor are observed in the same order by all processors.
2. Writes from an individual processor are NOT ordered with respect to the writes from other processors.
3. Memory ordering obeys causality (memory ordering respects transitive visibility).of what order? Is "memory ordering" `ob`? Is it the order of R and D events?
4. Any two stores are seen in a consistent order by processors other than those performing the stores
5. Locked instructions have a total order.

MFENCE – Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.microarchitectural?

Experimental validation

Essential – but not *enough* by itself:

- the architectural intent is typically looser than any specific hardware
- one can't always determine whether a strange observed behaviour is a hardware bug or not without asking the architects – it's their call

Experimental validation relies on having a good test suite and test harness, that exercises corners of the model and of hardware implementations

...and it relies on making the model *executable as a test oracle* – we make operational and axiomatic models *exhaustively executable* for (at least) litmus tests.

Interesting tests

We can usually restrict to tests with some potential non-SC behaviour
(assuming no h/w bugs)

By the SC characterisation theorem, these are those with a cycle in `po|rf|co|fr`

(*"critical cycles"* [37])

Generating tests

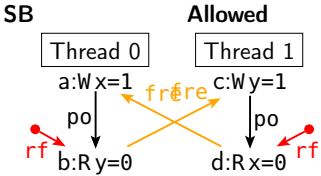
Hand-writing tests is sometimes necessary, but it's also important to be able to auto-generate them.

This is made much easier by the fact that we have executable-as-test-oracle models: we can generate any potentially interesting test, and then use the models to determine the model-allowed behaviour.

Usually, interesting tests have at least one potential execution, consistent with the instruction-local semantics, which is a critical cycle

Tests only identify an interesting outcome; they don't specify whether it is allowed or forbidden. And in fact we compare all outcomes, not just that one.

Generating a single test from a cycle



Use `diyone7` to generate a single test from a cycle, e.g. `Fre PodWR Fre PodWR`:

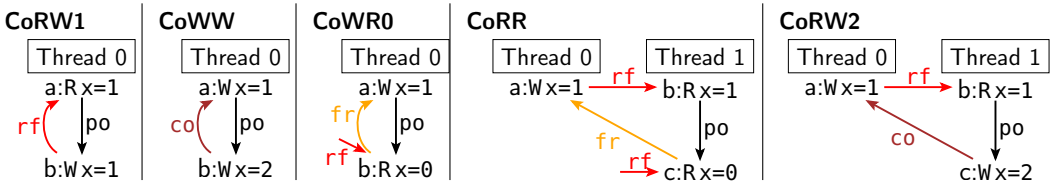
```
diyone7 -arch X86_64 -type uint64_t -name SB "Fre PodWR Fre PodWR"
```

```
X86_64 SB
"Fre PodWR Fre PodWR"
Generator=diyone7 (version 7.56)
Prefetch=0:x=F,0:y=T,1:y=F,1:x=T
Com=Fr Fr
Orig=Fre PodWR Fre PodWR
Align=
{
uint64_t y; uint64_t x; uint64_t 1:rax; uint64_t 0:rax;
}
 P0            | P1            ;
 movq $1,(x)   | movq $1,(y)   ;
 movq (y),%rax | movq (x),%rax ;
exists (0:rax=0 /\ 1:rax=0)
```

Documentation: http://diy.inria.fr/doc/gen.html

For small tests, we can be exhaustive, in various ways

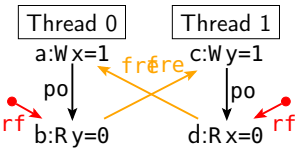
e.g. the earlier coherence tests



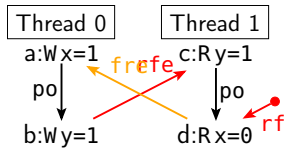
Basic 4-edge test shapes

All 4-edge critical-cycle tests, with a pod pair of different-location memory accesses on each thread. There are only six:

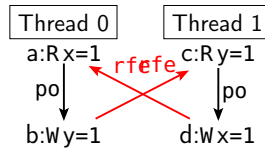
SB



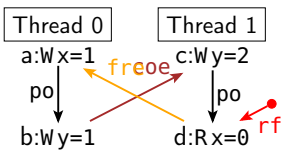
MP



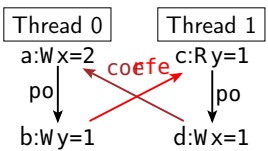
LB



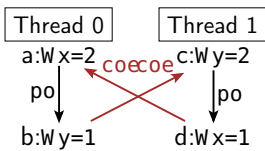
R



S



2+2W



Generating the basic 4-edge tests

Use a configuration file `X86_64-basic-4-edge.conf`

```
# diy7 configuration file for basic x86 tests with four pod or rf/co/fr external edges
-arch X86_64
-nprocs 2
-size 4
-num false
-safe Pod**,Pos**,Fre,Rfe,Wse
-mode critical
-type uint64_t
```

(Ws, for "write serialisation", is original diy7 syntax for coherence co, updated in newer versions)

Then

```
diy7 -conf X86_64-basic-4-edge.conf
```

generates those six critical-cycle tests

Running a batch of tests on hardware using `litmus`

```
litmus7 -r 100 src-X86_64-basic-4-edge/@all > run-hw.log
```

This runs each of those tests 10^7 times, logging to `run-hw.log`. It takes $\sim$40s.

For serious testing, one should increase that by 10–1000, and typically will be using many more tests.

This log contains, for each test, the histogram of observed final states. It also records whether the identified final-state condition was observed or not.

```
Test SB Allowed            (* NB: don't get confused by these "Allowed"s, or the "Ok"s - just look at the "Observation" line *)
Histogram (4 states)
95    *>0:rax=0; 1:rax=0;
4999871:>0:rax=1; 1:rax=0;
4999876:>0:rax=0; 1:rax=1;
158   :>0:rax=1; 1:rax=1;
[...]
Observation SB Sometimes 95 9999905
```

Running a batch of tests in x86-TSO operational using rmem

```
rmem -model tso -interactive false -eager true -q
  src-X86_64-basic-4-edge/@all > run-rmem.log.tmp

cat run-rmem.log.tmp | sed 's/RAX/rax/g' | sed 's/RBX/rbx/g' > run-rmem.log
```

This runs each of those tests exhaustively in the x86-TSO operational model, logging to `run-rmem.log`. And, ahem, fixes up the register case.

This log contains, for each test, a list of the final states that are possible in the operational model:

```
Test SB Allowed
States 4
0:rax=0; 1:rax=0;
0:rax=0; 1:rax=1;
0:rax=1; 1:rax=0;
0:rax=1; 1:rax=1;
[...]
Observation SB Sometimes 1 3
```

Running a batch of tests in x86-TSO axiomatic using herd

```
herd7 -cat x86-tso.cat src-X86_64-basic-4-edge/@all > run-herd.log
```

This runs each of those tests exhaustively in the x86-TSO axiomatic model, logging to `run-herd.log`.

This log contains, for each test, a list of the final states that are possible in the axiomatic model:

```
Test SB Allowed
States 4
0:rax=0; 1:rax=0;
0:rax=0; 1:rax=1;
0:rax=1; 1:rax=0;
0:rax=1; 1:rax=1;
[...]
Observation SB Sometimes 1 3
```

Herd web interface: `http://diy.inria.fr/www`

Comparing results

```
$ mcompare7 -nohash run-hw.log run-rmem.log run-herd.log
*Diffs*
    |Kind | run-hw.log          run-rmem.log run-herd.log
------------------------------------------------------------
------------------------------------------------------------
2+2W|Allow| [x=1; y=1;]         ==           ==
    |No   | [x=1; y=2;]
    |     | [x=2; y=1;]
------------------------------------------------------------
LB  |Allow| [0:rax=0; 1:rax=0;] ==           ==
    |No   | [0:rax=0; 1:rax=1;]
    |     | [0:rax=1; 1:rax=0;]
------------------------------------------------------------
MP  |Allow| [1:rax=0; 1:rbx=0;] ==           ==
    |No   | [1:rax=0; 1:rbx=1;]
    |     | [1:rax=1; 1:rbx=1;]
------------------------------------------------------------
[...]
------------------------------------------------------------
SB  |Allow| [0:rax=0; 1:rax=0;] ==           ==
    |Ok   | [0:rax=0; 1:rax=1;]
    |     | [0:rax=1; 1:rax=0;]
    |     | [0:rax=1; 1:rax=1;]
```

Or use `-pos <file>` and `-neg <file>` to dump positive and negative differences.
Normally we would check test hashes for safety, without `-nohash`, but they have temporarily diverged between the tools.
One can also use this to compare models directly against each other.

Generating more tests

Allow up to 6 edges on up to 4 threads, and include MFENCE edges

diy7 configuration file `X86_64-basic-6-edge.conf`

```
# diy7 configuration file for basic x86 tests with six pod or rf/co/fr external edges
-arch X86_64
-nprocs 4
-size 6
-num false
-safe Pod**,Pos**,Fre,Rfe,Wse,MFenced**,MFences**
-mode critical
-type uint64_t
```

Then

```
diy7 -conf X86_64-basic-6-edge.conf
```

generates 227 critical-cycle tests, including SB, SB+mfence+po, SB+mfences, ..., IRIW, ...

Generating more more tests

To try to observe some putative relaxation (some edge that we think should not be in ob), remove it from the `-safe` list and add it to `-relax`, then `diy7` will by default generate cycles of exactly one relaxed edge and some safe edges.

x86-rfi.conf

```
#rfi x86 conf file
-arch X86
-nprocs 4
-size 6
-name rfi
-safe PosR* PodR* PodWW PosWW Rfe Wse Fre FencesWR FencedWR
-relax Rfi
```

x86-podwr.conf

```
#podrw x86 conf file
-arch X86
-nprocs 4
-size 6
-name podwr
-safe Fre
-relax PodWR
```

From http://diy.inria.fr/doc/gen.html#sec52
Many more options in the docs

Generating more more tests

There's a modest set of x86 tests at:

`https://github.com/litmus-tests/litmus-tests-x86`

Arm-A, IBM Power, and RISC-V

Armv8-A application-class architecture

Armv8-A is Arm's main *application profile* architecture. It includes the AArch64 execution state, supporting the A64 instruction-set, and AArch32, supporting A32 and T32. Arm also define Armv8-M and Armv8-R profiles, for microcontrollers and real-time, and ARMv7 and earlier are still in use.

Many cores designed by Arm and by others, in many SoCs. https://en.wikipedia.org/wiki/Comparison_of_ARMv8-A_cores

- Samsung Exynos 7420 and Qualcomm Snapdragon 810 SoCs, each containing 4xCortex-A57+4xCortex-A53 cores, both ARMv8.0-A
- Apple A14 Bionic SoC (in iPhone 12) https://en.wikipedia.org/wiki/Apple_A14

Each core implements some specific version (and optional features) of the architecture, e.g. Cortex-A57 implements Armv8.0-A. Armv8-A architecture versions:

2013	A.a	Armv8.0-A (first non-confidential beta)
2016	A.k	Armv8.0-A (EAC)
2017	B.a	Armv8.1-A (EAC), Armv8.2-A (Beta) (simplification to MCA)
...		
2020	F.c	Armv8.6-A (initial EAC)

IBM Power architecture

The architecture of a line of high-end IBM server and supercomputer processors, now under the OpenPOWER foundation

Date	Architecture version	Processor
2004	Power ISA 2.03	POWER5
2007	Power ISA 2.03	POWER6
2010	Power ISA 2.06	POWER7
2014	Power ISA 2.07	POWER8
2017	Power ISA 3.08B	POWER9
2021	Power ISA 3.1	POWER10
2021	Power ISA 3.1B	
2024	Power ISA 3.1C	

Power ISA 3.0B

POWER10: 240 hw threads/socket

POWER9: 96 hw threads/die https://en.wikipedia.org/wiki/POWER9

POWER 8: up to 192 cores, each with up to 8 h/w threads https://en.wikipedia.org/wiki/POWER8

Power7: IBM's Next-Generation Server Processor Kalla, Sinharoy, Starke, Floyd

RISC-V

Nascent open standard architecture, originated UCB, now under RISC-V International – a large industry and academic consortium

Cores available or under development from multiple vendors

- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [34]
- The RISC-V Instruction Set Manual Volume II: Privileged Architecture [35]

Industry collaborations

2007	we started trying to make sense of the state of the art
2008/2009	discussion, still ongoing, with IBM Power and ARM architects
2017–	contributed to RISC-V memory-model task group
2018	RISC-V memory-model spec ratified
2018	Arm simplified their concurrency model and included a formal definition

x86

- programmers can assume instructions execute in program order, but with FIFO store buffer
- (actual hardware may be more aggressive, but not visibly so)

ARM, IBM POWER, RISC-V

- by default, instructions can observably execute out-of-order and speculatively
- ...except as forbidden by coherence, dependencies, barriers
- much weaker than x86-TSO
- similar but not identical to each other
- (for RISC-V, this is "RVWMO"; the architecture also defines an optional "RVTSO", the Ztso extension)

Abstract microarchitecture – informally

As before:

Observable relaxed-memory behaviour arises from hardware optimisations

So we have to understand just enough about hardware to explain and define the envelopes of programmer-observable (non-performance) behaviour that comprise the architectures.

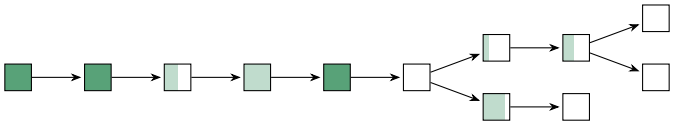
But no more – see a Computer Architecture course for that.

(Computer Architecture courses are typically largely about hardware implementation, aka *microarchitecture*, whereas here we focus exactly on *architecture* specification.)

Abstract microarchitecture – informally

Many observable relaxed phenomena arise from out-of-order and speculative execution.

Each hardware thread might have many instructions in flight, executing out-of-order, and this may be speculative: executing even though there are unresolved program-order-predecessor branches, or po-predecessor instructions that are not yet known not to raise an exception, or po-predecessor instructions that might access the same address in a way that would violate coherence.



Think of these as a per-thread tree of instruction instances, some finished ■ and some not.

The hardware checks, and rolls back as needed, to ensure that none of this violates the architected guarantees about sequential per-thread execution, coherence, or synchronisation.

Abstract microarchitecture – informally

Observable relaxed phenomena also arise from the hierarchy of store buffers and caches, and the interconnect and cache protocol connecting them.

We've already seen the effects of a FIFO store buffer, in x86-TSO. One can also have observably hierarchical buffers, as we discussed for IRIW; non-FIFO buffers; and buffering of read requests in addition to writes, either together with writes or separately. High-performance interconnects might have separate paths for different groups of addresses; high-performance cache protocols might lazily invalidate cache lines; and certain atomic RMW operations might be done "in the interconnect" rather than in the core.

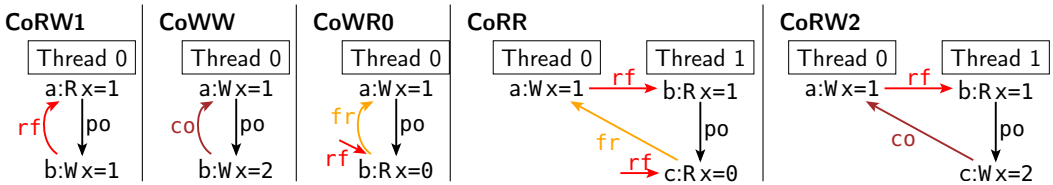
We descibe all of this as the "storage subsystem" of a hardware implementation or operational model.

Some phenomena can be seen as arising either from thread or storage effects – then we can choose, in an operational model, whether to include one, the other, or both.

Phenomena

Coherence

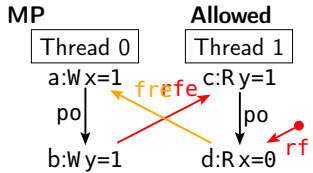
Coherence



Still all forbidden

Out-of-order accesses

Out-of-order pod WW and pod RR: MP (Message Passing)



MP AArch64

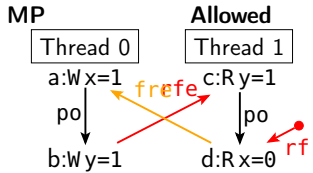
Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1] //a`	`LDR X0,[X1] //c`
`STR X0,[X2] //b`	`LDR X2,[X3] //d`

Allowed: `1:X0=1; 1:X2=0;`

Arm: YYYYY YYYYY YYYYY NY Power:Y RISC-V:N

Out-of-order pod WW and pod RR: MP (Message Passing)



MP AArch64

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1] //a`	`LDR X0,[X1] //c`
`STR X0,[X2] //b`	`LDR X2,[X3] //d`

Allowed: `1:X0=1; 1:X2=0;`

Arm: YYYYY YYYYY YYYYY NY Power:Y RISC-V:N

Microarchitecturally, as x and y are distinct locations, this could be:

- thread: out-of-order execution of the writes
- thread: out-of-order satisfaction of the reads
- non-FIFO write buffering
- storage subsystem: write *propagation* in either order

We don't distinguish between those when we say WW and RR can be (observably) out-of-order
We check both WW and RR are possible by adding a barrier (MP+po+fen and MP+fen+po)

Experimental data

arch	key	directory	device	SoC	core	arch version	release
Armv8-A	a	ec2-a1	Amazon EC2 A1 instance	AWS Graviton or Graviton 2	16xCortex-A72 or 64xNeoverse N1		
	b	BCM2711	Raspberry Pi 4 Model B	Broadcom BCM2711	4xCortex-A72	Armv8-A	2016
	c	h955	LG H955 phone	Qualcomm Snapdragon 810	4xCortex-A57/A53		2015
	d	AMD	?	? AMD Opteron A1100	4xCortex-A57		2016
	e	Juno	Arm Juno development board		2xCortex-A57+4xCortex-A53		
	f	Kirin6220	HiKey development board	HiSilicon Kirin 620	8xCortex-A53		
	g	HelioG25	?	MediaTek Helio G25	8xCortex-A53		2020
	h	S905	ODROID-C2 development board	Amlogic S905	4xCortex-A53		
	i	Snapdragon425		Qualcomm Snapdragon 425	4xCortex-A53		
	j	a10x-fusion	?	Apple A10X Fusion	3xHurricane+3xZephyr	Armv8.1-A	
	k	iphone7	Apple iPhone 7	Apple A10 Fusion	2xHurricane+2xZephyr	Armv8.1-A	2016
	l	ipadair2	Apple iPad air 2	Apple A8X	3xTyphoon	Armv8-A	2014
	m	APM883208	?	Applied Micro APM883208	8xStorm	Armv8-A	2012
	n	Cavium	?	? Cavium ThunderX or X2			
	o	Exynos9	?	? Samsung, could be custom or A77 or A55 or A53	?		
	p	nexus9	Google Nexus 9 tablet	NVIDIA Tegra K1	2xDenver	Armv8-A	2014
	q	openq820	Open-Q 820 development kit	Qualcomm Snapdragon 820 (APQ 8096)	4xQualcomm Kryo		2016
Power	r	bim		POWER7			
RISC-V	s		HiFi board	SiFive Freedom U540 SoC			

We'll show experimental data for Arm, Power, and RISC-V in an abbreviated form: Y/N indicating whether the final state is observed or not, or – for no data, for each of several hardware implementations, for each architecture. Detailed results for the tests in these slides are at Page 520. Key: Arm:$^{\text{abcde fghij}}_{\text{klmno pq}}$ Power:r RISC-V:s

This shows only some of the data gathered over the years, largely by Luc Maranget and Shaked Flur. More details of the former at
http://cambium.inria.fr/~maranget/cats7/model-aarch64/

Architectural intent and model behaviour

Except where discussed, for all these examples the architectural intent, operational model, and axiomatic model all coincide, and are the same for Armv8-A, IBM Power, and RISC-V.

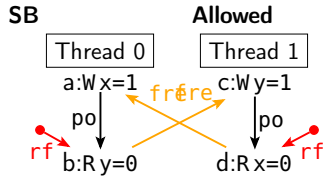
We write Allowed or Forbidden to mean the given execution is allowed or forbidden in all these.

Generally, if the given execution is Allowed, that means programmers should not depend on any program idiom involving that shape; additional synchronisation will have to be added.

Comparing models and test results

model	experimental observation	conclusion
Allowed	Y	ok
Allowed	N	ok, but model is looser than hardware (or testing not aggressive)
Forbidden	Y	model not sound w.r.t. hardware (or hardware bug)
Forbidden	N	ok

Out-of-order pod WR: SB ("Store Buffering")



SB AArch64

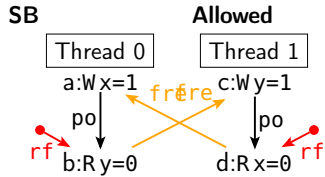
Initial state: `0:X3=y; 0:X1=x; 0:X0=1; 0:X2=0; 1:X3=x; 1:X1=y; 1:X0=1; 1:X2=0; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1]//a`	`STR X0,[X1]//c`
`LDR X2,[X3]//b`	`LDR X2,[X3]//d`

Allowed: `0:X2=0; 1:X2=0;`

Arm: YYYYY YYYYY YYYYY NY Power:Y RISC-V:N

Out-of-order pod WR: SB ("Store Buffering")



SB AArch64

Initial state: `0:X3=y; 0:X1=x; 0:X0=1; 0:X2=0; 1:X3=x; 1:X1=y; 1:X0=1; 1:X2=0; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1]//a`	`STR X0,[X1]//c`
`LDR X2,[X3]//b`	`LDR X2,[X3]//d`

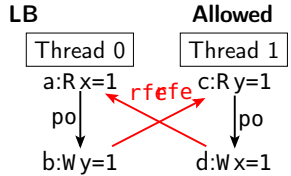
Allowed: `0:X2=0; 1:X2=0;`

Arm: YYYYY YYYYY YYYYY NY Power:Y RISC-V:N

Microarchitecturally:

- pipeline: out-of-order execution of the store and load
- storage subsystem: write buffering

Out-of-order pod RW: LB ("Load Buffering")



LB AArch64

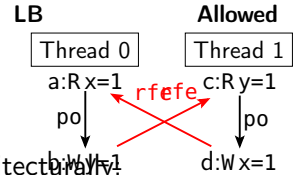
Initial state: `0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=0; y=0; x=0;`

Thread 0	Thread 1
`LDR X0,[X1]//a`	`LDR X0,[X1]//c`
`STR X2,[X3]//b`	`STR X2,[X3]//d`

Allowed: `0:X0=1; 1:X0=1;`

Arm: NNNNN NNNNN NNNNN NY Power:N RISC-V:N

Out-of-order pod RW: LB ("Load Buffering")



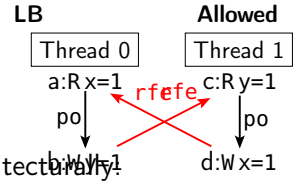
LB	AArch64
Initial state: 0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=0; y=0; x=0;	
Thread 0	Thread 1
LDR X0,[X1] //a STR X2,[X3] //b	LDR X0,[X1] //c STR X2,[X3] //d
Allowed: 0:X0=1; 1:X0=1;	

Arm: NNNNN NNNNN NNNNN NY Power:N RISC-V:N

Microarchitecturally:

- pipeline: out-of-order execution of the store and load
- storage subsystem: read-request buffering

Out-of-order pod RW: LB ("Load Buffering")



LB AArch64

Initial state: `0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=0; y=0; x=0;`

Thread 0	Thread 1
`LDR X0,[X1] //a`	`LDR X0,[X1] //c`
`STR X2,[X3] //b`	`STR X2,[X3] //d`

Allowed: `0:X0=1; 1:X0=1;`

Arm: NNNNN NNNNN NNNNN NY Power:N RISC-V:N

Microarchitecturally:

- pipeline: out-of-order execution of the store and load
- storage subsystem: read-request buffering

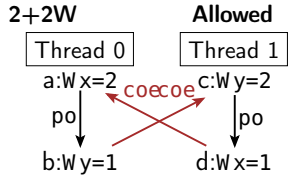
Architecturally allowed, but unobserved on most devices

Why the asymmetry between reads and writes (WR SB vs RW LB)? For LB, the hardware might have to make writes visible to another thread before it knows that the reads won't fault, and then roll back the other thread(s) if they do – but hardware typically treats inter-thread writes as irrevocable. In contrast, re-executing a read that turns out to have been satisfied too early is thread-local, relatively cheap.

Why architecturally allowed? Some hardware has exhibited LB, presumed via read-request buffering. But mostly this seems to be on general principles, to maintain flexibility.

However, architecturally allowing LB interacts very badly with compiler optimisations, making it very hard to define sensible programming language models – we return to this later.

Out-of-order pod WW again: 2+2W



2+2W AArch64

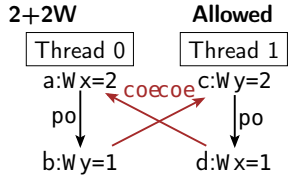
Initial state: `0:X3=y; 0:X2=1; 0:X1=x; 0:X0=2; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=2; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1] //a`	`STR X0,[X1] //c`
`STR X2,[X3] //b`	`STR X2,[X3] //d`

Allowed: `y=2; x=2;`

Arm: YYYYY YYYYY YNYYY NY Power:- RISC-V:N

Out-of-order pod WW again: 2+2W



2+2W AArch64

Initial state: `0:X3=y; 0:X2=1; 0:X1=x; 0:X0=2; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=2; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1] //a`	`STR X0,[X1] //c`
`STR X2,[X3] //b`	`STR X2,[X3] //d`

Allowed: `y=2; x=2;`

Arm: YYYYY YYYYY / YNYYY NY Power:- RISC-V:N

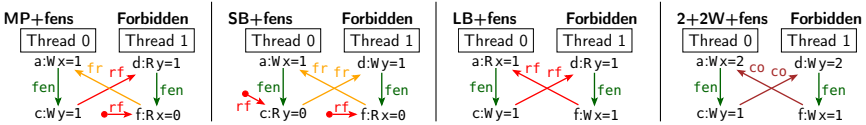
Microarchitecturally:

- pipeline: out-of-order execution of the stores
- storage subsystem: non-FIFO write buffering

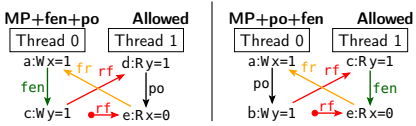
Barriers

Enforcing Order with Barriers

Each architecture has a variety of memory barrier (or fence) instructions. For normal code, the ARMv8-A `dmb sy`, POWER `sync`, and RISC-V `fence rw,rw` prevent observable reordering of any pair of loads and stores. Where these behave the same, we just write `fen`, so e.g. the Armv8-A version of MP+fen+po is MP+dmb.sy+po. Adding `fen` between both pairs of accesses makes the preceding tests forbidden:



Adding `fen` on just one thread leaves them allowed. For MP, this confirms WW and RR pod reordering are both observable:



Note: these barriers go *between* accesses, enforcing ordering between them; they don't synchronise with other barriers or other events.

Weaker Barriers

Enforcing ordering can be expensive, especially write-to-read ordering, so each architecture also provides various weaker barriers:

Armv8-A	`dmb ld`	read-to-read and read-to-write
	`dmb st`	write-to-write
Power	`lwsync`	read-to-read, write-to-write, and read-to-write
	`eieio`	write-to-write
RISC-V	`fence` *pred*`,`*succ*	*pred*,*succ* $\subseteq_{\text{nonempty}} \{\mathtt{r}, \mathtt{w}\}$

Plus variations for inner/outer shareable domains, IO, and systems features, all of which we ignore here

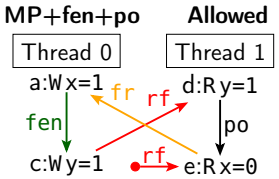
Note: later we'll see that preventing pairwise reordering is not all these do.

There are also various forms of labelled access, sometimes better or clearer than barriers.

Dependencies

Enforcing order with dependencies: read-to-read address dependencies

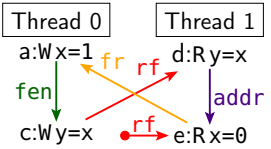
Recall MP+fen+po is allowed:



But in many message-passing scenarios we want to enforce ordering between the reads but don't need the full force (or cost) of a strong barrier. Dependencies give us that in some cases.

Enforcing order with dependencies: read-to-read address dependencies

MP+fen+addr.real Forbidden



Initial state: x=0; y=z; z=2;

Thread 0	Thread 1
x=1; y=&x;	r1=y; r2=∗r1;

Forbidden: 1:r1=y; 1:r2=0;

MP+dmb.sy+addr.real AArch64

Initial state: 0:X3=y; 0:X1=x; 0:X0=1;
1:X3=0; 1:X2=z; 1:X1=y; x=0; y=z; z=2;

Thread 0	Thread 1
STR X0,[X1] //a **DMB SY** //b STR X1,[X3] //c	LDR X2,[X1] //d LDR X3,[X2] //e

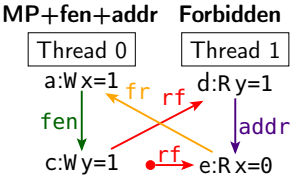
Forbidden: 1:X2=x; 1:X3=0;

Say there is an *address dependency* from a read to a program-order later read, written as an addr edge, if there is a chain of "normal" register dataflow from the first read's value to the address of the second. (What's "normal"? Roughly: via general-purpose and flag registers, excluding the PC, and for Armv8-A excluding writes by store-exclusives. System registers are another story, too.)

These are architecturally guaranteed to be respected.

Microarchitecturally, this means hardware cannot observably speculate the *value* used for the address of the second access.

Enforcing order with dependencies: natural vs artificial



Initial state: x=0; y=0;

Thread 0	Thread 1
x=1; y=1;	r1=y; r2=*(x+(r1^r1));

Forbidden: 1:r1=y; 1:r2=0;

MP+dmb.sy+addr AArch64

Initial state: 0:X2=y; 0:X1=x;
0:X0=1; 1:X4=x; 1:X1=y; 1:X0=0;
1:X3=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a DMB SY //b STR X0,[X2] //c	LDR X0,[X1] //d EOR X2,X0,X0 LDR X3,[X4,X2] //e

Forbidden: 1:X0=1; 1:X3=0;

Arm: --NNN N-N-N NNNNN NN Power:N RISC-V:N

Architectural guarantee to respect read-to-read address dependencies even if they are "artificial"/"false" (vs "natural"/"true"), i.e. if they could "obviously" be optimised away.

In simple cases one can intuitively distinguish between artificial and natural dependencies, but it's very hard to make a meaningful non-syntactic precise distinction in general: one would have to somehow bound the information available to optimisation, and optimisation is w.r.t. the machine semantics, which itself involves dependencies.

Enforcing order with dependencies: intentional artificial dependencies

That architectural guarantee means that introducing an artificial dependency can sometimes be a useful assembly programming idiom for enforcing read-to-read (or read-to-write) order.

In some architectures one can enforce similar orderings with a labelled access, e.g. the Arm release/acquire access instructions, which may or may not be preferable in any particular situation.

Enforcing order with dependencies: in high-level languages?

But beware! These and certain other dependencies are guaranteed to be respected by these architectures, but not by C/C++. Conventional compiler optimisations will optimise them away, e.g. replacing `r2^r2` by `0`, and then the compiler or hardware might reorder the now-independent accesses.

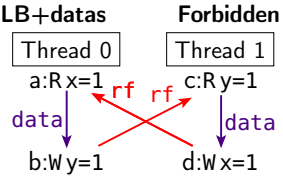
Inlining and link-time optimisation (and value range analysis?) mean this can happen unexpectedly, and make it very hard to rule out – c.f. the original C++11 `memory_order_consume` proposal, which has turned out not to be implementable.

This is an open problem, as high-performance concurrent code (e.g. RCU in the Linux kernel) does rely on dependencies. Currently, one hopes the compilers won't remove the specific dependencies used.

Enforcing order with dependencies: read-to-write address dependencies

Read to write address dependencies are similarly respected.

Enforcing order with dependencies: read-to-write data dependencies



Initial state: x=0; y=0;	
Thread 0	Thread 1
r1=x; y=1+r1-r1;	r1=y; x=1+r1-r1;
Forbidden: 0:r1=1; 1:r1=1;	

LB+datas AArch64

Initial state: 0:X3=y; 0:X1=x; 0:X0=0; 1:X3=x; 1:X1=y; 1:X0=0; y=0; x=0;	
Thread 0	Thread 1

LDR X0,[X1]//a
EOR X2,X0,X0
ADD X2,X2,#1
STR X2,[X3]//b
	LDR X0,[X1]//c
	EOR X2,X0,X0
	ADD X2,X2,#1
	STR X2,[X3]//d

Forbidden: 0:X0=1; 1:X0=1;	

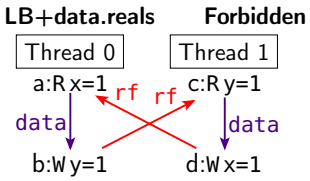
Arm: NNNNN N-NNN NNNNN NN Power:N RISC-V:N

Say there is an *data dependency* from a read to a program-order later write, written as a `data` edge, if there is a chain of "normal" register dataflow from the first read's value to the value of the write.

Read-to-write data dependencies are architecturally guaranteed to be respected, just as read-to-write address dependencies are (again irrespective of whether they are artificial).

(Note that because plain LB is not observable on most/all current implementations, experimental results for LB variants don't say much).

Enforcing order with dependencies: read-to-write data dependencies and no-thin-air



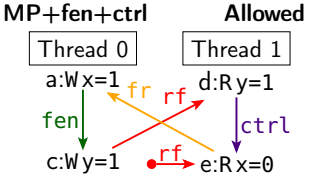
Initial state: x=0; y=0;	
Thread 0	Thread 1
r1=x; y=r1;	r1=y; x=r1;
Forbidden: 0:r1=1; 1:r1=1;	

LB+data.reals AArch64

Initial state: 0:X3=y; 0:X1=x; 1:X3=x; 1:X1=y; x=0; y=0;	
Thread 0	Thread 1
LDR X2,[X1]//a STR X2,[X3]//b	LDR X2,[X1]//c STR X2,[X3]//d
Forbidden: 0:X2=1; 1:X2=1;	

If read-to-write data dependencies weren't respected, then the architecture would allow *any value*. Such *thin-air reads* would make it impossible to reason about general code.

Not enforcing order with dependencies: read-to-read control dependencies



Initial state: x=0; y=0;	
Thread 0	Thread 1

x=1;
DMB SY;
y=1;
	r1=y;
	if (r1!=1) **goto** L;
	L:
	r2=x;

Allowed: 1:r1=1; 1:r2=0;	

MP+dmb.sy+ctrl AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;	
Thread 0	Thread 1
STR X0,[X1] //a **DMB SY** //b STR X0,[X2] //c	LDR X0,[X1] //d CBNZ X0,LC00 LC00: LDR X2,[X3] //e
Allowed: 1:X0=1; 1:X2=0;	

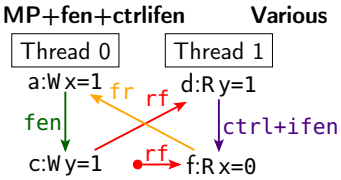
Arm: YYYYY Y-YYY YYYNY NY Power:Y RISC-V:N

Read-to-read control dependencies are not architecturally respected.

Microarchitecturally, the hardware might speculate past conditional branches and satisfy the second read early.

In this example the second read is reachable by both paths from the conditional branch, but the observable behaviour and architectural intent would be the same for a branch conditional on `r1 != 1` to after the second read. (Some ambiguity in Arm, [32, B2.3.2]?)

Enforcing order with dependencies: read-to-read ctrlifen dependencies



MP+dmb.sy+ctrlisb AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	CBNZ X0,LC00
STR X0,[X2] //c	LC00:
	ISB //e
	LDR X2,[X3] //f

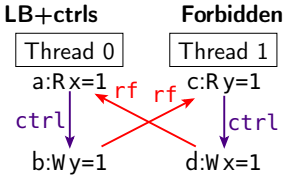
Forbidden: 1:X0=1; 1:X2=0;

Arm: NNNNN N-NNN NNNNN NN Power:N RISC-V:-

Read-to-read control dependencies are not architecturally respected.

But with an `isb` (Arm) or `isync` (Power) (generically, `ifen`) between the conditional branch and the second read, they are. The RISC-V `fence.i` does not have this strength.

Enforcing order with dependencies: read-to-write control dependencies



LB+ctrls AArch64

Initial state: 0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=0; y=0; x=0;

Thread 0	Thread 1
LDR X0,[X1]//a CBNZ X0,LC00 LC00: STR X2,[X3]//b	LDR X0,[X1]//c CBNZ X0,LC01 LC01: STR X2,[X3]//d

Forbidden: 0:X0=1; 1:X0=1;

Arm: NNNNN N-NNN NNNNN NN Power:N RISC-V:N

Read-to-write control dependencies *are* architecturally respected.

(even if the write is reachable by both paths from the conditional branch)

Microarchitecturally, one doesn't want to make uncommitted writes visible to other threads.

Enforcing Order with Dependencies: Summary

Read-to-read: address and control-isb/control-isync/control-fence.i dependencies respected; control dependencies *not* respected

Read-to-write: address, data, *and control* dependencies all respected (writes are not observably speculated, at least as far as other threads are concerned)

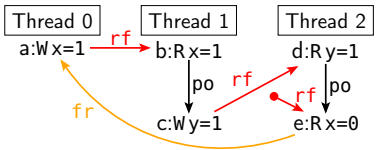
All whether natural or artificial.

Multi-copy atomicity

Iterated message-passing, x86

In the x86-TSO operational model, when a write has become visible to some other thread, it is visible to all other threads.

That, together with thread-local read-to-write ordering, means that iterated message-passing, across multiple threads, works on x86 without further ado
:



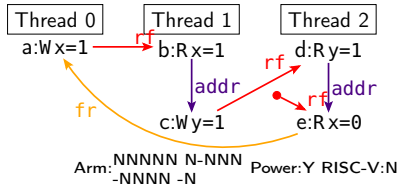
Initial state: x=0; y=0;		
Thread 0	Thread 1	Thread 2
x=1;	**while** (x==0) {}; y=1;	**while** (y==0) {}; r3=x;
Forbidden: 2:r3=0;		

WRC x86

Initial state: 1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;		
Thread 0	Thread 1	Thread 2
movq $1, (x) //a	movq (x), %rax //b movq $1, (y) //c	movq (y), %rax //d movq (x), %rbx //e
Forbidden: 1:rax=1; 2:rax=1; 2:rbx=0;		

Iterated message-passing

On Armv8, Power, and RISC-V, WRC would be allowed just by thread-local reordering. But what if we add dependencies to rule that out? Test WRC+addrs:



- IBM POWER: Allowed
- ARMv7-A and old ARMv8-A (first public beta, 2013 – first non-beta, June 2016): Allowed
- current ARMv8-A (March 2017 –) : Forbidden
- RISC-V: Forbidden

Multicopy atomicity

Say an architecture is *multicopy atomic* (MCA) if, when a write has become visible to some other thread, it is visible to all other threads.

And *non-multicopy-atomic* (non-MCA) otherwise.

So x86, Armv8-A (now), and RISC-V are MCA, and Power is non-MCA

Terminology: Arm say "other multicopy atomic" where we (and others) say MCA.
Terminology: "single-copy atomicity" is not the converse of MCA.

Multicopy atomicity: Arm strengthening

Arm strengthened the Armv8-A architecture, from non-MCA to MCA, in 2017

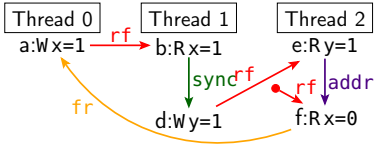
- Armv8-A implementations (by Arm and by its Architecture Partners) had not exploited the freedom that non-MCA permits, e.g.
  - shared pre-cache store buffers that allow early forwarding of data among a subset of threads, and
  - cache protocols that post snoop invalidations without waiting for their acknowledgement,

  partly as the common ARM bus architecture (AMBA) has always been MCA.
- Allowing non-MCA added substantial complexity to the model, esp. combined with the previous architectural desire for a model providing as much implementation freedom as possible, and the Armv8-A store-release/load-acquire instructions.
- Hence, in the Arm context, the potential performance benefits were not thought to justify the complexity of implementation, validation, and reasoning.

See [19, Pulte, Flur, Deacon,...].

Cumulative barriers

In a non-MCA architecture, e.g. current Power, one needs *cumulative* barriers to support iterated message-passing:



WRC+sync+addr Power

Initial state: 0:r2=x; 0:r1=1; 1:r4=y; 1:r3=1; 1:r2=x; 1:r1=0; 2:r5=x; 2:r2=y; 2:r1=0; 2:r4=0; y=0; x=0;

Thread 0	Thread 1

std r1,0(r2)//a	ld r1,0(r2) //b
	sync //c
	std r3,0(r4)//d

	Thread 2

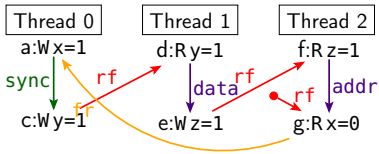
	ld r1,0(r2) //e
	xor r3,r1,r1
	ldx r4,r3,r5//f

Forbidden: 1:r1=1; 2:r1=1; 2:r4=0;

Here the sync keeps all writes that have propagated to Thread 1 (and its own events) before the sync (and hence before any writes by this thread after the sync) in order as far as other threads are concerned – so writes a and d are kept in order as far as reads e

Cumulative barriers, on the right

Cumulative barriers also ensure that chains of reads-from and dependency edges after such a barrier are respected:



ISA2+sync+data+addr Power

Initial state: `0:r3=y; 0:r2=x; 0:r1=1; 1:r5=z; 1:r4=1; 1:r2=y; 1:r1=0; 2:r5=x; 2:r2=z; 2:r1=0; 2:r4=0; z=0; y=0; x=0;`

Thread 0	Thread 1	Thread 2
`std r1,0(r2)//a`	`ld r1,0(r2) //d`	`ld r1,0(r2) //f`
`sync` `//b`	`xor r3,r1,r1`	`xor r3,r1,r1`
`std r1,0(r3)//c`	`add r3,r3,r4`	`ldx r4,r3,r5//g`
	`std r3,0(r5)//e`	

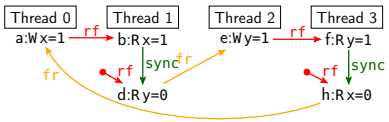
Forbidden: `1:r1=1; 2:r1=1; 2:r4=0;`

Explain in terms of write and barrier propagation:

- Writes (a) and (c) are separated by the barrier
- ...so for Thread 1 to read from (c), both (a) and the barrier have to propagate there, in that order
- But now (a) and (e) are separated by the barrier
- ...so before Thread 2 can read from (e), (a) (and the barrier) has to propagate there too
- and hence (g) has to read from (a), instead of the initial state.

Cumulative barriers

A strong cumulative barrier is also needed to forbid IRIW in a non-MCA architecture:



IRIW+syncs Power

Initial state: `0:r2=x; 1:r4=y; 1:r2=x; 2:r2=y; 3:r4=x; 3:r2=y;`

Thread 0	Thread 1	Thread 2	Thread 3

`li r1,1`
`stw r1,0(r2)//a`
	`lwz r1,0(r2)//b`
	`sync` `//c`
	`lwz r3,0(r4)//d`
		`li r1,1`
		`stw r1,0(r2)//e`
			`lwz r1,0(r2)//f`
			`sync` `//g`
			`lwz r3,0(r4)//h`

Forbidden: `1:r1=1; 1:r3=0; 3:r1=1; 3:r3=0;`

(the `lwsync` barrier does not suffice, even though it does locally order read-read pairs)

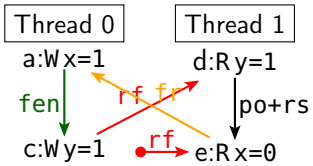
In operational-model terms, the sync's block po-later accesses until their "Group A" writes have been propagated to all other threads.

Further thread-local subtleties

These are various subtle cases that come up when defining architectural models that are good for arbitrary code, not just for simple idioms.

From a programmer's point of view, they illustrate some kinds of ordering that one might falsely imagine are respected.

Programmer-visible shadow registers



MP+fen+rs Pseudocode

Thread 0	Thread 1
x=1	r0=y
fen	r4=r0
y=1	r0=x
Allowed: 1:r4=1 ∧ 1:r0=0	

Arm: YYYYY Y-YYY YYYNY NY Power:Y RISC-V:-

MP+dmb.sy+rs AArch64

Initial state: `0:X3=y; 0:X1=x; 1:X3=x; 1:X1=y;`

Thread 0	Thread 1
`MOV W0,#1`	`LDR W0,[X1]//d`
`STR W0,[X1]//a`	`ADD W4,W0,#0`
`DMB SY` `//b`	`LDR W0,[X3]//e`
`MOV W2,#1`	
`STR W2,[X3]//c`	

Allowed: `1:X0=0; 1:X4=1;`

Reuse of the same architected register name does not enforce local ordering.

Microarchitecturally: there are shadow registers and register renaming.

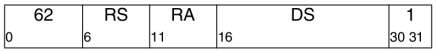
Register updates and dependencies

Armv8-A and Power include memory access instructions with addressing modes that, in addition to the load or store, do a register writeback or update of a modified value into a register used for address calculation, e.g.

```
STR <Xt>, [<Xn|SP>], #<simm>       (post-index)
STR <Xt>, [<Xn|SP>, #<simm>]!      (pre-index)

[...]
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
```

Store Doubleword with Update ***DS-form***

stdu RS,DS(RA)

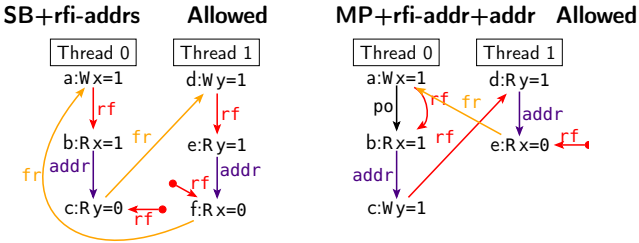



```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

But this apparent ordering of memory access before register writeback in the intra-instruction pseudocode is misleading: later instructions dependent on Xn or RA can go ahead as soon as the register dataflow is resolved.

Satisfying reads by write forwarding

As in x86, threads can see their own writes "early":

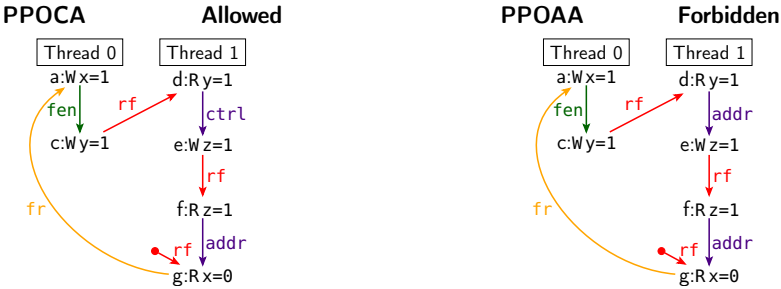


On the left is a variant of the SB+rfi-pos test we saw for x86, but with addr to prevent out-of-order satisfaction of the reads.

On the right is an essentially equivalent MP variant.

They both show write(s) visible to same-thread po-later reads before becoming visible to the other thread.

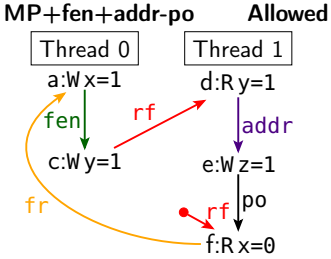
Satisfying reads by write forwarding on a speculative branch: PPOCA



In PPOCA, write e can be forwarded to f, resolving the address dependency to g and letting it be satisfied, before read d is (finally) satisfied and its control dependency is resolved.

Writes on speculatively executed branches are not visible to other threads, but can be forwarded to po-later reads on the same thread. Microarchitecturally: they can be read from an L1 store queue.

Satisfying reads before an unknown-address po-previous write: restarts



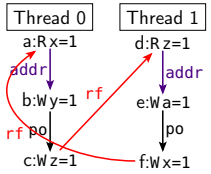
A microarchitecture that satisfies a load early, out-of-order, may later discover that this violates coherence, and have to *restart* the load – and any po-successors that were affected by it. (Speculative execution is not just speculation past branches.)

Here the Thread 0 writes are kept in order by `fen`. For Thread 1 f to read 0 early (but in an execution where d sees 1), i.e. for f to be satisfied before those writes propagate to Thread 1, f must be able to be restarted, in case resolving the address dependency revealed that e was to the same address as f, which would be a coherence violation.

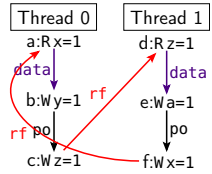
Committing writes before an unknown-address po-previous write

AKA "Might-access-same-address"

LB+addrs+WW Forbidden



LB+datas+WW Allowed



Address and data dependencies to a write both prevent the write being visible to other threads before the dependent value is fixed. But they are not completely identical: the existence of a address dependency to a write might mean that another program-order-later write cannot be propagated to another thread until it is known that the first write is not to the same address, otherwise there would be a coherence violation, whereas the existence of a data dependency to a write has no such effect on program-order-later writes that are already known to be to different addresses.

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB+addrs+WW	Forbid	0/30G	0/8.7G	0/208G	0/16G	0/23G	0/18G	0/2.1G
LB+datas+WW	Allow	0/30G	0/9.2G	0/208G	15k/6.3G	224/854M	0/18G	23/1.9G
LB+addrs+RW	Forbid	0/30G	0/6.0G	0/128G	0/13G	0/23G	0/16G	—

Intra-instruction ordering of address and data inputs to a write

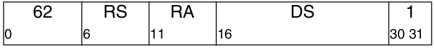
To let the later writes (c,f) in LB+datas+WW be propagated early, the addresses of the intervening writes (b,e) have to be resolvable even while there are still unresolved data dependencies to them.

If one interprets the intra-instruction pseudocode sequentially, that means the reads of registers that feed into the address have to precede those that feed into the data. (And there's no writeback into the data registers, so this is fine w.rt. that too.)

```
STR <Xt>,[<Xn|SP>],#<simm>    STR <Xt>,[<Xn|SP>,#<simm>]!

if n == 31 then
  CheckSPAlignment(); address = SP[];
else
  address = X[n];
if !postindex then
  address = address + offset;
if rt_unknown then
  data = bits(datasize) UNKNOWN;
else
  data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Store Doubleword with Update ***DS-form***

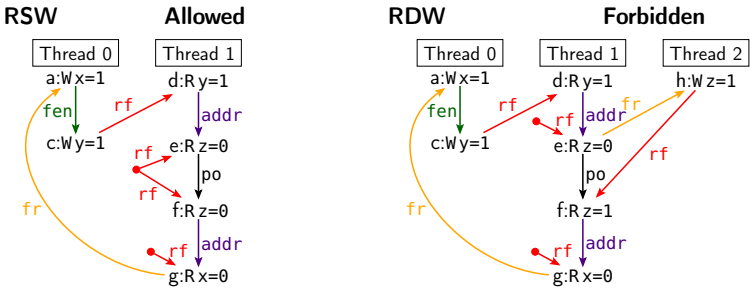
stdu RS,DS(RA)




```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

Satisfying reads from the same write: RSW and RDW

Coherence suggests that reads from the same address must be satisified in program order, but if they read from the same write event, that's not true. In RSW, f can be satisfied before e, resolving the address dependency to g and letting it be satisfied before d reads from c.

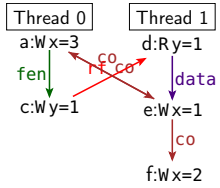


Microarchitecturally: the reads can in general be satisfied out-of-order, with coherence hazard checking that examines whether the x *cache line* changes between the two reads.

Making a write visible to another thread, following write subsumption

Conversely, one might think that, given two po-adjacent writes to the same address, the first could be discarded, along with any dependencies into it, as it is coherence-subsumed by the second. That would permit the following:

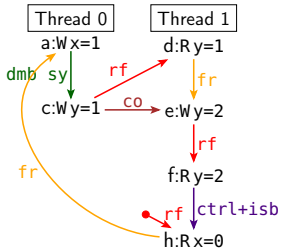
S+fen+data-wsi Forbidden



However, the Armv8-A and RISC-V architectures forbid this, as does our Power model and the Power architectural intent. Note that there is a subexecution S+fen+data, which all forbid, so allowing S+fen+data-wsi would require a more refined notion of coherence.

Non-atomic read satisfaction

MP+dmb.sy+fri-rfi-ctrlisb Various

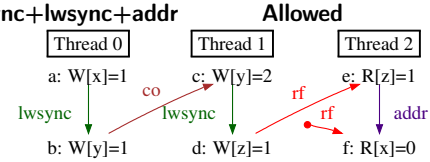


In our original PLDI11 [8] model for Power, to straightforwardly maintain coherence, the read d, write e, read f, isync (the Power analogue of the isb in the Arm version shown), and read h all have to commit in program order. However, for Arm, this behaviour was observable on at least one implementation, the Qualcomm APQ 8060, and the Arm architectural intent was determined to be that it was allowed.

Microarchitecturally, one can explain the behaviour in two ways. In the first, read d could be issued and then maintained in coherence order w.r.t. write e by keeping read requests and writes ordered in a storage hierarchy, letting e commit before the read is satisfied and hence letting f and h commit, still before d is satisfied. In the second, as write e is independent of read d in every respect except coherence, one can allow the thread to forward it to f and hence again commit the later instructions

Further Power non-MCA subtleties

Coherence and lwsync



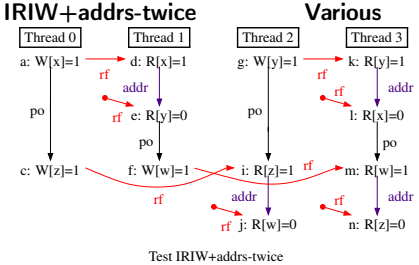
Test Z6.3+lwsync+lwsync+addr

This POWER example (blw-w-006 in [8]) shows that the transitive closure of lwsync and coherence does not guarantee ordering of write pairs. Operationally, the fact that the storage subsystem commits to b being before c in the coherence order has no effect on the order in which writes a and d propagate to Thread 2. Thread 1 does not read from either Thread 0 write, so they need not be sent to Thread 1, so no cumulativity is in play. In other words, coherence edges do not bring writes into the "Group A" of a POWER barrier. Microarchitecturally, coherence can be established late.

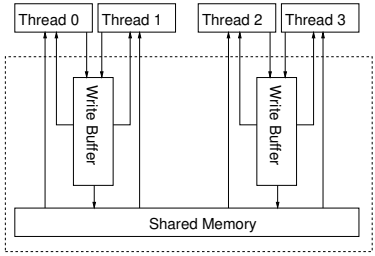
Replacing both lwsyncs by syncs forbids this behaviour. In the model, it would require a cycle in abstract-machine execution time, from the point at which a propagates to its last thread, to the Thread 0 sync ack, to the b write accept, to c propagating to Thread 0, to c propagating to its last thread, to the Thread 1 sync ack, to the d write accept, to d propagating to Thread 2, to e being satisfied, to f being satisfied, to a propagating to Thread 2, to a propagating to its last thread.

Armv8-A and RISC-V are (now) MCA (and do not have an analogue of `lwsync`), so there is no analogue of this example there.

Unobservable interconnect topology



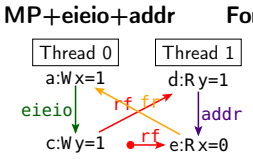
Test IRIW+addrs-twice

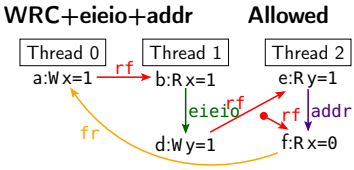


A simple microarchitectural explanation for IRIW+addrs would be a storage hierarchy in which Threads 0 and 1 are "neighbours", able to see each other's writes before the other threads do, and similarly Threads 2 and 3. If that were the only reason why IRIW+addrs were allowed, then one could only observe the specified behaviour for some specific assignments of the threads of the test to the hardware threads of the implementation (some specific choices of thread affinity). That would mean that two consecutive instances of IRIW+addrs, with substantially different assignments of test threads to hardware threads, could never be observed.

In fact, however, on some POWER implementations the cache protocol alone suffices to give the observed behaviour, symmetrically. ARMv8 and RISC-V are MCA, so no variants of IRIW+addrs are allowed there.

Power eieio





The Power `eieio` barrier *(Enforce In-order Execution of I/O)* orders pairs of same-thread writes as far as other threads are concerned, forbidding MP+eieio+addr. However, notwithstanding the architecture's mention of cumulativity [33, p.875], it does not prevent WRC+eieio+addr, because `eieio` does not order reads w.r.t. writes.

`eieio` also has other effects, e.g. for ordering for memory-mapped I/O, that are outside our scope here.

More features

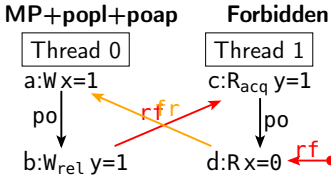
More features

- Armv8-A release/acquire accesses
- Load-linked/store-conditional (LL/SC)
- Atomics
- Mixed-size

For these, we'll introduce the basics, as they're important for concurrent programming, but we don't have time to be complete.

Armv8-A release/acquire accesses

Armv8-A release/acquire accesses



MP+popl+poap AArch64

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1] //a`	`LDAR X0,[X1]//c`
`STLR X0,[X2]//b`	`LDR X2,[X3] //d`

Forbidden: `1:X0=1; 1:X2=0;`

Armv8-A added *store-release* `STLR` and *load-acquire* `LDAR` instructions, which let message-passing idioms be expressed more directly, without needing barriers or dependencies.

In the (other-)MCA setting, their semantics is reasonably straightforward:

- a store-release keeps all po-before accesses before it, and
- a load-acquire keeps all po-after accesses after it.

(the above test only illustrates writes before a write-release and reads after a read-acquire, not all their properties)

Additionally, any po-related store-release and load-acquire are kept in that order.

Armv8-A acquirePC accesses

Armv8.3-A added "RCpc" variants of load-acquire, `LDAPR`, which lack the last property.

Compare with C/C++11 SC atomics and release/acquire atomics.

Armv8-A release/acquire accesses

See [19, Pulte, Flur, Deacon, et al.] for more details, and [17, Flur et al.] for discussion of Armv8 release/acquire in the previous non-MCA architecture

Together with the Arm architecture reference manual [32, Ch.B2 The AArch64 Application Level Memory Model]

Load-linked/store-conditional (LL/SC)

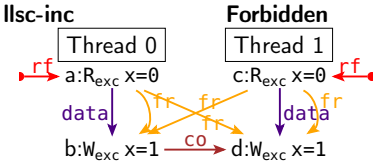
Load-linked/store-conditional (LL/SC)

LL/SC instructions, originating as a RISC alternative to compare-and-swap (CAS), provide simple optimistic concurrency – roughly, optimistic transactions on single locations.

Armv8-A	load exclusive / store exclusive	LDXR / STXR
Power	load and reserve / store conditional	lwarx / stwcx.
RISC-V	load-reserved / store-conditional	LR.D / SC.D

LL/SC atomic increment

Here are two concurrent increments of x, expressed with exclusives.



llsc-inc AArch64

Initial state: `0:X1=x; 1:X1=x; x=0;`

Thread 0	Thread 1
`LDXR X5,[X1] //a`	`LDXR X5,[X1] //c`
`ADD X5,X5,#1`	`ADD X5,X5,#1`
`STXR W6,X5,[X1] //b`	`STXR W6,X5,[X1] //d`

Forbidden: `0:X6=0; 1:X6=0; x=1;`

Exclusives should be used in matched pairs: a load-exclusive followed by a store exclusive to the same address, with some computation in between. The store exclusive can either:

- *succeed*, if the write can become the coherence immediate successor of the write the load read from (in this case the write is done and the success is indicated by a flag value), or
- *fail*, if that is not possible, e.g. because some other thread has already written a coherence successor, or for various other reasons. In this case the write is not done and the failure is indicated by a different flag value.

Often they are used within a loop, retrying on failure.

LL/SC – a few key facts:

Exclusives are not implicitly also barriers – load exclusives can be satisfied out of order and speculatively, though not until after all po-previous load exclusives and store exclusives are committed

...though Arm provide various combinations of exclusives and their release/acquire semantics

LL/SC is typically to a *reservation granule size*, not a byte address (architecturally or implementation-defined; microarchitecturally perhaps the store buffer or cache line size)

A store exclusive can succeed even if there are outstanding writes by different threads, so long as those can become coherence-later.

Arm, Power, and RISC-V differ w.r.t. what one can do within an exclusive pair, and what progress guarantees one gets.

Can a store exclusive commit to succeeding early? Likewise for an atomic RMW?

LL/SC – more details:

See [12, Sarkar et al.] for Power load-reserve/store-conditional, and [19, Pulte, Flur, Deacon, et al.] (especially its supplementary material `https://www.cl.cam.ac.uk/~pes20/armv8-mca/`), and [17, Flur et al.] for Armv8-A load-exclusive/store-exclusives.

Together with the vendor manuals:

- Power: [33, §1.7.4 Atomic Update]
- Arm: [32, Ch.B2 The AArch64 Application Level Memory Model]
- RISC-V: [34, Ch.8, "A" Standard Extension for Atomic Instructions, Ch.14 RVWMO Memory Consistency Model, App.A RVWMO Explanatory Material, App.B Formal Memory Model Specifications]

Atomics

Atomics

Armv8-A (in newer versions) and RISC-V also provide various *atomic* read-modify-write instructions

e.g. for Armv8-A: add, maximum, exclusive or, bit set, bit clear, swap, compare and swap

Mixed-size

Single-copy atomicity

Each architecture guarantees that certain sufficiently aligned loads and stores give rise to single single-copy-atomic reads and writes, where:

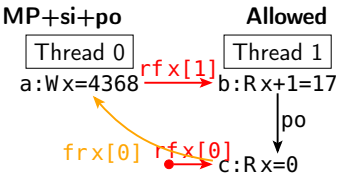
A single-copy-atomic read that reads a byte from a single-copy-atomic write must, for all other bytes of the common footprint, read either from that write or from a coherence successor thereof.

Misaligned accesses

Other, *"misaligned"* accesses architecturally give rise to multiple single-byte reads and writes, with no implicit ordering among them.

(In typical implementations, they might be split at cache-line or store-buffer-size boundaries but not necessarily into single bytes – more intentional architectural looseness)

Mixed-size: just a taste



MP+si+po AArch64

Initial state: `0:X1=0x1110; 0:X0=x; 1:X0=x; x=0x0;`

Thread 0	Thread 1
`STRH W1,[X0]//a`	`LDRB W1,[X0,#1]//b` `LDRB W2,[X0] //c`

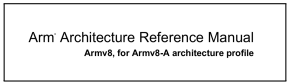
Allowed: `1:X1=0x11; 1:X2=0x0;`

Mixed-size: further details

See [18, Flur et al.] for more details for Power and Arm mixed-size.

ISA semantics

Architecture again



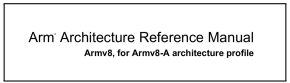
- **Concurrency**
  Subtle, and historically poorly specified, but small

  Operational models in executable pure functional code (rmem, in Lem)
  Axiomatic models in relational algebra (herd and isla-axiomatic)
- **Instruction-set architecture (ISA)**
  Relatively straightforward in detail, but large

  in Sail, a custom language for ISA specification

  integrated with rmem and isla-axiomatic concurrency models

Architecture again



Instruction-set architecture (ISA)

- **ARMv8-A:** Historically only pseudocode. Arm transitioned internally to mechanised ASL [38, 39, Reid et al.]. We automatically translate that ASL to Sail:
- **RISC-V:** Historically only text. We hand-wrote a Sail specification, now adopted by RISC-V Foundation.
- **Power:** Only pseudocode. We semi-automatically translated a fragment from an XML export of the Framemaker sources to Sail
- **x86:** Only pseudocode. We hand-wrote a fragment in Sail (and Patrick Taylor semi-automatically translated the Goel et al. ACL2 model)

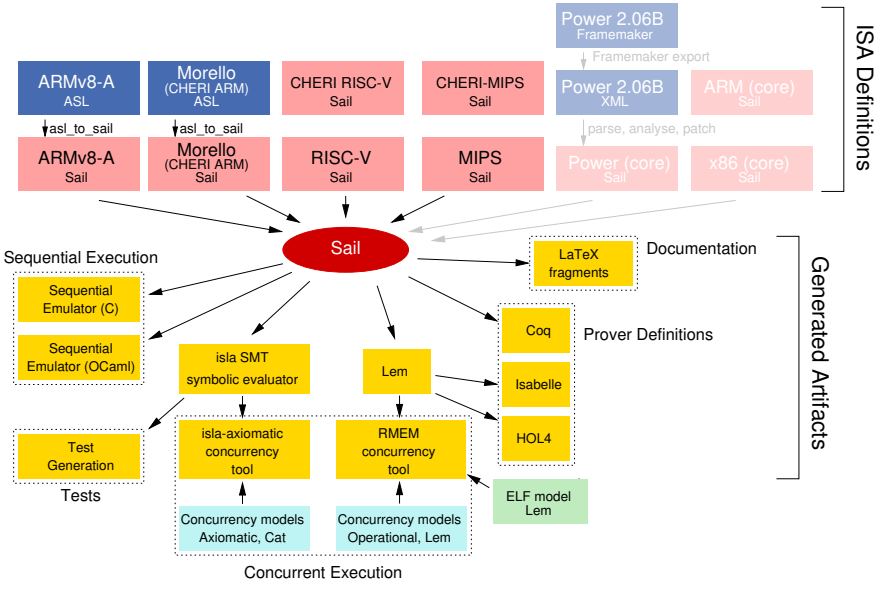
(the Power model and the first x86 model are in an old version of Sail)

Sail

Custom language for expressing the sequential behaviour of instructions (including decode, address translation, etc.) [20, Armstrong et al.],[16, Gray et al.]

- Imperative first-order language for ISA specification
- Lightweight dependent types for bitvectors (checked using Z3)
- Very simple semantics; good for analysis
- Behaviour of memory actions left to external memory model
  ... so can plug into tools for relaxed-memory concurrency
- Open-source public tooling

From Sail, we generate multiple artifacts...


Power 2.06B
Framemaker
Framemaker export
ARMv8-A
ASL
Morello
(CHERI ARM)
ASL
CHERI RISC-V
Sail
CHERI-MIPS
Sail
Power 2.06B
XML
ARM (core)
Sail
ISA Definitions
asl_to_sail
asl_to_sail
parse, analyse, patch
ARMv8-A
Sail
Morello
(CHERI ARM)
Sail
RISC-V
Sail
MIPS
Sail
Power (core)
Sail
x86 (core)
Sail
Sail
Sequential Execution
LaTeX
fragments
Documentation
Generated Artifacts
Sequential
Emulator (C)
Sequential
Emulator (OCaml)
isla SMT
symbolic evaluator
Lem
Coq
Isabelle
HOL4
Prover Definitions
Test
Generation
Tests
isla-axiomatic
concurrency
tool
RMEM
concurrency
tool
ELF model
Lem
Concurrency models
Axiomatic, Cat
Concurrency models
Operational, Lem
Concurrent Execution

Sail ARMv8-A

Includes full ISA: Floating-point, address translation & page-table walks, synchronous exceptions, hypervisor mode, crypto instructions, vector instructions (NEON and SVE), memory partitioning and monitoring, pointer authentication, etc. . .

Such a complete authoritative architecture description not previously publicly available for formal reasoning

ARMv8.5-A Sail model now available (125 KLoS), and the generated prover definitions

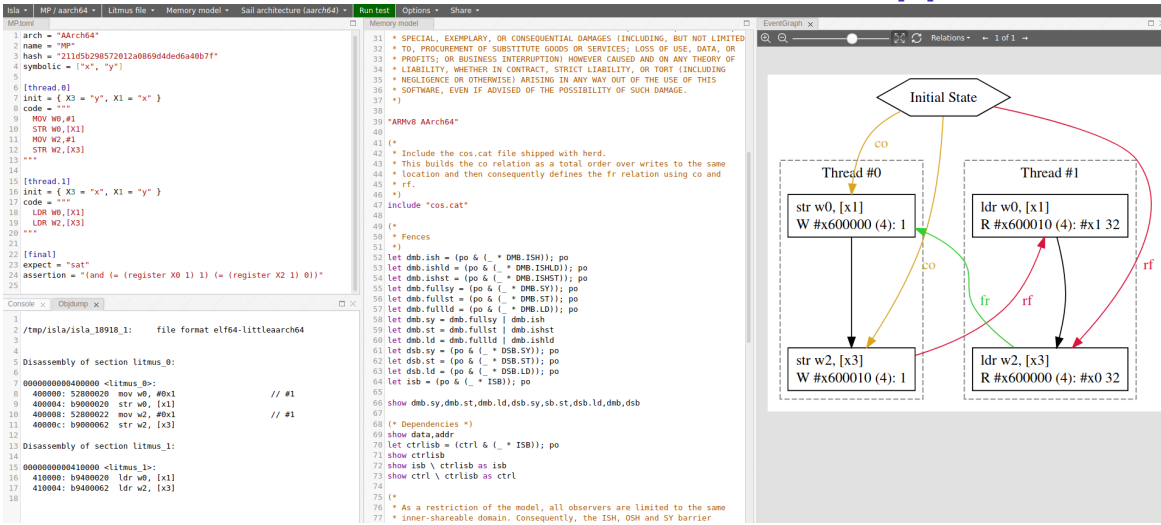
- Is it correct? Sail ARMv8.3-A tested on Arm-internal Architecture Validation Suite [Reid]; passed 99.85% of 15 400 tests as compared with Arm ASL. Boots Linux and Hafnium.
- Is it usable for sequential testing? Sail-generated v8.5-A emulator 200 KIPS
- Is it usable for proof? Proved characterisation of address translation, in Isabelle [Bauereiss] (also found some small bugs in ASL)

Sail RISC-V

Historically only text. We hand-wrote a Sail specification, now adopted by RISC-V International as the official formal model.

Integrating ISA and axiomatic models

Arm Concurrency: isla-axiomatic tool, for axiomatic models [?]



Armv8-A/RISC-V operational model

For more details, see [19, Pulte, Flur, Deacon, et al.] (especially its supplementary material https://www.cl.cam.ac.uk/~pes20/armv8-mca/), together with [20, 18, 17, 12, 8]

Together with the RISC-V manual:

- RISC-V: [34, Ch.14 RVWMO Memory Consistency Model, App.A RVWMO Explanatory Material, App.B Formal Memory Model Specifications]

As before: We have to understand *just enough* about hardware to explain and define the envelopes of programmer-visible behaviour that comprise the architectures.

As before: We have to understand *just enough* about hardware to explain and define the envelopes of programmer-visible behaviour that comprise the architectures.

x86

Programmers can assume instructions execute in program order, but with FIFO store buffer.

ARM, RISC-V, Power

By default, instructions can observably execute out-of-order and speculatively, except as forbidden by coherence, dependencies, barriers.

As with x86-TSO, structure the model into

- Thread semantics
- Storage/memory semantics

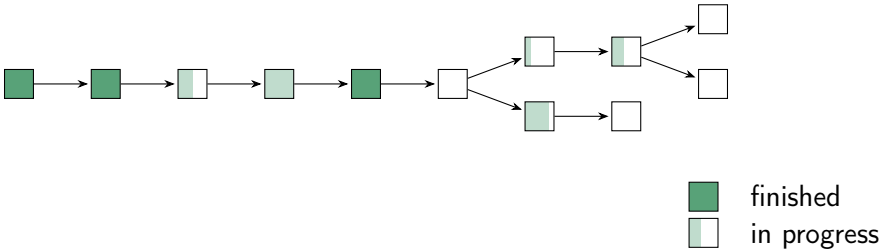
Model is integrated with Sail ISA semantics and executable in rmem.

Thread semantics: out-of-order, speculative execution abstractly

Our thread semantics has to account for out-of-order and speculative execution.

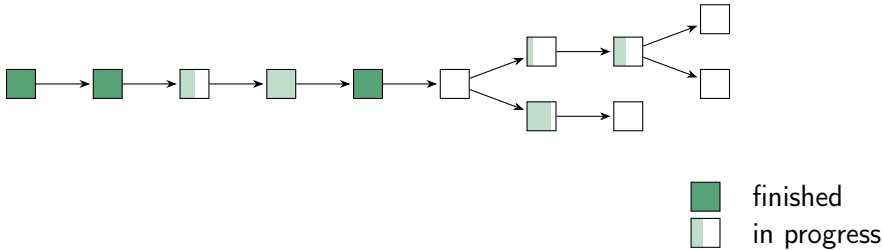
Thread semantics: out-of-order, speculative execution abstractly

Our thread semantics has to account for out-of-order and speculative execution.



Thread semantics: out-of-order, speculative execution abstractly

Our thread semantics has to account for out-of-order and speculative execution.



- instructions can be fetched before predecessors finished
- instructions independently make progress
- branch speculation allows fetching successors of branches
- multiple potential successors can be explored

NB actual hardware implementations can and do speculate even more, e.g. beyond strong barriers, so long as it is not observable

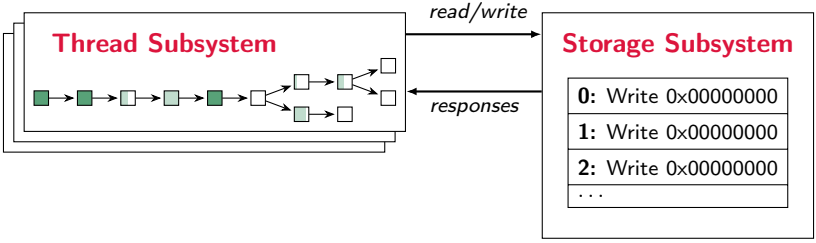
Memory/storage semantics

We could have an elaborate storage semantics, capturing caching effects of processors.

But it turns out, for Armv8 and RISC-V: *the observable relaxed behaviour is already explainable by an out-of-order (and speculative) thread semantics*.

Operational model

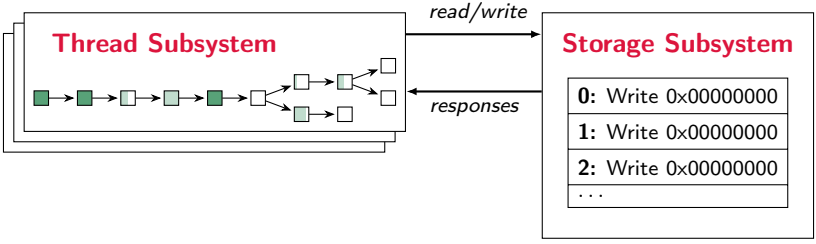
- each thread has a tree of instruction instances;
- *no register state*;
- threads execute in parallel above a flat memory state: mapping from addresses to write requests



(For now: plain memory reads, writes, strong barriers. All memory accesses same size.)

Operational model

- each thread has a tree of instruction instances;
- *no register state*;
- threads execute in parallel above a flat memory state:
  mapping from addresses to write requests
- for Power: need more complicated memory state to handle non-MCA

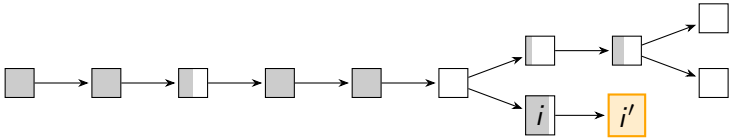


(For now: plain memory reads, writes, strong barriers. All memory accesses same size.)

Next: model transitions.

We will look at the Arm version of the model.
The RISC-V model is the same, except for model features not covered here.

Fetch instruction instance

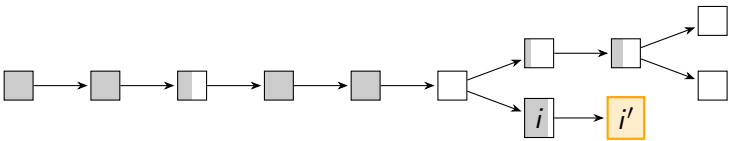


Condition:

A possible program-order successor i' of instruction instance i can be fetched from address *loc* and decoded if:

1. it has not already been fetched as successor of i
2. there is a decodable instruction in program memory at *loc*; and
3. *loc* is a possible next fetch address for i:
   - 3.1 for a non-branch/jump instruction, the successor instruction address ($i.program\_loc+4$);
   - 3.2 for an instruction that has performed a write to the program counter register (PC), the value that was written;
   - 3.3 for a conditional branch, either the successor address or the branch target address; or
   - 3.4 ....

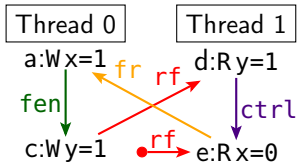
Fetch instruction instance



Action: construct a freshly initialised instruction instance i' for the instruction in program memory at *loc* and add i' to the thread's *instruction_tree* as a successor of i.

Example: speculative fetching

MP+fen+ctrl
(with "real" control dependency)



Allowed. The barrier orders the writes, but the control dependency is weak: *e* can be speculatively fetched and satisfied early (rmem web UI).

Instruction semantics (ignore the details)

How do instructions work?

Instruction semantics (ignore the details)

How do instructions work? Each instruction is specified as an imperative *Sail* program. For example:

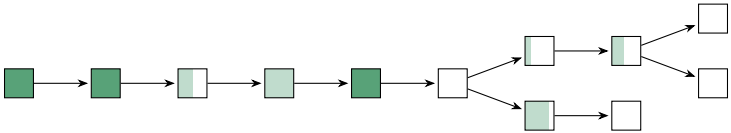
```
function clause execute ( LoadRegister(n,t,m,acctype,memop, ...) ) = {
  (bit[64]) offset := ExtendReg(m, extend_type, shift);
  (bit[64]) address := 0;
  (bit['D]) data := 0;              (* some local definitions *)
  ...
  if n == 31 then { ...  } else
    address := rX(n);               (* read the address register *)

  if ~(postindex) then              (* some bitvector arithmetic *)
    address := address + offset;

  if memop == MemOp_STORE then      (* announce the address *)
    wMem_Addr(address, datasize quot 8, acctype, false);
  ...

  switch memop {
    case MemOp_STORE -> {
      if rt_unknown then
  ...
```

Sail outcomes (ignore the details)

The Sail code communicates with the concurrency model via *outcomes*.

```
type outcome =
  | Done                                                    (* Sail execution ended *)
  | Internal of .. * outcome                                (* Sail internal step *)
  | Read_mem of read_kind * addr * size * (mem_val -> outcome)  (* read memory *)
  | Write_ea of write_kind * addr * size * outcome          (* announce write address *)
  | Write_memv of mem_val * outcome                         (* write memory *)
  | Read_reg of reg * (reg_val -> outcome)                  (* read register *)
  | Write_reg of reg * reg_val * outcome                    (* write register *)
  | Barrier of barrier_kind * outcome                       (* barrier effect *)
```

Instruction instance states



each instruction instance has:

- **`instruction_kind`**: load, store, barrier, branch, ...
- **status**: finished, committed (for stores), ...
- **`mem_reads`**, **`mem_writes`**: memory accesses so far
- **`reg_reads`**: register reads so far, including:
  `read_sources`, the instruction instances whose register write the read was from
- **`reg_writes`**: register writes so far, including:
  `write_deps`, the register reads the register write depended on
- **`regs_in`**, **`regs_out`**: the statically known register footprint
- ...
- **`pseudocode_state`**: the Sail state

Sail pseudocode states (ignore the details)

```
type pseudocode_state =
  | Plain of outcome
  | Pending_memory_read of read_continuation
  | Pending_memory_write of write_continuation
```

```
type outcome =
  | Done                                                  (* Sail execution ended *)
  | Internal of .. * outcome                              (* Sail internal step *)
  | Read_mem of read_kind * addr * size * (mem_val -> outcome)  (* read memory *)
  | Write_ea of write_kind * addr * size * outcome        (* announce write address *)
  | Write_memv of mem_val * outcome                       (* write memory *)
  | Read_reg of reg * (reg_val -> outcome)                (* read register *)
  | Write_reg of reg * reg_val * outcome                  (* write register *)
  | Barrier of barrier_kind * outcome                     (* barrier effect *)
```

In the following:

- (CO) coherence
- (BO) ordering from barriers
- (DO) ordering from dependencies

Instruction life cycle: barrier instructions

- fetch and decode
- commit barrier
- finish

Commit Barrier

Condition:

A barrier instruction i in state Plain $($Barrier$(barrier_kind, next_state'))$ can be committed if:

1. all po-previous conditional branch instructions are finished;
2. (BO) if i is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished.

Commit Barrier

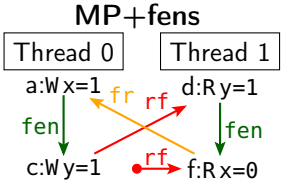
Action:

1. update the state of i to Plain $next\_state'$.

Barrier ordering

- so: a `dmb` barrier can only commit when all preceding memory accesses are finished
- a barrier commits before it finishes
- also (not seen yet): reads can only satisfy and writes can only propagate when preceding `dmb` barriers are finished

Barrier ordering



Forbidden. *c* can only propagate when the `dmb` is finished, the `dmb` can only finish when committed, and only commit when *a* is propagated; similarly, the `dmb` on Thread 1 forces *f* to satisfy after *d*.

Instruction life cycle: non-load/store/barrier instructions

for instance: `ADD`, branch, etc.

- fetch and decode
- register reads
- internal computation; just runs a Sail step (omitted)
- register writes
- finish

Register write

Condition:
An instruction instance i in state Plain $(\text{Write\_reg}(\textit{reg\_name}, \textit{reg\_value}, \textit{next\_state}'))$ can do the register write.

Register write

Action:

1. record *reg_name* with *reg_value* and *write_deps* in *i.reg_writes*; and
2. update the state of *i* to Plain *next_state*′.

where *write_deps* is the set of all *read_sources* from *i.reg_reads* . . .
write_deps: i.e. the sources all register reads the instruction has done so far

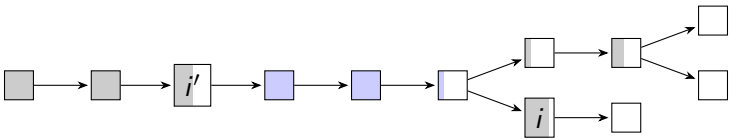
Register read

(remember: there is no ordinary register state in the thread state)

Condition:

An instruction instance i in state Plain *(*Read_reg*(reg_name, read_cont))* can do a register read if:

- (DO) the most recent preceding instruction instance i' that will write the register has done the expected register write.



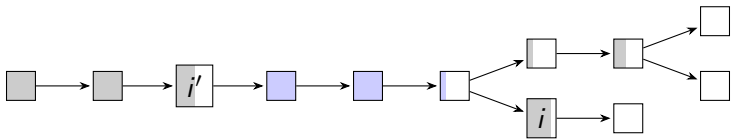
does not write *reg_name*

Register read

Let *read_source* be the write to *reg_name* by the most recent instruction instance i' that will write to the register, if any. If there is none, the source is the initial value. Let *reg_value* be its value.

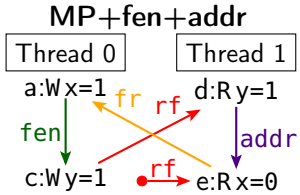
Action:

1. Record *reg_name*, *read_source*, and *reg_value* in *i.reg_reads*; and
2. update the state of *i* to Plain *(read_cont(reg_value))*.

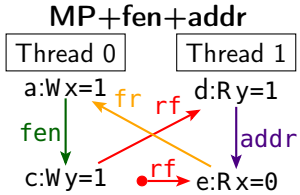


does not write *reg_name*

Example: address dependencies



Example: address dependencies



Forbidden. The barrier orders the writes, the address dependency prevents executing *e* before *d* (rmem web UI).

Instruction life cycle: loads

- fetch and decode
- register reads
- internal computation
- initiate read; when the address is available, constructs a read request (omitted)
- satisfy read
- complete load; hands the read value to the Sail execution (omitted)
- register writes
- finish

Satisfy read in memory

Condition:

A load instruction instance *i* in state Pending_mem_reads *read_cont* with unsatisfied read request *r* in *i.mem_reads* can satisfy *r* from memory if the read-request-condition predicate holds. This is if:

1. (BO) all po-previous `dmb sy` instructions are finished.

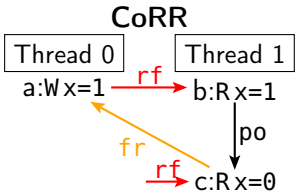
Satisfy read in memory

Let w be the write in memory to r's address. **Action:**

1. update r to indicate that it was satisfied by w; and
2. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e. for every non-finished po-successor instruction i' of i with a same-address read request r', if r' was satisfied from a write $w' \neq w$ that is not from a po-successor of i, restart i' and its data-flow dependents.

Let w be the write in memory to r's address. **Action:**

1. update r to indicate that it was satisfied by w; and
2. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e. for every non-finished po-successor instruction i' of i with a same-address read request r', if r' was satisfied from a write $w' \neq w$ that is not from a po-successor of i, restart i' and its data-flow dependents.



Think

- $r = b$, $r' = c$, $w = a$
- b is about to be satisfied by a
- c already satisfied from initial write

Forbidden. If c is satisfied from the initial write $x = 0$ before b is satisfied, once b reads from a it restarts c (rmem web UI).

Finish instruction

Condition:

A non-finished instruction i in state Plain *(*Done*)* can be finished if:

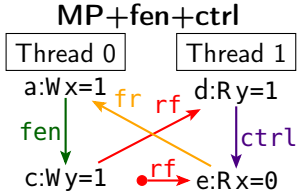
1. (CO) i has fully determined data;
2. all po-previous conditional branches are finished; and
3. if i is a load instruction:
   - 3.1 (BO) all po-previous `dmb sy` instructions are finished;
   - 3.2 (CO) it is guaranteed that the values read by the read requests of i will not cause coherence violations, i.e. ...

Finish instruction

Action:

1. record the instruction as finished, i.e., set *finished* to *true*; and
2. if *i* is a branch instruction, discard any untaken path of execution. I.e., remove any (non-finished) instructions that are not reachable by the branch taken in *instruction_tree*.

Example: finishing loads and discarding branches



Speculatively executing the load past the conditional branch does not allow finishing the load until the branch is determined. Finishing the branch discards untaken branches (rmem web UI).

Instruction life cycle: stores

- fetch and decode
- register reads and internal computation
- initiate write; when the address is available, constructs a write request without value (omitted)
- register reads and internal computation
- instantiate write; when the value is available, updates the write request's value (omitted)
- commit and propagate
- complete store; just resumes the Sail execution (omitted)
- finish

Commit and propagate store

Commit Condition:

For an uncommitted store instruction i in state Pending_mem_writes *write_cont*, i can commit if:

1. (CO) i has fully determined data (i.e., the register reads cannot change);
2. all po-previous conditional branch instructions are finished;
3. (BO) all po-previous `dmb sy` instructions are finished;
4. (CO) all po-previous memory access instructions have initiated and have a fully determined footprint

Propagate Condition:

For an instruction i in state Pending_mem_writes *write_cont* with unpropagated write, w in *i.mem_writes*, the write can be propagated if:

1. (CO) all memory writes of po-previous store instructions to the same address have already propagated
2. (CO) all read requests of po-previous load instructions to the same address have already been satisfied, and the load instruction is non-restartable.

Commit and propagate write

Commit Action: record i as committed.

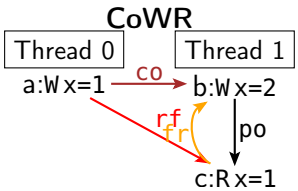
Propagate Action:

1. record w as propagated; and
2. update the memory with w; and
3. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e., for every non-finished instruction i' po-after i with read request r' that was satisfied from a write $w' \neq w$ to the same address, if w' is not from a po-successor of i,restart i' and its data-flow dependents.

Commit Action: record i as committed.

Propagate Action:

1. record w as propagated; and
2. update the memory with w; and
3. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e., for every non-finished instruction i' po-after i with read request r' that was satisfied from a write $w' \neq w$ to the same address, if w' is not from a po-successor of i,restart i' and its data-flow dependents.

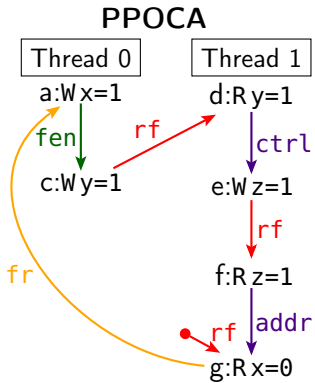


Think

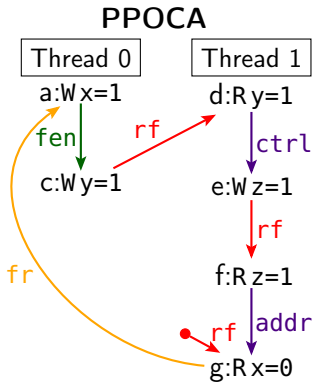
- $w = a$, $r' = b$, $w' = c$
- a is about to propagate
- b was already satisfied by c

Forbidden. If b is satisfied from c before a is propagated, a's propagation restarts b.

Write forwarding on a speculative branch



Write forwarding on a speculative branch



Allowed. But with just the previous rules we cannot explain this in the model.

Satisfy read by forwarding

Condition:

A load instruction instance *i* in state Pending_mem_reads *read_cont* with unsatisfied read request *r* in *i.mem_reads* can satisfy *r* by forwarding an unpropagated write by a program-order earlier store instruction instance, if the *read-request-condition* predicate holds. This is if:

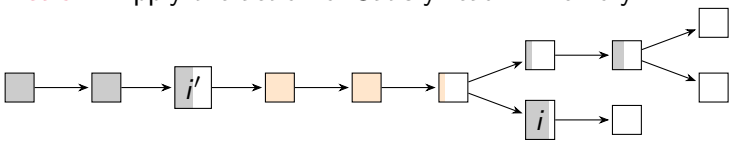
1. (BO) all po-previous `dmb sy` instructions are finished.

Satisfy read by forwarding

Let w be the most-recent write from a store instruction instance i' po-before i, to the address of r, and which is not superseded by an intervening store that has been propagated or read from by this thread. That last condition requires:

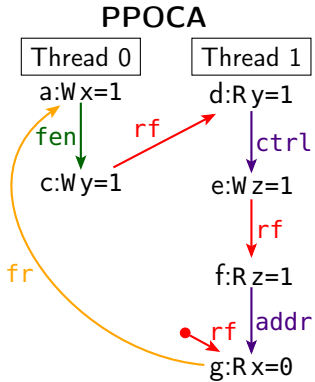
- (CO) that there is no store instruction po-between i and i' with a same-address write, and
- (CO) that there is no load instruction po-between i and i' that was satisfied by a same-address write from a different thread.

Action: Apply the action of Satisfy read in memory.



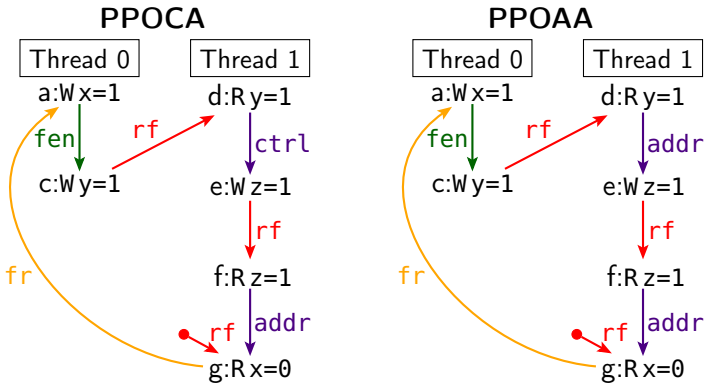
no same-address write or
same-address read from different thread

Write forwarding on a speculative branch



PPOCA allowed. (rmem web UI)

Write forwarding on a speculative branch



PPOCA allowed. (rmem web UI)
PPOAA forbidden.

Armv8-A/RISC-V axiomatic model

For more details, see [19, Pulte, Flur, Deacon, et al.] (especially its supplementary material https://www.cl.cam.ac.uk/~pes20/armv8-mca/), together with [15, 3].

Together with the vendor manuals:

- Arm: [32, Ch.B2 The AArch64 Application Level Memory Model]
- RISC-V: [34, Ch.8, "A" Standard Extension for Atomic Instructions, Ch.14 RVWMO Memory Consistency Model, App.A RVWMO Explanatory Material, App.B Formal Memory Model Specifications]

(Again) **By default, instructions can observably execute out-of-order and speculatively, except as forbidden by coherence, dependencies, barriers.**

Axiomatic model already allows "out-of-order" and speculative execution *by default* – everything is allowed unless ruled out by the axioms.

We will look at the Arm version of the model.
The RISC-V model is the same, except for model features not covered here.

Official axiomatic model

(without weaker barriers, release-/acquire-, and load-/store-exclusive instructions)

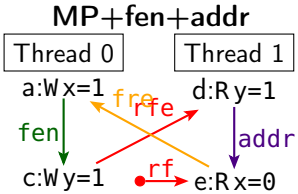
```
acyclic pos | fr | co | rf        (* coherence check *)

let obs = rfe | fre | coe         (* Observed-by *)

let dob = addr | data             (* Dependency-ordered-before *)
        | ctrl; [W]
        | addr; po; [W]
        | (ctrl | data); coi      (* Think 'coi' (globally equivalent) *)
        | (addr | data); rfi
        ...
let bob = po; [dmb.sy]; po        (* Barrier-ordered-before *)
          ...
let ob = obs | dob | aob | bob    (* Ordered-before *)

acyclic ob                        (* external check *)
```

Executable axiomatic models

Axiomatic model executable in:

- Herd [Alglave + Maranget]:
  `http://diy.inria.fr/doc/herd.html`
  `http://diy.inria.fr/www`
- Isla [Armstrong], with integrated Sail semantics:
  `https://isla-axiomatic.cl.cam.ac.uk/`

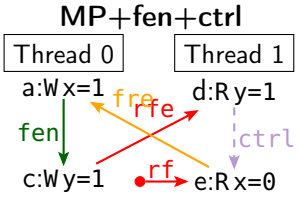
Example: address dependencies




```
acyclic pos | fr | co | rf

let obs = rfe | fre | coe

let dob = addr | data
        | ctrl; [W]
        | addr; po; [W]
        | (ctrl | data); coi
        | (addr | data); rfi
        ...
let bob = po; [dmb.sy]; po
          ...
let ob = obs | dob | aob | bob

acyclic ob
```

Forbidden. Each edge of the cycle is included in ob.

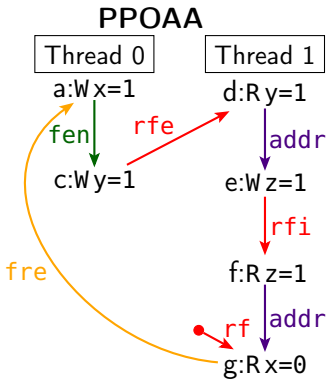
Example: speculative execution




```
acyclic pos | fr | co | rf

let obs = rfe | fre | coe

let dob = addr | data
        | ctrl; [W]
        | addr; po; [W]
        | (ctrl | data); coi
        | (addr | data); rfi
        ...
let bob = po; [dmb.sy]; po
          ...
let ob = obs | dob | aob | bob

acyclic ob
```

Allowed. The edges form a cycle, but `ctrl;[R]` to read events is not in ob

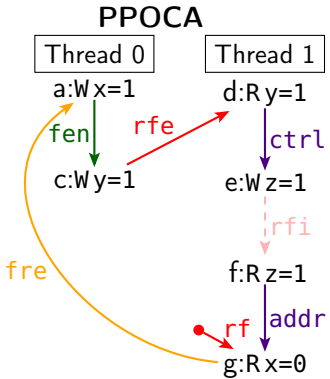
Write forwarding from an unknown-address write




```
acyclic pos | fr | co | rf

let obs = rfe | fre | coe

let dob = addr | data
        | ctrl; [W]
        | addr; po; [W]
        | (ctrl | data); coi
        | (addr | data); rfi
        ...
let bob = po; [dmb.sy]; po
          ...
let ob = obs | dob | aob | bob

acyclic ob
```

Forbidden. `ob` includes `addr;rfi`: forwarding is only possible when the address is determined

Write forwarding on a speculative path




```
acyclic pos | fr | co | rf

let obs = rfe | fre | coe

let dob = addr | data
        | ctrl; [W]
        | addr; po; [W]
        | (ctrl | data); coi
        | (addr | data); rfi
        ...
let bob = po; [dmb.sy]; po
          ...
let ob = obs | dob | aob | bob

acyclic ob
```

Allowed. Forwarding is allowed: `rfi` (and `ctrl;rfi` and `rfi;addr`) not in `ob` (compare x86-TSO)

Validation

lots...

Desirable properties of an architecture specification

1. Sound with respect to current hardware
2. Sound with respect to future hardware
3. Opaque with respect to hardware microarchitecture implementation detail
4. Complete with respect to hardware?
5. Strong enough for software
6. Unambiguous / precise
7. Executable as a test oracle
8. Incrementally executable
9. Clear
10. Authoritative?

Programming language concurrency

Introduction

For a higher-level programming language that provides some concurrent shared-memory abstraction, what semantics should (or can) it have?

For a higher-level programming language that provides some concurrent shared-memory abstraction, what semantics should (or can) it have?

NB: this is an *open problem*

Despite decades of research, we do not have a good semantics for any mainstream concurrent programming language that supports high-performance shared-memory concurrency.

(if you don't need high performance, you wouldn't be writing shared-memory concurrent code in the first place)

A general-purpose high-level language should provide a common abstraction over all those hardware architectures (and others).

A general-purpose high-level language should provide a common abstraction over all those hardware architectures (and others).

...that is efficiently implementable

A general-purpose high-level language should provide a common abstraction over all those hardware architectures (and others).

...that is efficiently implementable, w.r.t. both:

- the cost of providing whatever synchronisation the language-level model mandates above those various hardware models
- the impact of providing the language-level model on existing compiler optimisations

In other words...

At the language level, observable relaxed-memory behaviour arises from *the combination of:*

1. the hardware optimisations we saw before, and
2. a diverse collection of compiler optimisations,

both of which have been developed over many decades to optimise while preserving sequential behaviour, but which have substantial observable consequences for concurrent behaviour

Compiler optimisations routinely reorder, eliminate, introduce, split, and combine "normal" accesses, and remove or convert dependencies, in ways that vary between compilers, optimisation levels, and versions.

For example, in SC or x86, message passing should work as expected:

Thread 1	Thread 2
`x = 1` `y = 1`	`if (y == 1)` `  print x`

In SC, the program should only print nothing or 1, and an x86 assembly version will too (ARM/Power/RISC-V are more relaxed). What about Java/C/C++ etc.?

Compiler optimisations routinely reorder, eliminate, introduce, split, and combine "normal" accesses, and remove or convert dependencies, in ways that vary between compilers, optimisation levels, and versions.

For example, in SC or x86, message passing should work as expected:

Thread 1	Thread 2
`x = 1`	`int r1 = x`
`y = 1`	`if (y == 1)`
	`  print x`

In SC, the program should only print nothing or 1, and an x86 assembly version will too (ARM/Power/RISC-V are more relaxed). What about Java/C/C++ etc.?

If there's some other read of x in the context...

Compiler optimisations routinely reorder, eliminate, introduce, split, and combine "normal" accesses, and remove or convert dependencies, in ways that vary between compilers, optimisation levels, and versions.

For example, in SC or x86, message passing should work as expected:

Thread 1	Thread 2
`x = 1`	`int r1 = x`
`y = 1`	`if (y == 1)`
	`  print x`

In SC, the program should only print nothing or 1, and an x86 assembly version will too (ARM/Power/RISC-V are more relaxed). What about Java/C/C++ etc.?

If there's some other read of x in the context...
then common subexpression elimination can rewrite

```
print x    ⟹    print r1
```

Compiler optimisations routinely reorder, eliminate, introduce, split, and combine "normal" accesses, and remove or convert dependencies, in ways that vary between compilers, optimisation levels, and versions.

For example, in SC or x86, message passing should work as expected:

Thread 1	Thread 2
`x = 1`	`int r1 = x`
`y = 1`	`if (y == 1)`
	`  print r1`

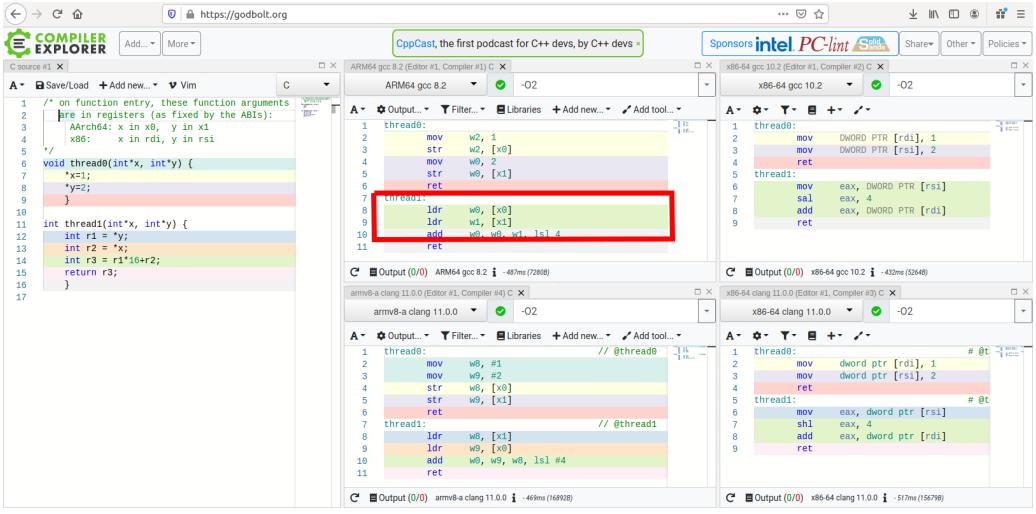
In SC, the program should only print nothing or 1, and an x86 assembly version will too (ARM/Power/RISC-V are more relaxed). What about Java/C/C++ etc.?

If there's some other read of x in the context...
then common subexpression elimination can rewrite

`print x` $\implies$ `print r1`

So the compiled program can print 0

Here ARM64 gcc 8.2 reorders the `thread1` loads, even without that control dependency.



Compiler Explorer (short link) (full link) NB: these are MP-shaped, but it's not legal C to run these in parallel!

Compiler analysis and transform passes

LLVM

Analysis passes

-aa-eval: Exhaustive Alias Analysis Precision Evaluator
-basic-aa: Basic Alias Analysis (stateless AA impl)
-basiccg: Basic CallGraph Construction
-count-aa: Count Alias Analysis Query Responses
-da: Dependence Analysis
-debug-aa: AA use debugger
-domfrontier: Dominance Frontier Construction
-domtree: Dominator Tree Construction
-dot-callgraph: Print Call Graph to "dot" file
-dot-cfg: Print CFG of function to "dot" file
-dot-cfg-only: Print CFG of function to "dot" file (with no function bodies)
-dot-dom: Print dominance tree of function to "dot" file
-dot-dom-only: Print dominance tree of function to "dot" file (with no function bodies)
-dot-postdom: Print postdominance tree of function to "dot" file
-dot-postdom-only: Print postdominance tree of function to "dot" file (with no function bodies)
-globalsmodref-aa: Simple mod/ref analysis for globals
-instcount: Counts the various types of Instructions
-intervals: Interval Partition Construction
-iv-users: Induction Variable Users
-lazy-value-info: Lazy Value Information Analysis
-libcall-aa: LibCall Alias Analysis
-lint: Statically lint-checks LLVM IR
-loops: Natural Loop Information
-memdep: Memory Dependence Analysis
-module-debuginfo: Decodes module-level debug info
-postdomfrontier: Post-Dominance Frontier Construction
-postdomtree: Post-Dominator Tree Construction
-print-alias-sets: Alias Set Printer
-print-callgraph: Print a call graph
-print-callgraph-sccs: Print SCCs of the Call Graph
-print-cfg-sccs: Print SCCs of each function CFG
-print-dom-info: Dominator Info Printer
-print-externalfnconstants: Print external fn callsites passed constants
-print-function: Print function to stderr
-print-module: Print module to stderr
-print-used-types: Find Used Types
-regions: Detect single entry single exit regions
-scalar-evolution: Scalar Evolution Analysis
-scev-aa: ScalarEvolution-based Alias Analysis
-stack-safety: Stack Safety Analysis
-targetdata: Target Data Layout

Transform passes

-adce: Aggressive Dead Code Elimination
-always-inline: Inliner for `always_inline` functions
-argpromotion: Promote 'by reference' arguments to scalars
-bb-vectorize: Basic-Block Vectorization
-block-placement: Profile Guided Basic Block Placement
-break-crit-edges: Break critical edges in CFG
-codegenprepare: Optimize for code generation
-constmerge: Merge Duplicate Global Constants
-dce: Dead Code Elimination
-deadargelim: Dead Argument Elimination
-deadtypeelim: Dead Type Elimination
-die: Dead Instruction Elimination
-dse: Dead Store Elimination
-function-attrs: Deduce function attributes
-globaldce: Dead Global Elimination
-globalopt: Global Variable Optimizer
-gvn: Global Value Numbering
-indvars: Canonicalize Induction Variables
-inline: Function Integration/Inlining
-instcombine: Combine redundant instructions
-aggressive-instcombine: Combine expression patterns
-internalize: Internalize Global Symbols
-ipsccp: Interprocedural Sparse Conditional Constant Propagation
-jump-threading: Jump Threading
-lcssa: Loop-Closed SSA Form Pass
-licm: Loop Invariant Code Motion
-loop-deletion: Delete dead loops
-loop-extract: Extract loops into new functions
-loop-extract-single: Extract at most one loop into a new function
-loop-reduce: Loop Strength Reduction
-loop-rotate: Rotate Loops
-loop-simplify: Canonicalize natural loops
-loop-unroll: Unroll loops
-loop-unroll-and-jam: Unroll and Jam loops
-loop-unswitch: Unswitch loops
-loweratomic: Lower atomic intrinsics to non-atomic form
-lowerinvoke: Lower invokes to calls, for unwindless code generators
-lowerswitch: Lower SwitchInsts to branches
-mem2reg: Promote Memory to Register
-memcpyopt: MemCpy Optimization
-mergefunc: Merge Functions
-mergereturn: Unify function exit nodes
-partial-inliner: Partial Inliner
-prune-eh: Remove unused exception handling info
-reassociate: Reassociate expressions
-reg2mem: Demote all values to stack slots
-sroa: Scalar Replacement of Aggregates
-sccp: Sparse Conditional Constant Propagation
-simplifycfg: Simplify the CFG
-sink: Code sinking
-strip: Strip all symbols from a module
-strip-dead-debug-info: Strip debug info for unused symbols
-strip-dead-prototypes: Strip Unused Function Prototypes
-strip-debug-declare: Strip all llvm.dbg.declare intrinsics
-strip-nondebug: Strip all symbols, except dbg symbols, from a module
-tailcallelim: Tail Call Elimination

GCC

IPA passes
IPA free lang data
IPA remove symbols
IPA OpenACC
IPA points-to analysis
IPA OpenACC kernels
Target clone
IPA auto profile
IPA tree profile
IPA free function summary
IPA increase alignment
IPA transactional memory
IPA lower emulated TLS
IPA whole program visibility
IPA profile
IPA identical code folding
IPA devirtualization
IPA constant propagation
IPA scalar replacement of aggregates
IPA constructor/destructor merge
IPA function summary
IPA inline
IPA pure/const analysis
IPA free function summary
IPA reference
IPA single use
IPA comdats
Materialize all clones
IPA points-to analysis
OpenMP simd clone
Tree SSA passes
Remove useless statements
OpenMP lowering
OpenMP expansion
Lower control flow
Lower exception handling control flow
Build the control flow graph
Find all referenced variables
Enter static single assignment form
Warn for uninitialized variables
Dead code elimination
Dominator optimizations
Forward propagation of single-use variables
Copy Renaming
PHI node optimizations
May-alias optimization
Profiling
Static profile estimation
Lower complex arithmetic
Scalar replacement of aggregates
Dead store elimination
Tail recursion elimination
Forward store motion
Partial redundancy elimination
Full redundancy elimination
Loop optimization
Loop invariant motion.
Canonical induction variable creation.
Induction variable optimizations.
Loop unswitching
Loop splitting
Vectorization
SLP Vectorization
Autoparallelization
Tree level if-conversion for vectorizer
Conditional constant propagation
Conditional copy propagation
Value range propagation
Folding built-in functions
Split critical edges
Control dependence dead code elimination
Tail call elimination
Warn for function return without value
Leave static single assignment form
Merge PHI nodes that feed into one another
Return value optimization
Return slot optimization
Optimize calls to `__builtin_object_size`
Loop invariant motion
Loop nest optimizations
Removal of empty loops
Unrolling of small loops
Predictive commoning
Array prefetching
Reassociation
Optimization of stdarg functions
RTL passes
Generation of exception landing pads
Control flow graph cleanup
Forward propagation of single-def values
Common subexpression elimination
Global common subexpression elimination
Loop optimization
Jump bypassing
If conversion
Web construction
Instruction combination
Mode switching optimization
Modulo scheduling
Instruction scheduling
Register allocation
The integrated register allocator (IRA)
Reloading
Basic block reordering
Variable tracking
Delayed branch scheduling
Branch shortening
Register-to-stack conversion
Final

Compiler analysis and transform passes

Hard to confidently characterise what all those syntactic transformations might do – and there are more, e.g. language implementations involving JIT compilation can use runtime knowledge of values.

But one can usefully view many, abstractly, as reordering, elimination, and introduction of memory reads and writes [40, Ševčík].

Defining PL Memory Models

Option 1: Don't. No Concurrency

Tempting... but poor match for current practice

Defining PL Memory Models

Option 2: Don't. No Shared Memory

A good match for *some* problems

(c.f. Erlang, MPI, ...)

Defining PL Memory Models

Option 3: sequential consistency (SC) everywhere

It's probably going to be expensive. Naively, one would have to:

- add strong barriers between *every* memory access, to prevent hardware reordering (or x86 LOCK'd accesses, Arm RCsc release/acquire pairs, etc.)
- disable all compiler optimisations that reorder, introduce, or eliminate accesses

(smarter: one could do analysis to approximate the thread-local or non-racy accesses, but aliasing always hard)

It's also not clear that SC is really more intuitive for real concurrent code than (e.g.) release/acquire-based models (c.f. Paul McKenney).

Defining PL Memory Models

Option 4: adopt a hardware-like model for the high-level language

If the aim is to enable implementations of language-level loads and stores with plain machine loads and stores, without additional synchronisation, the model would have to be as weak as any of the target hardware models.

But compiler optimisations do much more aggressive optimisations, based on deeper analysis, than hardware – so this would limit those.

Data races

All these hardware and compiler optimisations don't change the meaning of single-threaded code (any that do would be implementation bugs)

The interesting non-SC phenomena are only observable by code in which multiple threads are accessing the same data in conflicting ways (e.g. one writing and the other reading) without sufficient synchronisation between them – *data races*

(caution: the exact definition of what counts as a data race varies)

DRF-SC

Option 5: Use Data race freedom as a *definition*

Previously we had h/w models defining the allowed behaviour for arbitrary programs, and for x86-TSO had DRF as a *theorem* about some programs.

For a programming language, we could *define* a model by:

- programs that are race-free in SC semantics have SC behaviour
- programs that have a race in some execution in SC semantics can behave in any way at all

Kourosh Gharachorloo et al. [41, 42]; Sarita Adve & Mark Hill [43, 44]

DRF-SC

Option 5: Use Data race freedom as a *definition*

To implement: choose the high-level language synchronisation mechanisms, e.g. locks:

- prevent the compiler optimising across them
- ensure the implementations of the synchronisation mechanisms insert strong enough hardware synchronisation to recover SC in between (e.g. fences, x86 LOCK'd instructions, ARM "load-acquire"/"store-release" instructions,...)

DRF-SC

Option 5: Use Data race freedom as a *definition*

Pro:

- Simple!
- Only have to check race-freedom w.r.t. SC semantics
- Strong guarantees for most code
- Allows lots of freedom for compiler and hardware optimisations

"Programmer-Centric"

Option 5: Use Data race freedom as a *definition*

Con:

- programs that have a race in some execution in SC semantics
  can behave in any way at all
  - Undecidable premise.
  - Imagine debugging based on that definition. For any surprising behaviour, you have a disjunction: either bug is X ... or there is a potential race in *some* execution
  - No guarantees for untrusted code
    ...impact of that depends on the context
- restrictive. Forbids fancy high-performance concurrent algorithms
- need to define exactly what a race is

  what about races in synchronisation and concurrent datastructure libraries?

Java

Java (as of JSR-133): DRF-SC plus committing semantics

Option 6: Use Data race freedom as a definition, with committing semantics for safety

Java has integrated multithreading, and it attempts to specify the precise behaviour of concurrent programs.

By the year 2000, the initial specification was shown:

- to allow unexpected behaviours;
- to prohibit common compiler optimisations,
- to be challenging to implement on top of a weakly-consistent multiprocessor.

Superseded around 2004 by the JSR-133 memory model [45, Manson, Pugh, Adve]

Java (as of JSR-133): DRF-SC plus committing semantics

Option 6: Use Data race freedom as a definition, with committing semantics for safety

- Goal 1: data-race free programs are sequentially consistent;
- Goal 2: all programs satisfy some memory safety and security requirements;
- Goal 3: common compiler optimisations are sound.

Idea: an axiomatic model augmented with a committing semantics to enforce a causality restriction – there must exist an increasing sequence of subsets of the events satisfying various conditions. See [45, 46] for details.

Java (as of JSR-133): DRF-SC plus committing semantics

Option 6: Use Data race freedom as a definition, with committing semantics for safety

The model is intricate, and *fails to meet Goal 3*: Some optimisations may generate code that exhibits more behaviours than those allowed by the un-optimised source.

As an example, JSR-133 allows `r2=1` in the optimised code below, but forbids `r2=1` in the source code:

`x = y = 0`	
`r1=x`	`r2=y`
`y=r1`	`x=(r2==1)?y:1`

HotSpot optimisation $\longrightarrow$

`x = y = 0`	
`r1=x`	`x=1`
`y=r1`	**`r2=y`**

[46, Ševčík & Aspinall]

C/C++11

C/C++11: DRF-SC plus low-level atomics

Option 7: Use Data race freedom as a definition, extended with low-level atomics

C and C++ already require the programmer to avoid various *undefined behaviour* (UB), and give/impose no guarantees for programs that don't.

So DRF-SC is arguably a reasonable starting point

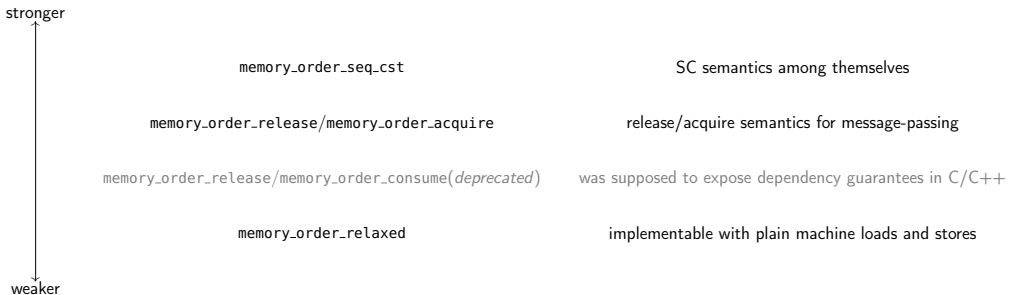
circa 2004 – 2011: effort by Boehm et al. in ISO WG21 C++ concurrency subgroup, adopted in C++11 and C11, to define a model based on DRF-SC but with *low-level atomics* to support high-performance concurrency

[47, Boehm & Adve]; `https://hboehm.info/c++mm/`; many ISO WG21 working papers Boehm, Adve, Sutter, Lea, McKenney, Saha, Manson, Pugh, Crowl, Nelson, ...

C/C++11 low-level atomics

Normal C/C++ accesses are deemed *non-atomic*, and any race on such (in any execution) gives rise to UB (NB: the whole program has UB, not just that execution)

Atomic accesses are labelled with a *"memory order"* (really a *strength*), and races are allowed.



C/C++11 low-level atomics

Normal C/C++ accesses are deemed *non-atomic*, and any race on such (in any execution) gives rise to UB (NB: the whole program has UB, not just that execution)

Atomic accesses are labelled with a *"memory order"* (really a *strength*), and races are allowed.

C concrete syntax – either:

- annotate the type, then all accesses default to SC atomics:
  ```
  _Atomic(Node *) top;
  ```
- or annotate the accesses with a memory order:
  ```
  t = atomic_load_explicit(&st->top, memory_order_acquire);
  ```

C++ concrete syntax – either:

- annotate the type and default to SC atomics, or
- annotate the accesses:
  ```
  x.store(v, memory_order_release)
  r = x.load(memory_order_acquire)
  ```

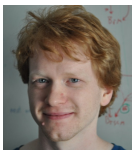
C/C++11 formalisation

WG21 worked initially just with prose definitions, and paper maths for a fragment

In 2009–2011 we worked with them to formalise the proposal:

- theorem-prover definitions in HOL4 and Isabelle/HOL
- executable-as-test-oracle versions that let us compute the behaviour of examples, in the cppmem tool `http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/`
  (now mostly superseded by Cerberus BMC [21, Lau et al.] `http://cerberus.cl.cam.ac.uk/bmc.html`)
- found and fixed various errors in the informal version
  (but not all – see later, and the web-page errata)
- achieved tight correspondence between eventual C++11 standard prose and our mathematical definitions

[7, 24, 11, 12, Batty et al.]



C/C++11 formalisation: Candidate executions

In an axiomatic style, broadly similar to axiomatic hardware models

Candidate pre-execution has events E and relations:

- **sb** sequenced-before (like po program order, but can be partial)
- **asw** additional synchronizes with (synchronisation from thread creation etc.)

Candidate execution witness:

- **rf** – reads-from
- **mo** – modification order (like co coherence, but over atomic writes only)
- **sc** – SC order (total order over all SC accesses)

C/C++11 formalisation: structure

For any program P, compute the set of candidate pre-executions that are consistent with the thread-local semantics (but with unconstrained memory read values)

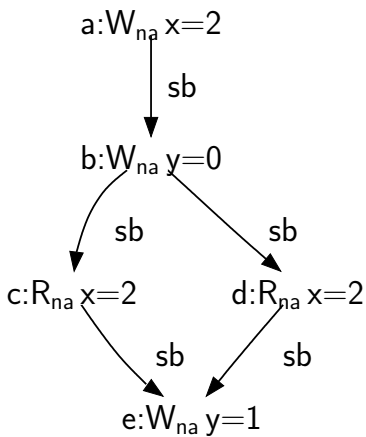
For each, enumerate all candidate execution witnesses, and take all of those that satisfy a *consistent execution* predicate

Check whether any consistent execution has a race. If so, P has undefined behaviour; otherwise, its semantics is the set of all those consistent executions.

Thanks to Mark Batty for the following slides

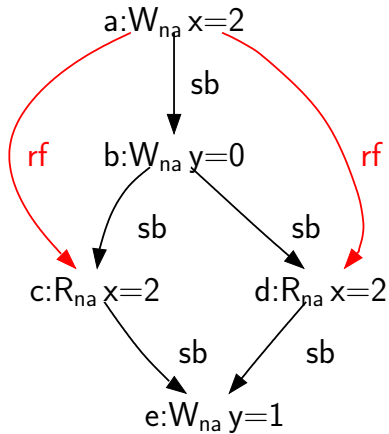
A single threaded program

```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0; }
```



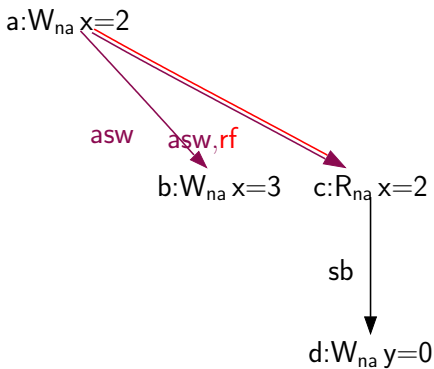
A single threaded program

```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0; }
```



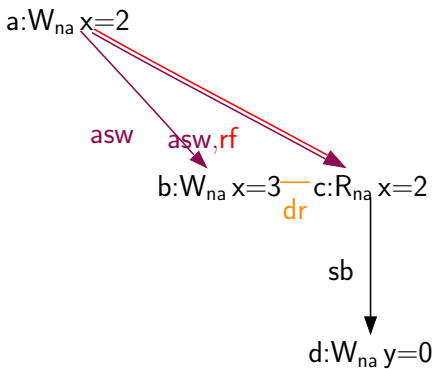
A data race

```
int y, x = 2;
x = 3;              | y = (x==3);
```



A data race

```
int y, x = 2;
x = 3;              | y = (x==3);
```



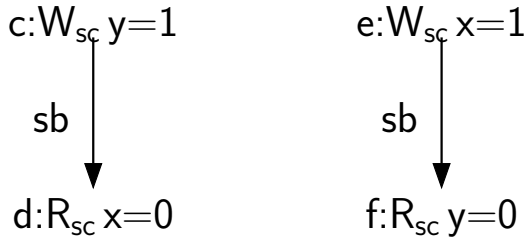
Simple concurrency: Decker's example and SC

```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  |  y.store(1, seq_cst);
y.load(seq_cst);      |  x.load(seq_cst);
```

Simple concurrency: Decker's example and SC

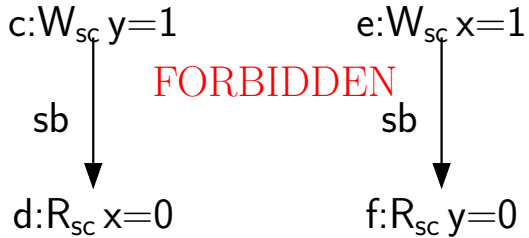
```
atomic_int x = 0;
atomic_int y = 0;
```

`x.store(1, seq_cst);`	`y.store(1, seq_cst);`
`y.load(seq_cst);`	`x.load(seq_cst);`



Simple concurrency: Decker's example and SC

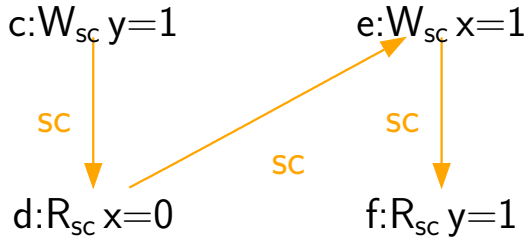
```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  |  y.store(1, seq_cst);
y.load(seq_cst);      |  x.load(seq_cst);
```



Simple concurrency: Decker's example and SC

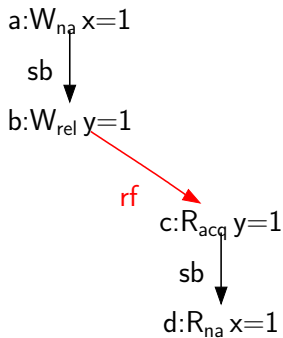
```
atomic_int x = 0;
atomic_int y = 0;
```

`x.store(1, seq_cst);`	`y.store(1, seq_cst);`
`y.load(seq_cst);`	`x.load(seq_cst);`



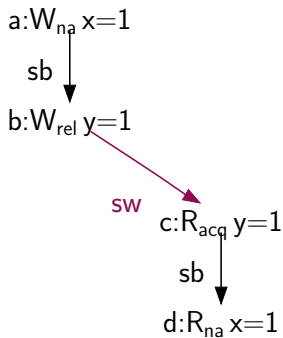
Expert concurrency: The release-acquire idiom

```
// sender                          // receiver
x = ...                            while (0 == y.load(acquire));
y.store(1, release);               r = x;
```



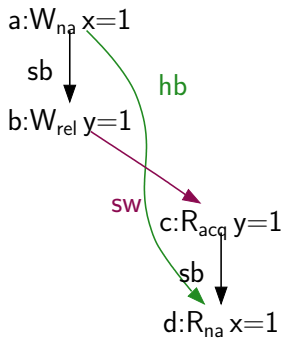
Expert concurrency: The release-acquire idiom

```
// sender                          // receiver
x = ...                            while (0 == y.load(acquire));
y.store(1, release);               r = x;
```



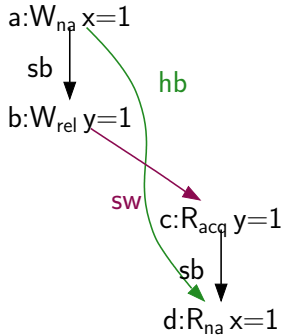
Expert concurrency: The release-acquire idiom

```
// sender                          // receiver
x = ...                            while (0 == y.load(acquire));
y.store(1, release);               r = x;
```



Expert concurrency: The release-acquire idiom

```
// sender                          // receiver
x = ...                            while (0 == y.load(acquire));
y.store(1, release);               r = x;
```



$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^{+}$$

Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

```
m.lock();
x = ...
m.unlock();
```

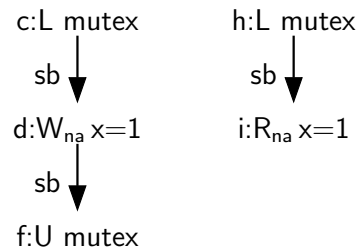
```
m.lock();
r = x;
```

Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

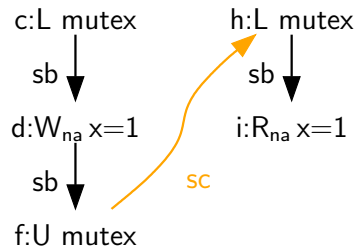


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

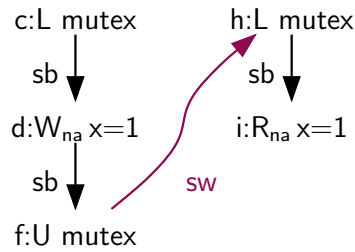


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

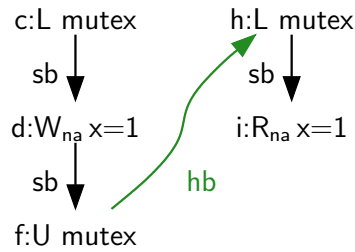


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

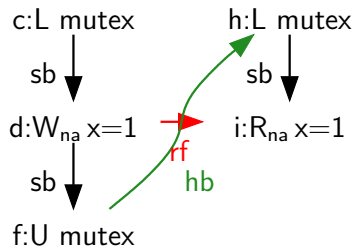


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	



Happens before is key to the model

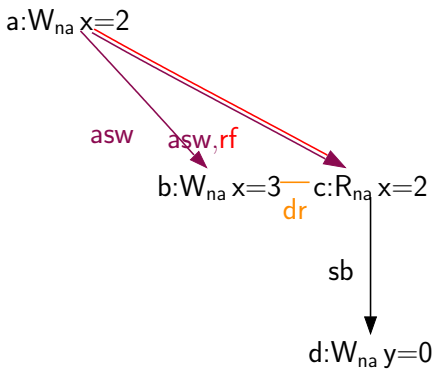
Non-atomic loads read the most recent write in happens before. (This is unique in DRF programs)

The story is more complex for atomics, as we shall see.

Data races are defined as an absence of happens before.

A data race

```
int y, x = 2;
x = 3;              | y = (x==3);
```



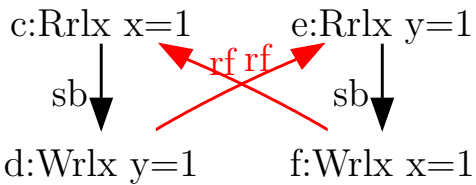
Data race definition

$$
\begin{aligned}
&\text{let } \mathit{data\_races}\ \mathit{actions}\ \mathit{hb} = \\
&\quad \{\ (a,\ b) \mid \forall\ a{\in}\mathit{actions}\ b{\in}\mathit{actions} \mid \\
&\qquad \neg\ (a = b)\ \wedge \\
&\qquad \mathrm{same\_location}\ a\ b\ \wedge \\
&\qquad (\mathrm{is\_write}\ a\ \vee\ \mathrm{is\_write}\ b)\ \wedge \\
&\qquad \neg\ (\mathrm{same\_thread}\ a\ b)\ \wedge \\
&\qquad \neg\ (\mathrm{is\_atomic\_action}\ a\ \wedge\ \mathrm{is\_atomic\_action}\ b)\ \wedge \\
&\qquad \neg\ ((a,\ b) \in \mathit{hb}\ \vee\ (b,\ a) \in \mathit{hb})\ \}
\end{aligned}
$$

A program with a data race has undefined behaviour.

Relaxed writes: load buffering

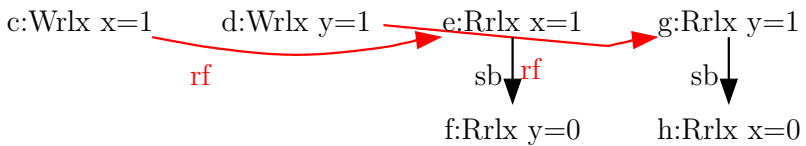
```
x.load(relaxed);          | y.load(relaxed);
y.store(1, relaxed);      | x.store(1, relaxed);
```



No synchronisation cost, but weakly ordered.

Relaxed writes: independent reads, independent writes

```
atomic_int x = 0;
atomic_int y = 0;
x.store(1, relaxed); | y.store(2, relaxed); | x.load(relaxed); | y.load(relaxed);
                                           | y.load(relaxed); | x.load(relaxed);
```



Expert concurrency: fences avoid excess synchronisation

```
// sender                    | // receiver
x = ...                      | while (0 == y.load(acquire));
y.store(1, release);         | r = x;
```

Expert concurrency: fences avoid excess synchronisation

```
// sender                  | // receiver
x = ...                    | while (0 == y.load(acquire));
y.store(1, release);       | r = x;
```

```
// sender                  | // receiver
x = ...                    | while (0 == y.load(relaxed));
y.store(1, release);       | fence(acquire);
                           | r = x;
```

Expert concurrency: The fenced release-acquire idiom

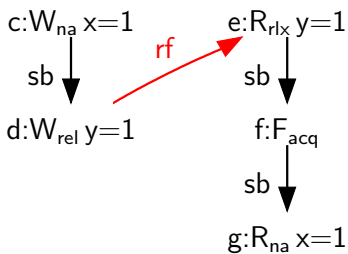
```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```

Expert concurrency: The fenced release-acquire idiom

```
// sender
x = ...
y.store(1, release);
```

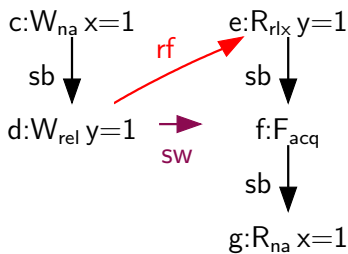
```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



Expert concurrency: The fenced release-acquire idiom

```
// sender
x = ...
y.store(1, release);
```

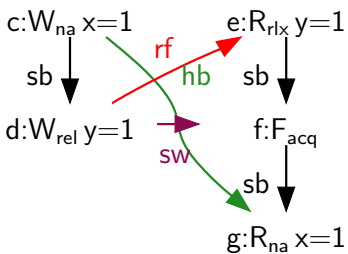
```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



Expert concurrency: The fenced release-acquire idiom

```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



Expert concurrency: modification order

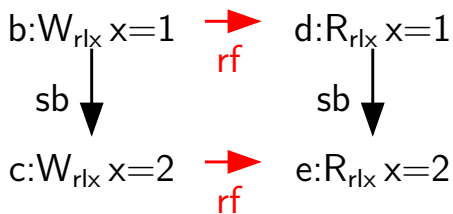
Modification order is a per-location total order over atomic writes of any memory order.

```
x.store(1, relaxed);      | x.load(relaxed);
x.store(2, relaxed);      | x.load(relaxed);
```

Expert concurrency: modification order

Modification order is a per-location total order over atomic writes of any memory order.

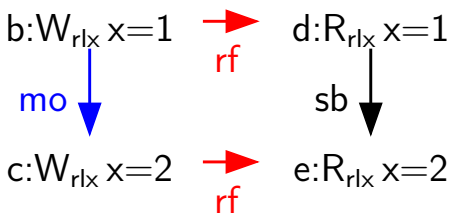
```
x.store(1, relaxed);      | x.load(relaxed);
x.store(2, relaxed);      | x.load(relaxed);
```



Expert concurrency: modification order

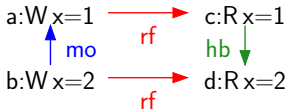
Modification order is a per-location total order over atomic writes of any memory order.

```
x.store(1, relaxed);      | x.load(relaxed);
x.store(2, relaxed);      | x.load(relaxed);
```

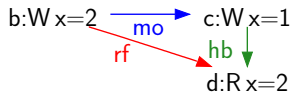


Coherence and atomic reads

All forbidden!

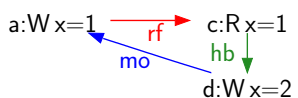


CoRR



CoWR

CoWW



CoRW

Atomics cannot read from later writes in happens before.

Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

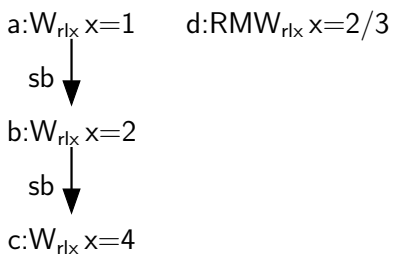
```
x.store(1, relaxed);  | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed);  |
x.store(4, relaxed);  |
```

Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed);
x.store(4, relaxed);
```

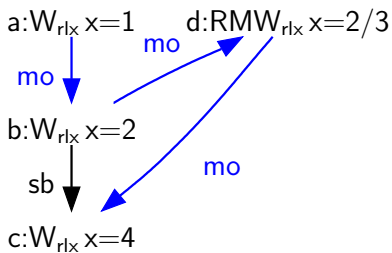


Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed);  | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed);  |
x.store(4, relaxed);  |
```

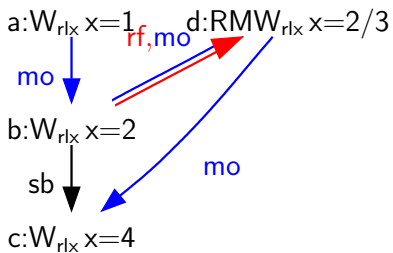


Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed);  | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed);  |
x.store(4, relaxed);  |
```



Very expert concurrency: consume

Weaker than acquire

Stronger than relaxed

Non-transitive happens before! (only fully transitive through data dependence, dd)

Consume

It turned out to be impractical to ensure that compilers preserve such data dependencies (which might go via compilation units that don't even use atomics)

The model as a whole

C1x and C++11 support many modes of programming:

- sequential

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest

The model as a whole

C1x and C++11 support many modes of programming:

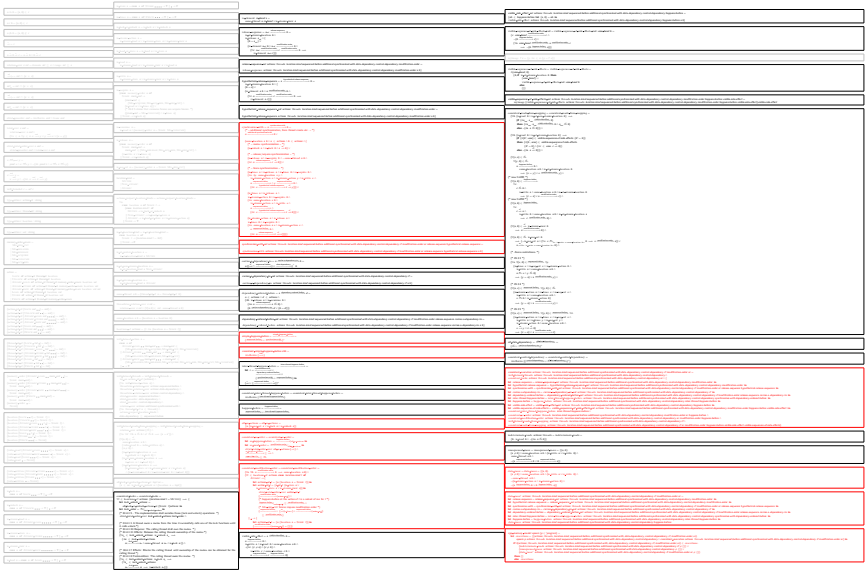
- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest
- with all of the above plus consume

C/C++11 models and tooling

The original formal model of [7, Batty et al.] is in executable typed higher-order logic, in Isabelle/HOL, from which we generated OCaml code to use in a checking tool.

This was later re-expressed in Lem [?], a typed specification language which can be translated into OCaml and multiple provers.

The full model

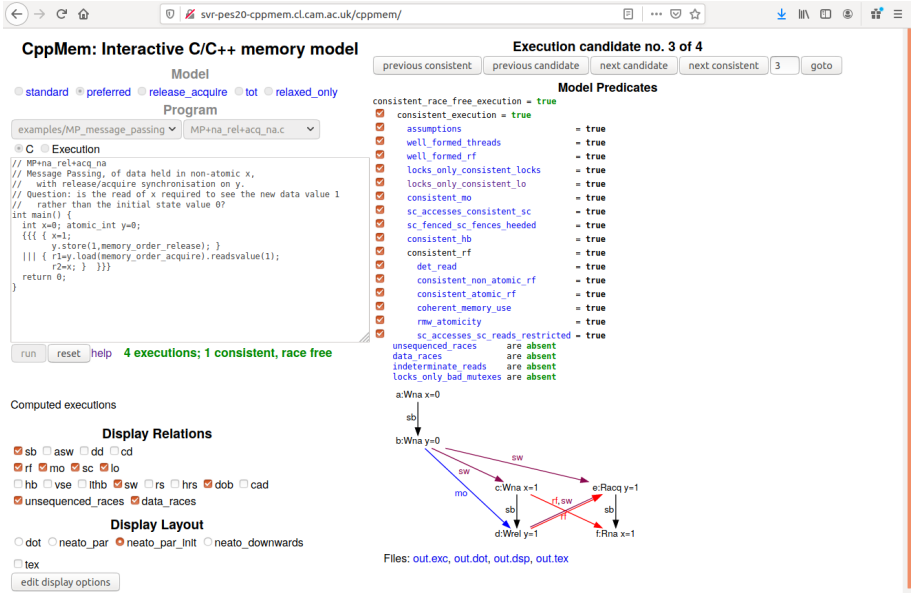


CppMem

CppMem: makes C/C+11 executable as a test oracle, and with a web interface for exploring candidate executions [Batty, Owens, Pichon-Pharabod, Sarkar, Sewell]

Enumerates candidate pre-executions for a small C-like language and applies the consistent-execution and race predicates to them.

http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/


svr-pes20-cppmem.cl.cam.ac.uk/cppmem/
CppMem: Interactive C/C++ memory model
Model
standard preferred release_acquire tot relaxed_only
Program
examples/MP_message_passing
MP+na_rel+acq_na.c
C Execution
// MP+na_rel+acq_na
// Message Passing, of data held in non-atomic x,
// with release/acquire synchronisation on y.
// Question: is the read of x required to see the new data value 1
// rather than the initial state value 0?
int main() {
  int x=0; atomic_int y=0;
  {{{ { x=1;
        y.store(1,memory_order_release); }
  ||| { r1=y.load(memory_order_acquire).readsvalue(1);
        r2=x; } }}}
  return 0;
}
run reset help 4 executions; 1 consistent, race free
Computed executions
Display Relations
sb asw dd cd
rf mo sc lo
hb vse lthb sw rs hrs dob cad
unsequenced_races data_races
Display Layout
dot neato_par neato_par_lnlt neato_downwards
tex
edit display options
Execution candidate no. 3 of 4
previous consistent previous candidate next candidate next consistent 3 goto
Model Predicates
consistent_race_free_execution = true
consistent_execution = true
assumptions = true
well_formed_threads = true
well_formed_rf = true
locks_only_consistent_locks = true
locks_only_consistent_lo = true
consistent_mo = true
sc_accesses_consistent_sc = true
sc_fenced_sc_fences_heeded = true
consistent_hb = true
consistent_rf = true
det_read = true
consistent_non_atomic_rf = true
consistent_atomic_rf = true
coherent_memory_use = true
rmw_atomicity = true
sc_accesses_sc_reads_restricted = true
unsequenced_races are absent
data_races are absent
indeterminate_reads are absent
locks_only_bad_mutexes are absent
a:Wna x=0
sb
b:Wna y=0
sw
sw
c:Wna x=1
e:Racq y=1
mo
sb
rf,sw
sb
rf
d:Wrel y=1
f:Rna x=1
Files: out.exc, out.dot, out.dsp, out.tex

C/C++11 and variants in `.cat`

Rephrased in relational algebra, in `.cat`, and improved in various ways:

- Overhauling SC atomics in C11 and OpenCL. Batty, Donaldson, Wickerson. [48]. Supplementary material: `http://multicore.doc.ic.ac.uk/overhauling/`

Usable in `herd`, for examples in a small C-like language

C11 cat from [48, Batty, Donaldson, Wickerson], adapted by Lau for [49]

```
// Modified from:
// https://github.com/herd/herdtools/tree/master/cats/c11/popl2016
// C11.cat w/o locks, consume
output addr
output data

let sb = po | I * (M \ I)
let mo = co

let cacq = [ACQ | (SC & (R | F)) | ACQ_REL]

let crel = [REL | (SC & (W | F)) | ACQ_REL]

let fr = rf_inv ; mo

let fsb = [F] ; sb
let sbf = sb ; [F]

//(* release_acquire_fenced_synchronizes_with,
//   hypothetical_release_sequence_set,
//   release_sequence_set *)

let rs_prime = int | (U * (R & W))
let rs = mo & (rs_prime \ ((mo \ rs_prime) ; mo))

let swra_head = crel ; fsb ? ; [A & W]
let swra_mid = [A & W] ; rs ? ; rf ; [R & A]
let swra_tail = [R & A] ; sbf ? ; cacq
let swra = (swra_head ; swra_mid ; swra_tail) & ext

let pp_asw = asw \ (asw ; sb)
let sw = pp_asw | swra

//(* happens_before,
//   inter_thread_happens_before,
//   consistent_hb *)
let ithbr = sw | (sw; sb)
let ithb_prime = (ithbr | (sb ; ithbr))
let ithb = ithb_prime+
let hb = sb | ithb
acyclic hb as hb_acyclic

//(* coherent_memory_use *)
let hbl = hb & loc

let coh_prime_head = rf_inv? ; mo
let coh_prime_tail = rf ? ; hb
let coh_prime = coh_prime_head ; coh_prime_tail

irreflexive coh_prime as coh_irreflexive

//(* visible_side_effect_set *)
let vis = ([W] ; hbl ; [R]) \ (hbl; [W]; hbl)

//(* consistent_atomic_rf *)
let rf_prime = rf ; hb
irreflexive rf_prime as rf_irreflexive

//(* consistent_non_atomic_rf  *)

let narf_prime = (rf ; nonatomicloc) \ vis
empty narf_prime as nrf_empty

let rmw_prime = rf | (mo ; mo ; rf_inv) | (mo ; rf)
irreflexive rmw_prime as rmw_irreflexive

//(* data_races *)
let cnf = ((W * U) | (U * W)) & loc
let dr = ext & (((cnf \ hb) \ (hb^-1)) \ (A * A))

//(* unsequenced_races *)
let ur = (((((W * M) | (M * W)) & int & loc) \ sb) \ sb^-1) \ id

let sc_clk_imm = [SC] ; (sc_clk \ (mo ; sc_clk))

let s1_prime = [SC]; sc_clk_imm ; hb
irreflexive s1_prime as s1

let s2_prime_head = [SC]; sc_clk ; fsb?
let s2_prime_tail = mo ; sbf?
let s2_prime = [SC]; s2_prime_head ; s2_prime_tail
irreflexive s2_prime as s2

let s3_prime_head = [SC]; sc_clk ; rf_inv ; [SC]
let s3_prime_tail = [SC] ; mo
let s3_prime = [SC]; s3_prime_head ; s3_prime_tail
irreflexive s3_prime as s3

let s4_prime = [SC]; sc_clk_imm ; rf_inv ; hbl ; [W]
irreflexive s4_prime as s4

let s5_prime = [SC]; sc_clk ; fsb ; fr
irreflexive s5_prime as s5

let s6_prime = [SC]; sc_clk ; fr ; sbf
irreflexive s6_prime as s6

let s7_prime_head = [SC]; sc_clk ; fsb
let s7_prime_tail = fr ; sbf
let s7_prime = [SC]; s7_prime_head ; s7_prime_tail
irreflexive s7_prime as s7

let ___bmc_hb = hb

undefined_unless empty dr as dr_ub
undefined_unless empty ur as unsequenced_race
```

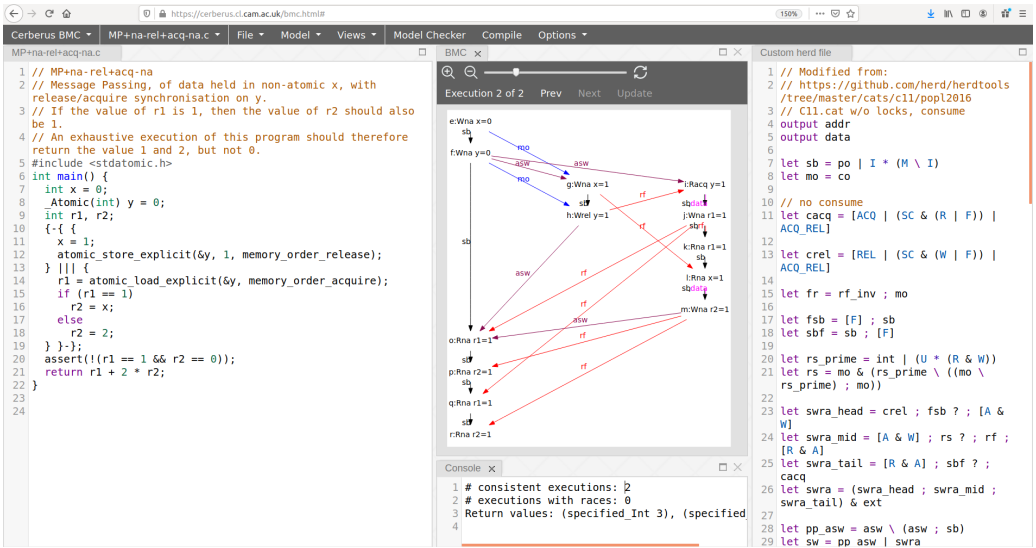
Cerberus BMC

- Cerberus-BMC: a Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. Lau, Gomes, Memarian, Pichon-Pharabod, Sewell. [49]

Integrates the Cerberus semantics for a substantial part of C [**?**, 50, Memarian et al.] with arbitrary concurrency semantics expressed in `.cat` relational style.

Translates both the C semantics and the concurrency model into SMT constraints.

`https://cerberus.cl.cam.ac.uk/bmc.html`


https://cerberus.cl.cam.ac.uk/bmc.html#
150%
Cerberus BMC
MP+na-rel+acq-na.c
File
Model
Views
Model Checker
Compile
Options
MP+na-rel+acq-na.c
1 // MP+na-rel+acq-na
2 // Message Passing, of data held in non-atomic x, with release/acquire synchronisation on y.
3 // If the value of r1 is 1, then the value of r2 should also be 1.
4 // An exhaustive execution of this program should therefore return the value 1 and 2, but not 0.
5 #include <stdatomic.h>
6 int main() {
7 int x = 0;
8 _Atomic(int) y = 0;
9 int r1, r2;
10 {-{ {
11 x = 1;
12 atomic_store_explicit(&y, 1, memory_order_release);
13 } ||| {
14 r1 = atomic_load_explicit(&y, memory_order_acquire);
15 if (r1 == 1)
16 r2 = x;
17 else
18 r2 = 2;
19 } }-};
20 assert(!(r1 == 1 && r2 == 0));
21 return r1 + 2 * r2;
22 }
23
24
BMC
Execution 2 of 2
Prev
Next
Update
e:Wna x=0
sb
f:Wna y=0
mo
asw
mo
asw
g:Wna x=1
i:Racq y=1
sb
sbasw
rf
h:Wrel y=1
j:Wna r1=1
sbrf
rf
sb
k:Rna r1=1
sb
asw
rf
l:Rna x=1
sbdata
rf
m:Wna r2=1
asw
rf
o:Rna r1=1
sb
rf
p:Rna r2=1
sb
q:Rna r1=1
sb
r:Rna r2=1
Console
1 # consistent executions: 2
2 # executions with races: 0
3 Return values: (specified_Int 3), (specified
4
Custom herd file
1 // Modified from:
2 // https://github.com/herd/herdtools/tree/master/cats/c11/popl2016
3 // C11.cat w/o locks, consume
4 output addr
5 output data
6
7 let sb = po | I * (M \ I)
8 let mo = co
9
10 // no consume
11 let cacq = [ACQ | (SC & (R | F))] | ACQ_REL]
12
13 let crel = [REL | (SC & (W | F))] | ACQ_REL]
14
15 let fr = rf_inv ; mo
16
17 let fsb = [F] ; sb
18 let sbf = sb ; [F]
19
20 let rs_prime = int | (U * (R & W))
21 let rs = mo & (rs_prime \ ((mo \ rs_prime) ; mo))
22
23 let swra_head = crel ; fsb ? ; [A & W]
24 let swra_mid = [A & W] ; rs ? ; rf ; [R & A]
25 let swra_tail = [R & A] ; sbf ? ; cacq
26 let swra = (swra_head ; swra_mid ; swra_tail) & ext
27
28 let pp_asw = asw \ (asw ; sb)
29 let sw = pp_asw | swra

RC11 .cat adapted for C++20 changes [51, 52] by Lahav, Vafeiadis (untested)

```
// RC11 .cat file without fences
// adpated for the changes that were approved for C++20
output addr
output data

let sb = po | I * (M \ I)
let rfstar = rf*
let rs = [W & ~NA] ; rfstar

//let sw = [REL | ACQ_REL | SC] ; ([F] ; sb)? ; rs ; rf ; [R & ~NA] ; (sb ; [F])? ; [ACQ | ACQ_REL | SC]

let sw_prime = [REL | ACQ_REL | SC] ; rs ; rf ; [R & ~NA & (ACQ | ACQ_REL | SC)]
let sw = sw_prime | asw
let hb = (sb | sw)+

let mo = co

let fr = (rf_inv ; mo) \ id
let eco = rf | mo | fr | mo ; rf | fr ; rf

irreflexive (hb ; eco) as coh

irreflexive eco as atomic1
irreflexive (fr ; mo) as atomic2

let fhb = [F & SC] ; hb?
let hbf = hb? ; [F & SC]
let scb =  sb | sb ; hb ; sb | hb & loc | mo | fr
let psc_base =  ([SC] | fhb) ; scb ; ([SC] | hbf)
let psc_f =  [F & SC] ; (hb | hb; eco; hb) ; [F & SC]
let psc =  psc_base | psc_f
acyclic psc as sc

let conflict = (((W * U) | (U * W)) & loc)
let race = ext & (((conflict \ hb) \ (hb^-1)) \ (A * A))

let __bmc_hb = hb

undefined_unless empty race as racy
```

Mappings from C/C++11 to hardware

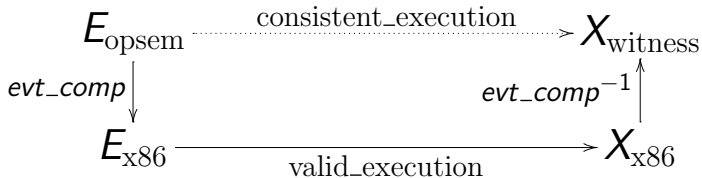
Implementability

Can we compile to x86?

Operation	x86 Implementation
load(non-seq_cst)	mov
load(seq_cst)	lock xadd(0)
store(non-seq_cst)	mov
store(seq_cst)	lock xchg
fence(non-seq_cst)	no-op

x86-TSO is stronger and simpler.

Theorem



We have a mechanised proof that C1x/C++11 behaviour is preserved.

Can we compile to Power? To ARMv7? To Armv8-A?

Mappings from C/C++11 operations to x86, Power, ARMv7, Itanium originally developed by C++11 contributors

Supposed paper proof for Power [11], but flawed – see errata (thanks to Lahav et al. and Manerkar et al.)

More recent mechanised proofs for fragments of C11 and variants by [53, Podkopaev, Lahav, Vafeiadis]

Mappings

Compilation from C/C++11 involves mapping each synchronisation operation to hardware *and* restricting compiler optimisations across these.

C/C++11 operation	x86	Armv8-A AArch64	Power	RISC-V
Load Relaxed	`mov`	`ldr`	`ld`	
Store Relaxed	`mov`	`str`	`st`	
Load Acquire	`mov`	`ldar`[2]	`ld;cmp;bc;isync`	
Store Release	`mov`	`stlr`	`lwsync;st`	
Load Seq_Cst	`mov`	`ldar`[3]	`sync;ld;cmp;bc;isync`[4]	
Store Seq_Cst	`xchg`[1]	`stlr`[3]	`sync;st`[4]	
Acquire fence	*nothing*	`dmb ld`	`lwsync`	
Release fence	*nothing*	`dmb`	`lwsync`	
Acq_Rel fence	*nothing*	`dmb`	`lwsync`	
Seq_Cst fence	`mfence`	`dmb`	`hwsync`	

[1] `xchg` is implicitly LOCK'd

[2] or `ldarp` for Armv8.3 or later?

[3] note that Armv8-A store-release and load-acquire are strong enough for SC atomics (developed for those)

[4] for Power this is the *leading sync* mapping. Note how it puts a `sync` between each pair of SC accesses

Note that the mapping has to be part of the ABI: e.g. one can't mix (by linking) a leading and trailing sync mapping

C/C++11 operational model

proved equivalent to that axiomatic model, in Isabelle [**?**, Nienhuis et al.]

C/C++11 after 2011

- Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. Batty, Memarian, Owens, Sarkar, Sewell. [11]
- Synchronising C/C++ and POWER. Sarkar, Memarian, Owens, Batty, Sewell, Maranget, Alglave, Williams. [12]
- Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. Morisset, Pawan, Zappa Nardelli. [?]
- Outlawing ghosts: avoiding out-of-thin-air results. Boehm, Demsky. [54]
- The Problem of Programming Language Concurrency Semantics. Batty, Memarian, Nienhuis, Pichon-Pharabod, Sewell. [?]
- Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. Vafeiadis, Balabonski, Chakraborty, Morisset, Zappa Nardelli. [?]
- Overhauling SC atomics in C11 and OpenCL. Batty, Donaldson, Wickerson. [48]
- An operational semantics for C/C++11 concurrency. Nienhuis, Memarian, Sewell. [?]
- Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. Manerkar, Trippel, Lustig, Pellauer, Martonosi. [55]
- Repairing sequential consistency in C/C++11. Lahav, Vafeiadis, Kang, Hur, Dreyer. [?]
- Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. Flur, Sarkar, Pulte, Nienhuis, Maranget, Gray, Sezgin, Batty, Sewell. [18]
- Bridging the gap between programming languages and hardware weak memory models. Podkopaev, Lahav, Vafeiadis. [53]
- Cerberus-BMC: a Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. Lau, Gomes, Memarian, Pichon-Pharabod, Sewell. [49]
- P0668R5: Revising the C++ memory model. Boehm, Giroux, Vafeiadis. [51]
- P0982R1: Weaken Release Sequences. Boehm, Giroux, Vafeiadis. [52]
- ...and more

...the last two in C++20

The thin-air problem

The thin-air problem

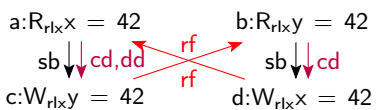
The C/C++11 concurrency model (with later modifications) is, as far as is known, sound w.r.t. existing compiler and hardware optimisations

But... for relaxed atomics, it admits undesirable executions where values seem to appear out of thin air, as noted at the time [**?**, 23.9p9]:

> *[Note: The requirements do allow r1 == r2 == 42 in the following example, with x and y initially zero:*

```
r1 = load_rlx(x);  ||  r2 = load_rlx(y);
if (r1 == 42)      ||  if (r2 == 42)
  store_rlx(y,r1)  ||    store_rlx(x,42)
```

LB+ctrldata+ctrl-single



> *However, implementations should not allow such behavior. – end note]*

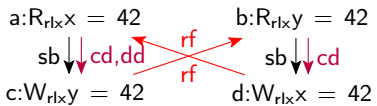
Using condensed syntax for brevity, not actual C++11. On the right cd and dd indicate control and data dependencies.

The thin-air problem

[Note: The requirements do allow r1 == r2 == 42 in the following example, with x and y initially zero:

LB+ctrldata+ctrl-single

```
r1 = load_rlx(x);  ||  r2 = load_rlx(y);
if (r1 == 42)      ||  if (r2 == 42)
  store_rlx(y,r1)  ||    store_rlx(x,42)
```



However, implementations should not allow such behavior. – end note]

There is no precise definition of what thin-air behaviour is—if there were, it could simply be forbidden by fiat, and the problem would be solved. Rather, there are a few known litmus tests (like the one above) where certain outcomes are undesirable and do not appear in practice (as the result of hardware and compiler optimisations). The problem is to draw a fine line between those undesirable outcomes and other very similar litmus tests which important optimisations do exhibit and which therefore must be admitted.

The thin-air problem

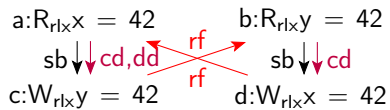
Batty et al. [**?**] observe that this cannot be solved with any per-candidate-execution model that uses the C/C++11 notion of candidate execution. Consider:

LB+ctrldata+ctrl-double

`r1 = load`$_{rlx}$`(x);`	`r2 = load`$_{rlx}$`(y);`
`if (r1 == 42)`	`if (r2 == 42)`
`  store`$_{rlx}$`(y,r1)`	`  store`$_{rlx}$`(x,42)`
	`else`
	`  store`$_{rlx}$`(x,42)`

Compilers will optimise the second thread's conditional, removing the control dependency, to:

`r1 = load`$_{rlx}$`(x);`	`r2 = load`$_{rlx}$`(y);`
`if (r1 == 42)`	`store`$_{rlx}$`(x,42)`
`  store`$_{rlx}$`(y,r1)`	



then compiler or hardware reordering of the second thread will make this observable in practice, so it has to be allowed.

But this is exactly the same candidate execution as that of LB+ctrldata+ctrl-single, which we want to forbid.

The thin-air problem

Basic issue: compiler analysis and optimisation passes examine and act on the program text, incorporating information from multiple executions

The thin-air problem

Possible approaches

- **Option 8a:** A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. Pichon-Pharabod, Sewell. [**?**]
- **Option 8b:** Explaining Relaxed Memory Models with Program Transformations. Lahav, Vafeiadis. [56]
- **Option 8c:** forbid load-to-store reordering, making rf $\cup$ sb acyclic [57, 54, **?**, **?**]
- **Option 8d:** Promising 2.0: global optimizations in relaxed memory concurrency. Lee, Cho, Podkopaev, Chakraborty, Hur, Lahav, Vafeiadis [58]
- **Option 8e:** Modular Relaxed Dependencies in Weak Memory Concurrency. Paviotti, Cooksey, Paradis, Wright, Owens, Batty. [59]
- **Option 8f:** Pomsets with Preconditions: A Simple Model of Relaxed Memory. Jagadeesan, Jeffrey, Riely [60]
- ...? See talk by Boehm and McKenney

Other languages

Defining PL Memory Models

Option 9: DRF-SC, but exclude races statically

By typing? Rust.

But not expressive enough for high-performance concurrent code, which needs unsafe blocks.

See RustBelt `https://plv.mpi-sws.org/rustbelt/#project` (Dreyer, Jung, et al.) for ongoing research on how to verify those

Option 10: Axiomatic model for Linux kernel concurrency primitives

Linux uses its own primitives, not C11: `READ_ONCE`, `WRITE_ONCE`, `smp_load_acquire()`, `smp_mb()`, ...

Axiomatic model for these:

- Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. Alglave, Maranget, McKenney, Parri, Stern. [61]

aiming to capture the intent (including RCU) – but it relies on dependencies. Those in use are believed/hoped to be preserved by compilers, but in general they are not, so this is not sound in general w.rt. compiler optimisations

GPU concurrency

- GPU Concurrency: Weak Behaviours and Programming Assumptions. Alglave, Batty, Donaldson, Gopalakrishnan, Ketema, Poetzl, Sorensen, Wickerson. [62]
- Remote-scope promotion: clarified, rectified, and verified. Wickerson, Batty, Beckmann, Donaldson. [63]
- Overhauling SC atomics in C11 and OpenCL. Batty, Donaldson, Wickerson. [48].
- Exposing errors related to weak memory in GPU applications. Sorensen, Donaldson. [?]
- Portable inter-workgroup barrier synchronisation for GPUs. Sorensen, Donaldson, Batty, Gopalakrishnan, Rakamaric. [64]

Option 11: broadly follow C/C++11
aim: DRF-SC model, with defined semantics for data-races (no thin-air), in a per-candidate-execution model, with the same compilation scheme as C/C++...

...tricky. And other issues, as discussed in:

- Repairing and mechanising the JavaScript relaxed memory model. Watt, Pulte, Podkopaev, Barbier, Dolan, Flur, Pichon-Pharabod, Guo. [65]
- Weakening WebAssembly. Watt, Rossberg, Pichon-Pharabod. [66]

Multicore OCaml

"local data race freedom"

- Bounding data races in space and time. Dolan, Sivaramakrishnan, Madhavapeddy. [67]

Conclusion

Taking stock

In 2008, all this was pretty mysterious. Now:

Hardware models

- "user" fragment – what you need for concurrent algorithms. In pretty good shape, for all these major architectures (albeit still some gaps, and we don't yet have full integration of ISA+concurrrency in theorem provers)
- "system" fragment – what you need in addition for OS kernels and hypervisors: instruction fetch, exceptions, virtual memory. Ongoing – e.g. [22, Simner et al.] for Armv8-A self-modifying code and cache maintenance.

Programming language models

- remains an open problem: C/C++ not bad, but thin-air is a big problem for reasoning about code that uses relaxed atomics in arbitrary ways

Verification techniques

- lots of ongoing work on proof-based verification and model-checking above the models, that we've not had time to cover

Overall: a big success for rigorous semantics inspired by, applied to, and impacting mainstream systems

Appendix: Selected Experimental Results

x86 Experimental Results

AArch64 Experimental Results

Power Experimental Results

RISC-V Experimental Results

References

NB: this is by no means a complete bibliography of all the relevant work – it's just the material that the course is most closely based on, and doesn't cover all the previous related work that that built on, or other parallel and recent developments.

[1] The Semantics of x86-CC Multiprocessor Machine Code.
S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave.
In *Proc. POPL 2009*.

[2] The Semantics of Power and ARM Multiprocessor Machine Code.
J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli.
In *Proc. DAMP 2009*.

[3] Fences in Weak Memory Models.
J. Alglave, L. Maranget, S. Sarkar, and P. Sewell.
In *Proc. CAV*.

[4] A better x86 memory model: x86-TSO.
S. Owens, S. Sarkar, and P. Sewell.
In *Proc. TPHOLs*.

[5] x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors.
P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen.
Communications of the ACM, 53(7):89–97, July 2010.

[6] Reasoning about the Implementation of Concurrency Abstractions on x86-TSO.
Scott Owens.
In *ECOOP 2010: Proceedings of the 24th European Conference on Object-Oriented Programming*.
[url].

[7] Mathematizing C++ Concurrency.
M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber.
In *Proc. POPL*.

[8] Understanding POWER Multiprocessors.
Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams.
In *Proc. PLDI*.

[9] Litmus: running tests against hardware.
Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell.

In *Proc. TACAS: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 6605.*
[url].

[10] Nitpicking C++ Concurrency.
Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar.
In PPDP 2011: *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming.*
[pdf].

[11] Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER.
Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell.
In *Proceedings of POPL 2012: The 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Philadelphia).*

[12] Synchronising C/C++ and POWER.
Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams.
In *Proceedings of PLDI, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing).*

[13] An Axiomatic Memory Model for POWER Multiprocessors.
Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams.
In *Proc. CAV, 24th International Conference on Computer Aided Verification, LNCS 7358.*

[14] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, Luc Maranget, Susmit Sarkar, and Peter Sewell., October 2012.
[pdf], Draft.

[15] Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory.
Jade Alglave, Luc Maranget, and Michael Tautschnig.
ACM Trans. Program. Lang. Syst., 36(2):7:1–7:74, 2014.
[url].

[16] An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors.
Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell.

In MICRO 2015: *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki)*.
[pdf].

[17] Modelling the ARMv8 architecture, operationally: concurrency and ISA.
Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell.
In POPL 2016: *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA)*.
[project page].
[pdf].

[18] Mixed-size Concurrency: ARM, POWER, C/C++11, and SC.
Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell.
In POPL 2017: *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Paris)*.
[project page].
[pdf].

[19] Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8.
Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell.
In POPL 2018: *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*.
[project page].
[pdf].

[20] ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS.
Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell.
In POPL 2019: *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*.
Proc. ACM Program. Lang. 3, POPL, Article 71.
[project page].
[pdf].

[21] Cerberus-BMC tool for exploring the behaviour of small concurrent C test programs with respect to an arbitrary axiomatic concurrency model, Stella Lau, Kayvan Memarian, Victor B. F. Gomes, Kyndylan Nienhuis, Justus Matthiesen, James Lingard, and Peter Sewell, 2019.
[project page].

[github], [web interface].

[22] ARMv8-A system semantics: instruction fetch in relaxed architectures.
Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell.
In ESOP 2020: *Proceedings of the 29th European Symposium on Programming*.
[project page].
[pdf].

[23] A Shared Memory Poetics.
Jade Alglave.
PhD thesis, l'Université Paris 7 – Denis Diderot, 2010.
http://www0.cs.ucl.ac.uk/staff/J.Alglave/these.pdf.

[24] The C11 and C++11 Concurrency Model.
Mark John Batty.
PhD thesis, University of Cambridge, 2014.
2015 SIGPLAN John C. Reynolds Doctoral Dissertation award and 2015 CPHC/BCS Distinguished Dissertation Competition winner.
[pdf].

[25] The Semantics of Multicopy Atomic ARMv8 and RISC-V.
Christopher Pulte.
PhD thesis, University of Cambridge, 2018.
https://www.repository.cam.ac.uk/handle/1810/292229.

[26] A no-thin-air memory model for programming languages.
Jean Pichon-Pharabod.
PhD thesis, University of Cambridge, 2018.
https://www.repository.cam.ac.uk/handle/1810/274465.

[27] The herdtools7 tool suite, Jade Alglave and Luc Maranget.
diy.inria.fr, https://github.com/herd/herdtools7/.
Accessed 2023-08-30.

[28] RMEM: Executable operational concurrency model exploration tool for ARMv8, RISC-V, Power, and x86, Susmit Sarkar, Peter Sewell, Luc Maranget, Shaked Flur, Christopher Pulte, Jon French, Ben Simner, Scott Owens, Pankaj Pawan, Francesco Zappa Nardelli, Sela Mador-Haim, Dominic Mulligan, Ohad Kammar, Jean Pichon-Pharabod, Gabriel Kerneis, Alasdair Armstrong, Thomas Bauereiss, and Jeehoon Kang, 2010–2023.
[github], [web interface].

[29] The isla-axiomatic tool, Alasdair Armstrong.
https://isla-axiomatic.cl.cam.ac.uk/.
Accessed 2020-10-10.

[30] Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, Intel Corporation.
https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html, May 2020.
Accessed 2020-09-23. 5052 pages.

[31] AMD64 Architecture Programmer's Manual Volumes 1-5, Advanced Micro Devices, Inc.
https://developer.amd.com/resources/developer-guides-manuals/, April 2020.
Accessed 2020-09-23. 3165 pages.

[32] Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile, Arm.
https://developer.arm.com/documentation/ddi0487/fc, July 2020.
Accessed 2020-09-23. 8248 pages.

[33] Power ISA Version 3.0B, IBM.
https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0, March 2017.
Accessed 2020-09-23. 1258 pages.

[34] The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213, RISC-V Foundation.
https://riscv.org/technical/specifications/, December 2019.
Contributors: Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Ken Dockser, Roger Espasa, Shaked Flur, Stefan Freudenberger, Marc Gauthier, Andy Glew, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur

Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau, Josh Scheid, Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert Watson, Derek Williams, Andrew Wright, Reinoud Zandijk, Sizhuo Zhang. 238 pages. Accessed 2023-08-30.

[35] The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203.
https://riscv.org/technical/specifications/, December 2021.
Accessed 2023-08-30. 155 pages.

[36] The Power of Processor Consistency.
Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger.
In Lawrence Snyder, editor, *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '93, Velen, Germany, June 30 - July 2, 1993*.
[url].

[37] Efficient and correct execution of parallel programs that share memory.
Dennis Shasha and Marc Snir.
ACM Trans. Program.Lang. Syst., 10(2):282–312, 1988.

[38] Trustworthy specifications of ARM® v8-A and v8-M system level architecture.
Alastair Reid.
In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*.
[url].

[39] Who guards the guards? formal validation of the Arm v8-m architecture specification.
Alastair Reid.
Proc. ACM Program. Lang., 1(OOPSLA):88:1–88:24, 2017.
[url].

[40] Safe optimisations for shared-memory concurrent programs.
Jaroslav Sevcík.
In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*.
[url].

[41] Memory Consistency Models for Shared-Memory Multiprocessors.
Kourosh Gharachorloo.
PhD thesis, Stanford University, 1995.

[42] Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.
Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy.
In Jean-Loup Baer, Larry Snyder, and James R. Goodman, editors, *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*.
[url].

[43] Designing Memory Consistency Models for Shared-Memory Multiprocessors.
S. V. Adve.
PhD thesis, University of Wisconsin-Madison, 1993.

[44] Weak Ordering - A New Definition.
Sarita V. Adve and Mark D. Hill.
In Jean-Loup Baer, Larry Snyder, and James R. Goodman, editors, *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*.
[url].

[45] The Java memory model.
Jeremy Manson, William Pugh, and Sarita V. Adve.
In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*.
[url].

[46] On Validity of Program Transformations in the Java Memory Model.
Jaroslav Sevcík and David Aspinall.
In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*.
[url].

[47] Foundations of the C++ concurrency memory model.
Hans-Juergen Boehm and Sarita V. Adve.

In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*.
[url].

[48] Overhauling SC atomics in C11 and OpenCL.
Mark Batty, Alastair F. Donaldson, and John Wickerson.
In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*.
[url].

[49] Cerberus-BMC: a Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C.
Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell.
In CAV 2019: *Proc. 31st International Conference on Computer-Aided Verification*.
[project page].
[pdf].

[50] Exploring C Semantics and Pointer Provenance.
Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell.
In POPL 2019: *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*.
Proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO/IEC JTC1/SC22/WG14 N2311.
[project page].
[pdf].

[51] P0668R5: Revising the C++ memory model, Hans-J. Boehm, Olivier Giroux, and Viktor Vafeiadis.
WG21 wg21.link/p0668, November 2018.

[52] P0982R1: Weaken Release Sequences, Hans-J. Boehm, Olivier Giroux, and Viktor Vafeiadis.
WG21 wg21.link/p0982, November 2018.

[53] Bridging the gap between programming languages and hardware weak memory models.
Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis.
Proc. ACM Program. Lang., 3(POPL):69:1–69:31, 2019.
[url].

[54] Outlawing ghosts: avoiding out-of-thin-air results.
Hans-Juergen Boehm and Brian Demsky.
In Jeremy Singer, Milind Kulkarni, and Tim Harris, editors, *Proceedings of the workshop on Memory Systems Performance and Correctness, MSPC '14, Edinburgh, United Kingdom, June 13, 2014.*
[url].

[55] Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings.
Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi.
CoRR, abs/1611.01507, 2016.
[url].

[56] Explaining Relaxed Memory Models with Program Transformations.
Ori Lahav and Viktor Vafeiadis.
In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings.*
[url].

[57] Relaxed Separation Logic: A Program Logic for C11 Concurrency.
Viktor Vafeiadis and Chinmay Narayan.
In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications.*
[url].

[58] Promising 2.0: global optimizations in relaxed memory concurrency.
Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis.
In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020.*
[url].

[59] Modular Relaxed Dependencies in Weak Memory Concurrency.
Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty.
In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.*
[url].

[60] Pomsets with Preconditions: A Simple Model of Relaxed Memory.
Radha Jagadeesan, Alan Jeffrey, and James Riely.
In *Proceedings of OOPSLA*.

[61] Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel.
Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern.
In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*.
[url].

[62] GPU Concurrency: Weak Behaviours and Programming Assumptions.
Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson.
In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*.
[url].

[63] Remote-scope promotion: clarified, rectified, and verified.
John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson.
In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*.
[url].

[64] Portable inter-workgroup barrier synchronisation for GPUs.
Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric.
In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*.
[url].

[65] Repairing and mechanising the JavaScript relaxed memory model.
Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo.

In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*.
[url].

[66] Weakening WebAssembly.
Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod.
Proc. ACM Program. Lang., 3(OOPSLA):133:1–133:28, 2019.
[url].

[67] Bounding data races in space and time.
Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy.
In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*.
[url].