Multicore Semantics: Making Sense of Relaxed Memory

Peter Sewell\textsuperscript{1}, Christopher Pulte\textsuperscript{1}, Shaked Flur\textsuperscript{1,2} 

with contributions from Mark Batty\textsuperscript{3}, Luc Maranget\textsuperscript{4}, Alasdair Armstrong\textsuperscript{1}

\textsuperscript{1} University of Cambridge, \textsuperscript{2} Google, \textsuperscript{3} University of Kent, \textsuperscript{4} INRIA Paris

October – November, 2022

Slides for Part 1 of the Multicore Semantics and Programming course, version of 2022-10-20

Part 2 is by Tim Harris, with separate slides
These Slides

These are the slides for the first part of the University of Cambridge *Multicore Semantics and Programming* course (MPhil ACS, Part III, Part II), 2021–2022.

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 Armv8-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 second part of the course, by Tim Harris, covers concurrent programming: simple algorithms, correctness criteria, advanced synchronisation patterns, transactional memory.
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, 30, 31], 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.
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

Contents

1.1 Introduction: Memory
"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).
Intel Xeon E7-8895 v3
36 hardware threads

Commonly 8 hardware threads.

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 \textit{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

<table>
<thead>
<tr>
<th>Initial state: x=0; y=0;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
</tr>
<tr>
<td>x=1;</td>
</tr>
<tr>
<td><strong>if</strong> (y==0) { ...critical section... }</td>
</tr>
</tbody>
</table>

Can both be in their critical sections at the same time, in SC?
A naive two-thread mutual-exclusion algorithm

<table>
<thead>
<tr>
<th>Initial state: $x=0$; $y=0$;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
</tr>
<tr>
<td>$x=1$; $r_0=y$</td>
</tr>
</tbody>
</table>

Is a final state with $r_0=0$ and $r_1=0$ possible in SC?
A naive two-thread mutual-exclusion algorithm

Initial state: $x=0; y=0$

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x=1$; $r0=y$</td>
<td>$y=1$; $r1=x$</td>
</tr>
</tbody>
</table>

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

Try all six interleavings of SC model:

```
0: Ry = 0
1: Wx = 1
1: Wy = 1
r0 = 1  r1 = 0

0: Wx = 1
1: Wy = 1
0: Ry = 1
r0 = 1  r1 = 1

1: Wx = 1
0: Wx = 1
0: Ry = 1
r0 = 0  r1 = 1

1: Wx = 1
0: Wx = 1
0: Ry = 1
r0 = 0  r1 = 1
```
Let's try...

We’ll use the litmus7 tool ([diy.inria.fr](http://diy.inria.fr), Alglave, Maranget, et al. [29])

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

```plaintext
1  X86_64 SB
2  "PodWR Fre PodWR Fre"
3  Syntax=gas
4  {
5    uint64_t x=0; uint64_t y=0;
6    uint64_t 0:rax; uint64_t 1:rax;
7  }
8  P0        | P1        ;
9  movq $1,(x) | movq $1,(y) ;
10  movq (y),%rax | movq (x),%rax ;
11  exists (0:rax=0 /\ 1:rax=0)
```

Use litmus7 to generate a test harness (C + embedded assembly), build it, and run it
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

```asm
#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:
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 [32], 5052 pages
▶ AMD64 Architecture Programmer’s Manual [33], 3165 pages
▶ Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile [34], 8248 pages
▶ Power ISA Version 3.0B [35], 1258 pages
▶ The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [36] and Volume II: Privileged Architecture [37], 238+135 pages
Architecture specifications

Thick books:

- Intel 64 and IA-32 Architectures Software Developer’s Manual [32], 5052 pages
- AMD64 Architecture Programmer’s Manual [33], 3165 pages
- Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile [34], 8248 pages
- Power ISA Version 3.0B [35], 1258 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; x=0; y=0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td></td>
</tr>
<tr>
<td>movq (y), %rax</td>
<td></td>
</tr>
<tr>
<td>movq $1, (y)</td>
<td></td>
</tr>
<tr>
<td>movq (x), %rax</td>
<td></td>
</tr>
</tbody>
</table>

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; x=0; y=0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td>movq (y), %rax</td>
<td>movq (x), %rax</td>
</tr>
</tbody>
</table>

**Final:** 0:rax=0; 1:rax=0;

**Observation:** 171/100000000

---

### 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]
SB x86

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000

▶ experimentally: observed
SB

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

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; x=0; y=0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000

▶ experimentally: observed

▶ possible microarchitectural explanation? buffer stores? out-of-order execution?

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]

**Write Buffer**

**Shared Memory**

**Thread**

**Write Buffer**

**Thread**

**Shared Memory**

Contents 2.1 x86: x86 basic phenomena
SB

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

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]
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movq (x), %rax</code> //a</td>
<td><code>movq (y), %rax</code> //c</td>
</tr>
<tr>
<td><code>movq $1, (y)</code> //b</td>
<td><code>movq $1, (x)</code> //d</td>
</tr>
</tbody>
</table>

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

**Final:** 0:rax=1; 1:rax=1;

**Observation:** 0/0

---

**x86**

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

**Final:** 0:rax=1; 1:rax=1;

**Observation:** 0/0

---

**Contents** 2.1 x86: x86 basic phenomena

---

**LB**

**x86**

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

**Final:** 0:rax=1; 1:rax=1;

**Observation:** 0/0

---

**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]
**Initial state:** 0:rax=0; 1:rax=0; y=0; x=0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq (x), %rax  //a</td>
<td>movq (y), %rax  //c</td>
</tr>
<tr>
<td>movq $1, (y)  //b</td>
<td>movq $1, (x)  //d</td>
</tr>
</tbody>
</table>

**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;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq (x), %rax //a</td>
<td>movq (y), %rax //c</td>
</tr>
<tr>
<td>movq $1, (y) //b</td>
<td>movq $1, (x) //d</td>
</tr>
</tbody>
</table>

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

Observation: 0/0

- experimentally: not observed
- possible microarchitectural explanation?
Initial state: 0: rax=0; 1: rax=0; y=0; x=0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq (x), %rax //a</td>
<td>movq (y), %rax //c</td>
</tr>
<tr>
<td>movq $1, (y) //b</td>
<td>movq $1, (x) //d</td>
</tr>
</tbody>
</table>

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

Observation: 0/0

- experimentally: not observed
- possible microarchitectural explanation?
  Buffer load requests?
  Out-of-order execution?
Initial state: 0:rax=0; 1:rax=0; y=0; x=0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq (x), %rax //a</td>
<td>movq (y), %rax //c</td>
</tr>
<tr>
<td>movq $1, (y) //b</td>
<td>movq $1, (x) //d</td>
</tr>
</tbody>
</table>

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?

[Drawings of thread execution and memory access to illustrate synchronization and reordering.]
### Example 8-1

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a movq (y), %rax //c</td>
<td></td>
</tr>
<tr>
<td>movq $1, (y) //b movq (x), %rbx //d</td>
<td></td>
</tr>
</tbody>
</table>

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

Observation: 0/100000000

---

MP x86

**Contents**

2.1 x86: x86 basic phenomena
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;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq (y), %rax</td>
</tr>
<tr>
<td>movq $1, (y)</td>
<td>movq (x), %rbx</td>
</tr>
</tbody>
</table>

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

Thread 0

movq $1, (x) //a
movq (y), %rax //c

Thread 1

movq $1, (y) //b
movq (x), %rbx //d

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

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //d</td>
</tr>
<tr>
<td>movq (x), %rax //b</td>
<td>movq (y), %rax //e</td>
</tr>
<tr>
<td>movq (y), %rbx //c</td>
<td>movq (x), %rbx //f</td>
</tr>
</tbody>
</table>

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

Observation: 320/100000000
SB+rfi-pos

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //d</td>
</tr>
<tr>
<td>movq (x), %rax //b</td>
<td>movq (y), %rax //e</td>
</tr>
<tr>
<td>movq (y), %rbx //c</td>
<td>movq (x), %rbx //f</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //d</td>
</tr>
<tr>
<td>movq (x), %rax //b</td>
<td>movq (y), %rax //e</td>
</tr>
<tr>
<td>movq (y), %rbx //c</td>
<td>movq (x), %rbx //f</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td>movq (x), %rax</td>
<td>movq (y), %rax</td>
</tr>
<tr>
<td>movq (y), %rbx</td>
<td>movq (x), %rbx</td>
</tr>
</tbody>
</table>

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

Observation: 0/100000000

Contents 2.1 x86: x86 basic phenomena 62
is that allowed in the previous model sketch?
is that allowed in the previous model sketch?
we think the T2,3 read pairs are not reorderable – so no
is that allowed in the previous model sketch?
we think the T2,3 read pairs are not reorderable – so no
is it microarchitecturally plausible?
### Initial state:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq (x), %rax //b</td>
<td>movq (y), %rbx //c</td>
<td>movq (y), %rax //d</td>
</tr>
<tr>
<td>movq (y), %rbx //e</td>
<td>movq (x), %rbx //f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Final:**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W x = 1 rf</td>
<td>b: R x = 1 rf</td>
<td>d: W y = 1 rf</td>
<td>e: R y = 1 rf</td>
</tr>
<tr>
<td>c: R y = 0 po</td>
<td>f: R x = 0 po</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

---

![Diagram](image-url)

**Contents**

2.1 **x86: x86 basic phenomena**
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a movq (x), %rax //b movq $1, (y) //d movq (y), %rax //e movq (y), %rbx //f</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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
IR/W

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq (x), %rax //b</td>
<td>movq $1, (y) //d</td>
<td>movq (y), %rax //e</td>
</tr>
<tr>
<td>movq (y), %rbx //c</td>
<td>movq (x), %rbx //f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq (x), %rax //b</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //d</td>
<td>movq (x), %rbx //e</td>
<td></td>
</tr>
</tbody>
</table>

Final: 1:rax=1; 2:rax=1; 2:rbx=0;
Observation: 0/100000000

Thread 0

a: W x=1
b: R x=1
c: W y=1
d: R y=1
e: R x=0

Thread 1

Thread 2

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.

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

Contents 2.1 x86: x86 basic phenomena
WRC x86

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq %rax, (x) //b</td>
<td>movq (x), %rax //c</td>
</tr>
<tr>
<td>movq $1, (y) //d</td>
<td>movq %rax, (y) //e</td>
<td>movq (x), %rbx //f</td>
</tr>
</tbody>
</table>

Final: 1: rax=1; 2: rax=1; 2: rbx=0;
Observation: 0/100000000

—is that allowed in the previous model sketch?
is that allowed in the previous model sketch?

we think the T1 read-write pair and T2 read pair are not reorderable – so no
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

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

Model sketch remains experimentally plausible, but interpretation of vendor prose unclear
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
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;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td>a: W x=1</td>
<td>d: W y=1</td>
</tr>
<tr>
<td>mfence</td>
<td>mfence</td>
</tr>
<tr>
<td>c: R y=0</td>
<td>f: R x=0</td>
</tr>
<tr>
<td>movq (y), %rax</td>
<td>movq (x), %rax</td>
</tr>
<tr>
<td>RF</td>
<td>RF</td>
</tr>
</tbody>
</table>

Final: 0:rax=0; 1:rax=0;
Observation: 0/100000000

Contents 2.1 x86: x86 basic phenomena
## Contents

2.1 **x86**: x86 basic phenomena

---

### SB+mfences x86

| Initial state: 0:rax=0; 1:rax=0; y=0; x=0; |
|---|---|
| Thread 0 | Thread 1 |
| movq $1, (x) | movq $1, (y) |
| //a | //d |
| mfence | mfence |
| //b | //e |
| movq (y), %rax | movq (x), %rax |
| //c | //f |

<table>
<thead>
<tr>
<th>Final: 0:rax=0; 1:rax=0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observation: 0/100000000</td>
</tr>
</tbody>
</table>

- experimentally: not observed
### Initial state: 0:rax=0; 1:rax=0; y=0; x=0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //d</td>
</tr>
<tr>
<td>mfence //b</td>
<td>mfence //e</td>
</tr>
<tr>
<td>movq (y), %rax //c</td>
<td>movq (x), %rax //f</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td></td>
<td>mfence</td>
<td>mfence</td>
</tr>
<tr>
<td></td>
<td>movq (y), %rax</td>
<td>movq (x), %rax</td>
</tr>
</tbody>
</table>

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

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

Contents 2.1 x86: x86 basic phenomena 79

![Diagram of threads and shared memory with mfence instructions and writes](image-url)
Adding Read-Modify-Write instructions

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

<table>
<thead>
<tr>
<th>INC</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state: x=0;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq (x) //a0,a1</td>
<td>incq (x) //b0,b1</td>
</tr>
<tr>
<td>Final: x=1;</td>
<td></td>
</tr>
</tbody>
</table>

Observation: 1441/1000000

Thread 0
- a0: R x=0
- a1: W x=1

Thread 1
- b0: R x=0
- b1: W x=1
Adding Read-Modify-Write instructions

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

\[
\text{INC} \quad \text{x86}
\]

<table>
<thead>
<tr>
<th>Initial state: (x=0;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>incq (x) //a0, a1</td>
</tr>
<tr>
<td>Final: (x=1;)</td>
</tr>
<tr>
<td>Observation: 1441/1000000</td>
</tr>
</tbody>
</table>

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

\[
\text{Initial state: } x=0; \\
\text{Final: } x=1; \\
\text{Observation: } 0/1000000
\]
Adding Read-Modify-Write instructions

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

<table>
<thead>
<tr>
<th>LOCKINC</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state: x=0;</td>
<td></td>
</tr>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>lock incq (x) //a0,a1</td>
<td>lock incq (x) //b0,b1</td>
</tr>
<tr>
<td>Final: x=1;</td>
<td></td>
</tr>
<tr>
<td>Observation: 0/1000000</td>
<td></td>
</tr>
</tbody>
</table>

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/):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>“Best-case” CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Single cache miss (off-core)</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss (off-core)</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Single cache miss (off-socket)</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss (off-socket)</td>
<td>95.9</td>
<td>266.4</td>
</tr>
</tbody>
</table>

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
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.

Note that the fact that it does not crash now is quite possibly because of either:

- **unlock()**
- **spinlock**

---

**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!*
A Pentium is an 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 infinitesimally small, most kernels don't hit it (Netware 4/5 uses this method but it's spinlocks understand this and the code is written to handle it. The most obvious aberrant behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the pipelines 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.
The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until it is committed state, and the earlier instructions are already committed by that time), so the any loads, stores, etc. absolutely have to have completed first, cache-miss or not.

He went on:

Since the instructions for the store in the spin_unlock have to have been externally observed for spin_lock to be acquired (presuming a correctly functioning spinlock, of course), then the earlier instructions to set "b" to the
History of x86 concurrency specs

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

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.
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
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.
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 $m'$.
Events $e$ ::= $a:t:W \ x=\nu$ a write of value $\nu$ to address $x$ by thread $t$, ID $a$
| $a:t:R \ x=\nu$ a read of $\nu$ from $x$ by $t$
| $a:t:D_w \ x=\nu$ an internal action of the abstract machine, dequeuing $w=(a':t':W \ x=\nu)$ from thread $t$'s write buffer to shared memory
| $a:t:F$ an MFENCE memory barrier by $t$
| $a:t:L$ start of an instruction with LOCK prefix by $t$
| $a:t:U$ 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 $addr$
- $\nu$ and $w$ are memory values, of type $value$
- $w$ is a write event $a:t:W \ x=\nu$, of type $write\_event$
x86-TSO Abstract Machine: Memory States

An x86-TSO abstract machine memory state \( m \) is a record

\[
m : \{ M : \text{addr} \rightarrow \text{value}; \\
B : \text{tid} \rightarrow \text{write_event list}; \\
L : \text{tid option}\}
\]

Here:

- \( m.M \) is the shared memory, mapping addresses to values
- \( m.B \) gives the store buffer for each thread, a list with most recent at the head (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

Write \( m_0 \) for the initial state with \( m.M = M_0 \), \( s.B \) empty for all threads, and \( m.L = \text{None} \) (lock not taken).
Notation

Some and None construct optional values

(·, ·) builds tuples

[] builds lists

@ appends lists

· ⊕ ⟨· := ·⟩ updates records

· ⊕ (· ↦→ ·) updates functions.

id(e), thread(e), addr(e), value(e) extract the respective components of event e

isread(e), iswrite(e), isdequeue(e), ismfence(e) identify the corresponding kinds
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 $\text{addr}(w) = x$.

Say $t$ is *not blocked* in machine state $s$ if either it holds the lock ($m.L = \text{Some } t$) or the lock is not held ($m.L = \text{None}$).
x86-TSO Abstract Machine: Behaviour

RM: Read from memory
\[
\begin{align*}
\text{not\_blocked}(m, t) \\
m.M(x) &= v \\
\text{no\_pending}(m.B(t), x)
\end{align*}
\]

\[
\frac{m \quad \text{a:t:R } x=v \rightarrow m}{m}
\]

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)
RB: Read from write buffer

\[ \text{not\_blocked}(m, t) \]
\[ \exists b_1 b_2. \; m.B(t) = b_1 \oplus [a':t:W x=v] \oplus b_2 \]
\[ \text{no\_pending}(b_1, x) \]

\[ \frac{}{m \quad a:t:R x=v \rightarrow m} \]

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.
WB: Write to write buffer

\[
m \xrightarrow{a:t:W x=v} m \oplus \{B := m.B \oplus (t \mapsto ([a:t:W x=v] @ m.B(t)))\}
\]

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

\[ \text{not\_blocked}(m, t) \quad m.B(t) = b @ [a':t:W x=v] \]

\[
\begin{array}{c}
\frac{a:t:D_{a':t:W x=v} x=v}{m} \rightarrow m \oplus \{M := m.M \oplus (x \mapsto v)\} \oplus \{B := m.B \oplus (t \mapsto b)\}
\end{array}
\]

If \( t \) 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)
M: MFENCE

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

If 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

Define the instruction semantics for locked instructions, e.g. lock inc x to bracket the transitions of inc with \( a:t:L \) and \( a':t:U \)

For example, lock inc x, in thread \( t \), will do

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

(this lets us reuse the inc semantics for lock inc, and to do so uniformly for all RMWs)
L: Lock

\[
\begin{align*}
  m.L &= \text{None} \\
  m.B(t) &= []
\end{align*}
\]

\[
\begin{array}{c}
  \quad m \xrightarrow{a:t:L} m \oplus \{L := \text{Some}(t)\}
\end{array}
\]

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 \( a:t:D_w x = v \) steps to empty the buffer and then proceed.
**U: Unlock**

\[
\begin{align*}
m.L &= \text{Some}(t) \\
m.B(t) &= []
\end{align*}
\]

\[
m \xrightarrow{a:t:U} m \oplus \{ L := \text{None} \}
\]

If \( t \) holds the lock, and its store buffer is empty, it can end a LOCK’d instruction.
First Example, Revisited

SB

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td>//a</td>
<td>//c</td>
</tr>
<tr>
<td>movq (y), %rax</td>
<td>movq (x), %rax</td>
</tr>
<tr>
<td>//b</td>
<td>//d</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000

Thread 0

a: W x=1
b: R y=0

Thread 1

c: W y=1
d: R x=0

Contents 2.3 x86: x86-TSO operational model
First Example, Revisited

<table>
<thead>
<tr>
<th>SB</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state: 0:rax=0; 1:rax=0; x=0; y=0;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

| Final: 0:rax=0; 1:rax=0; |

Observation: 171/100000000

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

Initial state: 0:rax=0; 1:rax=0; x=0; y=0;
SB x86
Final: 0:rax=0; 1:rax=0;
Observation: 171/100000000
```
First Example, Revisited

SB

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td>movq (y), %rax</td>
<td>movq (x), %rax</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000

SB

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td>movq (y), %rax</td>
<td>movq (x), %rax</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000

Contents 2.3 x86: x86-TSO operational model
First Example, Revisited

<table>
<thead>
<tr>
<th>SB</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial state:</strong> 0:rax=0; 1:rax=0; x=0; y=0;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)  //a</td>
<td>movq $1, (y)  //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

| 0:rax=0; 1:rax=0; |

Observation: 171/100000000

Thread 0:
- a: \( W \) x=1
- b: R y=0

Thread 1:
- c: W y=1
- d: R x=0

Contents 2.3 x86: x86-TSO operational model
First Example, Revisited

SB

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000
First Example, Revisited

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x)</td>
<td>movq $1, (y)</td>
</tr>
<tr>
<td>movq (y), %rax</td>
<td>movq (x), %rax</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000
First Example, Revisited

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $1, (y) //c</td>
</tr>
<tr>
<td>movq (y), %rax //b</td>
<td>movq (x), %rax //d</td>
</tr>
</tbody>
</table>

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

Observation: 171/100000000

Thread 0

a: W x=1
b: R y=0

d: R x=0

Thread 1

c: W y=1

Thread

Write Buffer

Lock

x=1

Shared Memory

y=1

Contents 2.3 x86: x86-TSO operational model 121
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 nops in place of mfences in SC.

What does “essentially identical” mean? The same set of interface traces except with the $a:t:F$ and $a:t:D_w x=v$ 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.

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 9, Revision SAV09R1459912. 1994

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.
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. [30]).

Either use the in-browser web interface:
http://www.cl.cam.ac.uk/users/pes20/rmem
or install locally and use the CLI interface
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
Storage subsystem state (TSO):

0

1

Lock = unlocked

Memory = \([1000:0:0]::0x1000 (y)/R=0, [1001:1:0]::0x1100 (x)/R=0\)

Thread 0 state:
0: 0x62000 fetched mov 1 (rax) reg reads: RAX=0x63'00000000001100 (x) from initial state micro_op_state: M5 pending mem_write
0 (1:0)::0x1100 (x)/R=1
0 0 translate memory write to storage: (0:1)::0x1100 (y)/R=1

Thread 1 state:
1: 0x251000 fetched mov 1 (rax) reg reads: RAX=0x63'00000000001000 (y) from initial state micro_op_state: M5 pending mem_write
1 (1:0)::0x1000 (y)/R=1
1 1 translate memory write to storage: (1:1)::0x1000 (y)/R=1

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

Console

Step 1 (0/2 finished, 0 trns) choose [0]: 0
Step 1 (0/2 finished, 0 trns) choose [0]: 1
Storage subsystem state (T50):

Thread 0 state:

Thread 1 state:

Choices so far: 3
3 enabled transitions
No disabled transitions
Memory = \{1000:0\}: W x0100 (y)/8=0, \(0:1\): W x01100 (x)/8=1, \(1000:1\): W x01100 (x)/8=0
Lock = unlocked
A -propagate write to memory: 1:1:0, W 0x1000 (y)/8=0

Thread 0 state:
0:1 8xS50000 fetched movq 1, (\vax) mem writes: (0:1:0) 1: W x1000 (y)/8=1 reg reads:
RAX=0x 63'000000000001000 \(x\) from initial state
Micro op state: M5G plain
0:2 8xS50000 fetched movq (\vbx), \vrcx mem reads: (0:2:0) R from (1000:0): W x1000 (y)/8=0 reg reads:
RXO=RX 63'000000000001000 \(x\) from initial state reg writes: RCX=RX 63'000000000000000000 micro op state: M5G plain

Thread 1 state:
1:1 8xS51000 fetched movq 1, (\vax) mem writes: (1:1:0) 1: W x1000 (y)/8=1 reg reads:
RAX=0x 63'000000000001000 \(y\) from initial state
Micro op state: M5G plain
1:2 8xS51000 fetched movq (\vbx), \vrcx mem reads: (1:2:0) R from (1000:1): W x1000 (x)/8=0 reg reads:
RXO=RX 63'000000000001000 \(x\) from initial state reg writes: RCX=RX 63'000000000000000000 micro op state: M5G plain

Choices so far (5): 0:1, 0:2, 1:1
1 enabled transitions
No disabled transitions
Storage subsystem state (T50):

0

Memory = [(1:1:0):W 0x1000 (y)/8=1, (0:1:0):W 0x1100 (x)/8=1, (0000:8:0):W 0x1000 (y)/8=0]
Lock = unlocked

Thread 0 state:
0:0 8x550000 fetched movq 1, (%rax) mem writes: (0:1:0):W 0x1100 (x)/8=1 reg reads: RAX=0x63'80000000000001100 (x) from initialstate
micro_op_state: MOS_plain
0:2 8x550000 fetched movq (%rbx), %rcx mem reads: (0:2:0):R from (0000:8:0):W 0x1000 (y)/8=0 reg writes: R0x63'8000000000001800 (y) from initialstate reg writes: RCX=0x63'8000000000000000
micro_op_state: MOS_plain

Thread 1 state:
1:1 8x510000 fetched movq 1, (%rax) mem writes: (1:1:0):W 0x1000 (y)/8=1 reg reads:
RAX=0x63'80000000000001100 (y) from initialstate
micro_op_state: MOS_plain
1:2 8x510000 fetched movq (%rbx), %rcx mem reads: (1:2:0):R from (0000:1:0):W 0x1000 (x)/8=0 reg writes: RAX=0x63'80000000000001100 (x) from initialstate reg writes: RCX=0x63'8000000000000000
micro_op_state: MOS_plain

Choices so far: 0; 1,0; 1,2,3
No enabled transitions
No disabled transitions
Making x86-TSO executable as a test oracle: the RMEM tool

To install RMEM locally:

1. install the opam package manager for OCaml: https://opam.ocaml.org/
2. opam repository add rems
   https://github.com/rems-project/opam-repository.git#opam2
3. opam install rmem

Docs at https://github.com/rems-project/rmem.

Better performance than the web interface
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
- set always_print true
- set always_graph true
- <N>
- b
- search exhaustive

list commands
print the current state after every command
generate a pdf graph in out.pdf after every step
take transition labelled <N>, and eager successors
step back one transition
exhaustive search from the current state

[...]

Contents 2.3 x86: x86-TSO operational model 137
Storage subsystem state (TSO):
0
1

Memory = [(1000:0:0):W 0x0000000000000100 (y)/8=0, (1000:1:0):W 0x0000000000000110 (x)/8=0]
Lock = unlocked

Thread 0 state:

Thread 1 state:

Choices so far (0):
Enabled transitions:
*** 0 0:1 propagate memory write to storage: (0:1:0):W 0x0000000000000100 (x)/8=0 ***
1 1:1 propagate memory write to storage: (1:1:0):W 0x0000000000000100 (y)/8=1

No disabled transitions
Step 1 (0/2 finished, 6 trns) Choose [0]:

Contents
x86
x86-TSO operational model
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 \( s ::= ... \mid \text{lock} \, x \mid \text{unlock} \, x \)

Say lock free if it holds 0, taken otherwise.

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

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

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

Consider

\[ P = \begin{align*}
& t_1 : \langle \text{lock } m; r = x; x = r + 1; \text{unlock } m, R_0 \rangle \\
& \mid t_2 : \langle \text{lock } m; r = x; x = r + 7; \text{unlock } m, R_0 \rangle 
\end{align*} \]

in the initial store \( M_0 \):

\[ \langle t_1 : \langle \text{skip}; r = x; x = r + 1; \text{unlock } m, R_0 \rangle | t_2 : \langle \text{lock } m; r = x; x = r + 7; \text{unlock } m, R_0 \rangle, M' \rangle \]

where \( M' = M_0 \oplus (m \mapsto 1) \)
lock $m$ can block (that's the point). Hence, you can *deadlock*.

$$P = \begin{align*} t_1 : & \langle \text{lock } m_1; \text{lock } m_2; x = 1; \text{unlock } m_1; \text{unlock } m_2, R_0 \rangle \\ \mid t_2 : & \langle \text{lock } m_2; \text{lock } m_1; x = 2; \text{unlock } m_1; \text{unlock } m_2, R_0 \rangle \end{align*}$$
Implementing mutexes with simple x86 spinlocks

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

\[
\begin{array}{c}
\text{lock } x \\
\text{critical section} \\
\text{unlock } x
\end{array}
\]
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
```

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
</tbody>
</table>
Spinlock Example (SC)

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

critical section
x ← 1

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td></td>
<td>critical</td>
</tr>
</tbody>
</table>
Spinlock Example (SC)

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

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>critical</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
</tbody>
</table>
Spinlock Example (SC)

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

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>critical</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
</tbody>
</table>
Spinlock Example (SC)

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

critical section
x ← 1
```

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>critical</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = 1</td>
<td>release, writing x</td>
<td></td>
</tr>
</tbody>
</table>
Spinlock Example (SC)

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

critical section
x ← 1
```

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>critical</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = 1</td>
<td>release, writing x</td>
<td></td>
</tr>
<tr>
<td>x = 1</td>
<td></td>
<td>read x</td>
</tr>
</tbody>
</table>
Spinlock Example (SC)

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

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>critical</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = 1</td>
<td>release, writing x</td>
<td></td>
</tr>
<tr>
<td>x = 1</td>
<td></td>
<td>read x</td>
</tr>
<tr>
<td>x = 0</td>
<td></td>
<td>acquire</td>
</tr>
</tbody>
</table>
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
```

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
</tbody>
</table>
while atomic_decrement(x) < 0 {
    while x \leq 0 { skip }
}

critical section
x \leftarrow 1

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
</tbody>
</table>
Spinlock Example (x86-TSO)

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

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
</tbody>
</table>
Spinlock Example (x86-TSO)

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

critical section
x ← 1
```

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = -1</td>
<td>release, writing x to buffer</td>
<td></td>
</tr>
</tbody>
</table>
Spinlock Example (x86-TSO)

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

**critical section**

\[ x \leftarrow 1 \]

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x = 0 )</td>
<td>acquire</td>
<td>acquire</td>
</tr>
<tr>
<td>( x = -1 )</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>( x = -1 )</td>
<td>critical</td>
<td>spin, reading ( x )</td>
</tr>
<tr>
<td>( x = -1 )</td>
<td>release, writing ( x ) to buffer</td>
<td>spin, reading ( x )</td>
</tr>
<tr>
<td>( x = -1 )</td>
<td>...</td>
<td>spin, reading ( x )</td>
</tr>
</tbody>
</table>
Spinlock Example (x86-TSO)

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

critical section
x ← 1
```

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>release, writing x to buffer</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = -1</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>x = 1</td>
<td>write x from buffer</td>
<td></td>
</tr>
</tbody>
</table>
Spinlock Example (x86-TSO)

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

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = -1</td>
<td>release, writing x to buffer</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = -1</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>x = 1</td>
<td>write x from buffer</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = 1</td>
<td></td>
<td>read x</td>
</tr>
</tbody>
</table>
Spinlock Example (x86-TSO)

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

critical section
x ← 1
```

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = -1</td>
<td>release, writing x to buffer</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = -1</td>
<td>...</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = 1</td>
<td>write x from buffer</td>
<td>read x</td>
</tr>
<tr>
<td>x = 1</td>
<td></td>
<td>acquire</td>
</tr>
<tr>
<td>x = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Spinlock SC Data Race

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

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td>critical</td>
<td></td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>acquire</td>
</tr>
<tr>
<td>x = -1</td>
<td>critical</td>
<td>spin, reading x</td>
</tr>
<tr>
<td>x = 1</td>
<td>release, writing x</td>
<td></td>
</tr>
</tbody>
</table>
while atomic_decrement(x) < 0 {
    while x ≤ 0 { skip }
}
critical section
x ← 1

Shared Memory   Thread 0   Thread 1
x = 1
Triangular Races

Owens [6, ECOOP 2010]

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

Triangular race

```
  y ← v_2
  x ← v_1
```

---

Contents  2.4 x86: x86-TSO operational model  169
Triangular Races

Owens [6, ECOOP 2010]

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

<table>
<thead>
<tr>
<th>Triangular race</th>
<th>Not triangular race</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>y ← v₂</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>x ← v₁</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Contents 2.4 x86: x86-TSO operational model
Triangular Races

Owens [6, ECOOP 2010]

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

<table>
<thead>
<tr>
<th>Triangular race</th>
<th>Not triangular race</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y ← v₂</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x ← v₁</td>
<td>mfence</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>x ← v₁</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Contents 2.4 x86: x86-TSO operational model 171
Triangular Races

Owens [6, ECOOP 2010]

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

<table>
<thead>
<tr>
<th>Triangular race</th>
<th>Not triangular race</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>y ← v₂</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>x ← v₁</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>
Triangular Races

Owens [6, ECOOP 2010]

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

<table>
<thead>
<tr>
<th>Triangular race</th>
<th>Not triangular race</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>y ← v_2</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>x ← v_1</td>
<td>x</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Contents 2.4 x86: x86-TSO operational model 173
Triangular Races

Owens [6, ECOOP 2010]

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

```plaintext
Triangular race

\[
\begin{array}{ll}
& y \leftarrow v_2 \\
\vdots & \vdots \\
\vdots & \vdots \\
\text{lock} & x \leftarrow v_1 \\
\vdots & \vdots \\
\end{array}
\]

```

Contents 2.4 x86: x86-TSO operational model
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*

\[ \text{x86-TSO} \equiv \text{SC} \]
Spinlock Data Race

\[
\text{while } \text{atomic_decrement}(x) < 0 \{ \\
  \text{while } x \leq 0 \{ \text{skip} \} \\
\}
\]

\text{critical section}

\[x \leftarrow 1\]

\[x = 1\]
\[x = 0 \quad \text{acquire}\]
\[x = -1 \quad \text{critical} \quad \text{acquire}\]
\[x = -1 \quad \text{critical} \quad \text{spin, reading } x\]
\[x = 1 \quad \text{release, writing } x\]

\[\blacktriangleright \text{ acquire's writes are locked}\]
**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.
We’ll include the coherence order in the data of a candidate execution, e.g.

<table>
<thead>
<tr>
<th>1+1W</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state: x=0;</td>
<td></td>
</tr>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>movq $1, (x) //a</td>
<td>movq $2, (x) //b</td>
</tr>
<tr>
<td>Final: x=2;</td>
<td></td>
</tr>
</tbody>
</table>

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.

Contents 2.5 x86: Axiomatic models
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 \([38, 3]\):

\[ w \xrightarrow{fr} r \iff r \text{ reads from a coherence-predecessor of } w. \]
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 \[38, 3\]:

\[ w \xrightarrow{fr} r \text{ iff } r \text{ reads from a coherence-predecessor of } w. \]

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

\[
\begin{align*}
r \xrightarrow{fr} w & \text{ iff } (\exists w_0. \ w_0 \xrightarrow{co} w \land w_0 \xrightarrow{rf} r) \lor \\
& (\text{iswrite}(w) \land \text{addr}(w) = \text{addr}(r) \land \neg \exists w_0. \ w_0 \xrightarrow{rf} r)
\end{align*}
\]

**Lemma 1.** For any same-address read \( r \) and write \( w \), either \( w \xrightarrow{co}^* \xrightarrow{rf} r \), or \( r \xrightarrow{fr} w \).

(writing \( \xrightarrow{co}^* \) for the reflexive-transitive closure of \( \xrightarrow{co} \))
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 $\rightarrow_{rf}$ and $\rightarrow_{fr}$ 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 \( e \ ::= \ a:t:W \ x = v \) a write of value \( v \) to address \( x \) by thread \( t \), ID \( a \)

\[ | \quad a:t:R \ x = v \] a read of \( v \) from \( x \) by \( t \), ID \( a \)

States \( m \) are just memory states:

\[ m : addr \rightarrow value \]

**RM: Read from memory**

\[
\begin{align*}
  m(x) &= v \\
  m &\xrightarrow{a:t:R \ x = v} m
\end{align*}
\]

**WM: Write to memory**

\[
\begin{align*}
  m &\xrightarrow{a:t:W \ x = v} m \oplus (x \mapsto v)
\end{align*}
\]
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
\[ \forall i, j \in 1..n. \ i \neq j \implies \text{id}(e_i) \neq \text{id}(e_j) \]

Write:
\[ e \prec e' \iff e \text{ is before } e' \text{ in the trace} \quad e \prec e' \iff \exists i, j. \ e = e_i \land e' = e_j \land i < j \]

Say the traces of the SC abstract machine memory are all traces $T = [e_1, \ldots, e_n]$ with unique IDs such that
\[ m_0 \xrightarrow{e_1} m_1 \ldots \xrightarrow{e_n} m_n \]

for the initial memory state $m_0 = \lambda x : \text{addr} \cdot 0$ 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 \( \langle E, \text{po} \rangle \), 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 for now.)

and a candidate execution witness \( X = \langle \text{rf}, \text{co} \rangle \), 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. \( \forall e, e'. \ e \neq e' \implies \text{id}(e) \neq \text{id}(e') \)
- **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 for now.)

\[
\forall e. \neg (e \xrightarrow{\text{po}} e) \quad \forall e, e', (\text{thread}(e) = \text{thread}(e') \land e \neq e') \implies e \xrightarrow{\text{po}} e' \lor e' \xrightarrow{\text{po}} e
\]

\[
\forall e, e', e''. (e \xrightarrow{\text{po}} e' \land e' \xrightarrow{\text{po}} e'') \implies e \xrightarrow{\text{po}} e''
\]

\[
\forall e, e'. e \xrightarrow{\text{po}} e' \implies \text{thread}(e) = \text{thread}(e')
\]


- 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)

\[
\forall e, e', e''. (e \xrightarrow{\text{rf}} e' \land e' \xrightarrow{\text{rf}} e'') \implies e = e'
\]

\[
\forall e, e'. e \xrightarrow{\text{rf}} e' \implies \text{iswrite}(e) \land \text{isread}(e') \land \text{addr}(e) = \text{addr}(e') \land \text{value}(e) = \text{value}(e')
\]

\[
\forall e. (\text{isread}(e) \land \neg \exists e'. e' \xrightarrow{\text{rf}} e) \implies \text{value}(e) = m_0(\text{addr}(e))
\]

  - **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

\[
\forall e. \neg (e \xrightarrow{\text{co}} e)
\]

\[
\forall e, e', e''. (e \xrightarrow{\text{co}} e' \land e' \xrightarrow{\text{co}} e'') \implies e \xrightarrow{\text{co}} e''
\]

\[
\forall e, e'. e \xrightarrow{\text{co}} e' \implies \text{iswrite}(e) \land \text{iswrite}(e') \land \text{addr}(e) = \text{addr}(e')
\]

\[
\forall a. \forall e, e'. (e \neq e' \land \text{iswrite}(e) \land \text{iswrite}(e') \land \text{addr}(e) = a \land \text{addr}(e') = a) \implies e \xrightarrow{\text{co}} e' \lor e' \xrightarrow{\text{co}} e
\]
SC, axiomatically

Say a trace $T = [e_1, \ldots, e_n]$ and a candidate pre-execution $\langle E, \xrightarrow{po} \rangle$ 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.
  $\xrightarrow{po} = \{(e, e') \mid e < e' \land \text{thread}(e) = \text{thread}(e')\}$

Then:

**Theorem 4.** If $T$ and $\langle E, \xrightarrow{po} \rangle$ 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 $X = \langle \xrightarrow{rf}, \xrightarrow{co} \rangle$ for $\langle E, \xrightarrow{po} \rangle$ such that $\text{acyclic}(\xrightarrow{po} \cup \xrightarrow{rf} \cup \xrightarrow{co} \cup \xrightarrow{fr})$. 
Proof. For left-to-right, given the trace order $<$, construct an execution witness:

$$e \xrightarrow{rf} e' \iff \text{iswrite}(e) \land \text{isread}(e') \land \text{addr}(e) = \text{addr}(e') \land e < e' \land \forall e''. (e < e'' \land e'' < e') \implies \neg(\text{iswrite}(e'') \land \text{addr}(e'') = \text{addr}(e'))$$

$$e \xrightarrow{co} e' \iff \text{iswrite}(e) \land \text{iswrite}(e') \land \text{addr}(e) = \text{addr}(e') \land e < e'$$

Now check the properties:

Checking po properties: ...all follow from "have the same program-order relations"
Checking rf properties:
- forall $e, e', e''$. $(e \xrightarrow{rf} e'' \land e' \xrightarrow{rf} e'') \implies e = e'$
- Suppose wlog $e < e'$ then that contradicts the no-intervening-write clause of the construction
- forall $e, e'$. $e \xrightarrow{rf} e' \implies \text{iswrite}(e) \land \text{isread}(e') \land \text{addr}(e) = \text{addr}(e')$
- ...by construction of rf
- forall $e, e'$. $e \xrightarrow{rf} e' \implies \text{value}(e) = \text{value}(e')$
- ...because there are no intervening writes to the same address between $e$ and $e'$, $m(\text{addr}(e))$ remains constant (by induction on that part of the execution trace), and hence is read at $e'$. 
- forall $e$ (isread $e$ & not exists $e'$. $e' \xrightarrow{rf} e$) => 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 $\xrightarrow{co}$ e)
- ...if $e \xrightarrow{co} e$ then $e < e$ but that contradicts the definition of $<$
- forall $e, e', e''$. $(e \xrightarrow{co} e' \land e' \xrightarrow{co} e'') \implies e \xrightarrow{co} e''$
- ...equivalence of iswrite and same-addr, and transitivity of $<$
- forall $e, e'$. $e \xrightarrow{co} e' \implies \text{iswrite}(e) \land \text{iswrite}(e') \land \text{addr}(e) = \text{addr}(e')$
- ...by construction of co
- forall $a$. forall $e, e'$. $(e \leq e' \land \text{iswrite}(e) \land \text{iswrite}(e') \land \text{addr}(e) = a \land \text{addr}(e') = e) \implies e \xrightarrow{co} e' || e' \xrightarrow{co} e$
- ...if $e \leq e'$ then either $e < e'$ or $e' < e$; then in either case construct a co
Now check each of po, rf, co, and fr 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 \( w_0 \), \( w_0 \) co \( w \) & \( w_0 \) rf \( r \)

\[
\begin{align*}
\text{w0} \\
| \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ fooled.
For the right-to-left direction, given an execution witness \( E = \langle rf, co \rangle \) such that acyclic(\( ob \)), where \( ob = (po \cup rf \cup co \cup fr) \), construct a trace \([e_1, \ldots, e_n]\) as an arbitrary linearisation of \( ob \).

By acyclic(\( ob \)), we know if \( e_i ob e_j \) then \( i < j \) (but not the converse).

Construct memory states \( m_i \) inductively along that trace, starting with \( m_0 \), 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 \( m_0 \langle e_1 \rightarrow \rightarrow \rightarrow m_1 \ldots \rightarrow e_n \rightarrow m_n \), it remains to check for each read, say \( r_j \) at index \( j \), that \( m_{j-1}(addr(r_j)) = value(r_j) \)

By the construction of the \( m_i \),

\[
m_{j-1}(addr(r_j)) = value(e_i) \quad \text{where } i \text{ is the largest } i < j \text{ such that iswrite } e_i \& addr e_i = addr r_j, \text{ if there is one}
\]

or \( m_0(addr(r_j)) \) otherwise

In the first case, write \( w_i \) for \( e_i \). We know by the fr lemma that either \( w_i co* rf r_j \) or \( r_j fr w_i \).

Case the latter (\( r_j fr w_i \)): then \( r_j ob w_i \) so \( j < i \), contradicting \( i < j \).

Case the former (\( w_i co* w_k rf r_j \) for some \( k \)):

We know \( i \leq k < j \), so unless \( i = k \) we contradict the "largest"

So \( w_i rf r_j \), so they have the same value

In the second case, there is no \( i < j \) such that iswrite \( e_i \& addr e_i = addr r_j \)

So there is no \( w ob r_j \) such that \( addr w = addr r_j \)

So there is no \( w rf r_j \)

So by the candidate-execution initial-state condition, \( value(r_j) = m_0(addr(r_j)) \)
This lets us take the predicate $\text{acyclic}(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$ 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 $\langle E, \xrightarrow{\text{po}} \rangle$, and
- a candidate execution witness $X = \langle \xrightarrow{\text{rf}}, \xrightarrow{\text{co}} \rangle$ for it,

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

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 [29, 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 [29], and

▶ translating the predicate into SMT constraints, as the isla-axiomatic [31, Armstrong et al.] tool does.

More on these later too.

Note how the construction of an arbitrary linearisation of $\mathbb{ob}$ illustrates some “irrelevant” interleaving in the SC operational model.
Expressing coherence axiomatically, on candidate executions

\[
\text{let } \text{pos} = \text{po} \& \text{loc} \quad (* \text{same-address part of po, aka po-loc} *)
\]

\[
\text{acyclic pos} \mid \text{rf} \mid \text{co} \mid \text{fr} \quad (* \text{coherence check} *)
\]

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 $\text{pos} \mid \text{co} \mid \text{rf} \mid \text{fr}$, 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) [25, 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, \( w = (a:t_0:W x=v) \) and \( a':t_0:D_w x=v \)

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

Binary relations are just sets of pairs.

We write

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

interchangeably.
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):

- \( r \cup s \) the union of \( r \) and \( s \) - \( \{(e, e') \mid e \in r \cup s \} \)
- \( r \cap s \) the intersection of \( r \) and \( s \) - \( \{(e, e') \mid e \in r \cap s \} \)
- \( r ; s \) the composition of \( r \) and \( s \) - \( \{(e, e'') \mid \exists e'. e \in r \cap s \} \)
- \( r \setminus s \) \( r \) minus \( s \) - \( \{(e, e') \mid e \in r \setminus s \} \)
- \( [S] \) the identity on some set \( S \) of events - \( \{(e, e) \mid e \in S \} \)
- \( S \times S' \) the product of sets \( S \) and \( S' \) - \( \{(e, e') \mid e \in S \times S' \} \)
- \( \text{loc} \) same-location, events at the same address - \( \{(e, e') \mid \text{addr}(e) = \text{addr}(e') \} \)
- \( \text{int} \) internal, events of the same thread - \( \{(e, e') \mid \text{thread}(e) = \text{thread}(e') \} \)
- \( \text{ext} \) external, events of different thread - \( \{(e, e') \mid \text{thread}(e) \neq \text{thread}(e') \} \)

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

\[
\begin{align*}
\text{rfe} & = \text{rf} \& \text{ext} = \{(e, e') \mid e \text{ rf } e' \land \text{thread}(e) \neq \text{thread}(e')\} \\
\text{rfi} & = \text{rf} \& \text{int} = \{(e, e') \mid e \text{ rf } e' \land \text{thread}(e) = \text{thread}(e')\}
\end{align*}
\]
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 $a:t:D_w x=v$ 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 $a:t:D_w x = v$ 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 $a:t:D_w x = v$ 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 $a:t:W x = v$ 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 $a:\text{t:D}_w x\leftarrow v$ 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 $a:\text{t:W}_x\leftarrow v$ 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:

\textbf{acyclic pos | rf | co | fr}  (* coherence check, where pos = po & loc *)
Back to coherence, axiomatically

Recall we expressed coherence axiomatically as:

\[ \text{acyclic} \text{ pos } | \text{ rf } | \text{ co } | \text{ fr} \quad (* \text{coherence, where pos = po \& loc} \ast) \]

It can be useful to think of this as the combination of a check that each thread locally preserves coherence, i.e. \( \text{rfi}, \text{ coi}, \text{ and fri all go forwards in program order:} \)

\[ \text{acyclic} \text{ pos } | \text{ rfi} \]
\[ \text{acyclic} \text{ pos } | \text{ coi} \]
\[ \text{acyclic} \text{ pos } | \text{ fri} \]

and a check that these intra-thread orderings are compatible with each other and the inter-thread interactions:

\[ \text{acyclic} \text{ pos } | \text{ coe } | \text{ rfe } | \text{ fre} \]
## Basic coherence shapes again

<table>
<thead>
<tr>
<th>CoRW1</th>
<th>CoWW</th>
<th>CoWR0</th>
<th>CoRR</th>
<th>CoRW2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Thread 0" /></td>
<td><img src="" alt="Thread 0" /></td>
<td><img src="" alt="Thread 0" /></td>
<td><img src="" alt="Thread 0" /></td>
<td><img src="" alt="Thread 0" /></td>
</tr>
<tr>
<td><img src="pos" alt="rfi" /></td>
<td><img src="coi" alt="pos" /></td>
<td><img src="frf" alt="pos" /></td>
<td><img src="pos" alt="rfe" /></td>
<td><img src="coe" alt="pos" /></td>
</tr>
<tr>
<td><img src="pos" alt="b:Wx=1" /></td>
<td><img src="pos" alt="b:Wx=2" /></td>
<td><img src="pos" alt="b:Rx=0" /></td>
<td><img src="pos" alt="b:Rx=0" /></td>
<td><img src="pos" alt="c:Wx=2" /></td>
</tr>
</tbody>
</table>

### 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:Wx=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 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 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 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:Wx=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.

Contents 2.6 x86: x86-TSO axiomatic model
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 $D(w)$.

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.
That leads us to:

```plaintext
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

SB

<table>
<thead>
<tr>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>a: W x=1</td>
</tr>
<tr>
<td>b: R y=0</td>
</tr>
<tr>
<td>d: R x=0</td>
</tr>
</tbody>
</table>

| Thread 1 |
| a: R y=1 |
| b: W y=1 |
| d: W x=1 |

| LB |
| Allowed |
| Thread 0 |
| a: R x=1 |
| b: W y=1 |
| d: W x=1 |

| Thread 1 |
| a: R y=1 |
| b: W y=1 |
| d: W x=1 |

| MP |
| Allowed |
| Thread 0 |
| a: W x=1 |
| b: W y=1 |
| e: R x=0 |
| d: W x=1 |

| Thread 1 |
| a: R y=1 |
| b: W y=1 |
| d: W x=1 |

| SB+rfi-pos |
| Allowed |
| Thread 0 |
| a: W x=1 |
| b: R x=1 |
| c: R y=0 |
| d: W y=1 |
| e: R y=1 |
| f: R x=0 |

| Thread 1 |
| a: W x=1 |
| b: W y=1 |
| e: R x=0 |
| d: W x=1 |

| LB |
| Allowed |
| Thread 0 |
| a: R x=1 |
| b: W y=1 |
| e: R x=0 |
| d: W x=1 |

| Thread 1 |
| a: R y=1 |
| b: W y=1 |
| d: W x=1 |

| MP |
| Allowed |
| Thread 0 |
| a: W x=1 |
| b: W y=1 |
| e: R x=0 |
| d: W x=1 |

| Thread 1 |
| a: R y=1 |
| b: W y=1 |
| d: W x=1 |

| WRC |
| Allowed |
| Thread 0 |
| a: W x=1 |
| b: R x=1 |
| c: W y=1 |
| e: R x=0 |

| Thread 1 |
| a: W x=1 |
| b: W y=1 |
| d: W x=1 |

| Thread 2 |
| a: W x=1 |
| b: W y=1 |
| c: W y=1 |
| e: R x=0 |

| 2+2W |
| Allowed |
| Thread 0 |
| a: W x=2 |
| b: W y=1 |
| e: R x=0 |
| d: W x=1 |

| Thread 1 |
| a: W x=2 |
| b: W y=1 |
| d: W x=1 |

| Thread 2 |
| a: W x=2 |
| b: W y=1 |
| c: W y=2 |
| e: R x=0 |

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

Contents 2.6 x86: x86-TSO axiomatic model
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 $e < e' \iff \exists i, j. e = e_i \land e' = e_j \land i < j$

Say an x86-TSO candidate pre-execution is $\langle E, po \rangle$ 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 $\langle rf, co \rangle$ satisfying the same conditions as for SC.

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

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

**Theorem 6.** For any candidate pre-execution $\langle E, \text{po} \rangle$, 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 $X = \langle \text{rf}, \text{co} \rangle$ for $\langle E, \text{po} \rangle$ such that $\text{acyclic}(\text{pos} \cup \text{rf} \cup \text{co} \cup \text{fr})$ and $\text{acyclic ob}$. 
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.

$E = \{ e \mid e \in \{ e_1, \ldots, e_n \} \land (\text{iswrite}(e) \lor \text{isread}(e)) \}$

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 $PW(r) = \{ w \mid w \in E \land \text{iswrite}(w) \land \text{addr}(w) = \text{addr}(r) \}$

\[
\begin{array}{ll}
\text{w rf r} & \iff \text{isread}(r) \land w \in PW(r) \land ( \\
& \left( \begin{array}{l}
(* \text{from-buffer, same-thread } *) \\
\left( \begin{array}{l}
(* \text{w in buffer } *) \\
\left( \begin{array}{l}
(* \text{no intervening in buffer } *) \\
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array}
\]

\[
\begin{array}{ll}
& \lor \\
& \left( \begin{array}{l}
(* \text{from-memory, any-thread } *) \\
\left( \begin{array}{l}
(* \text{w in memory } *) \\
\left( \begin{array}{l}
(* \text{no intervening in buffer } *) \\
(* \text{no intervening in memory } *) \\
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array}
\]

For co, say $w \text{ co } w'$ if $\text{iswrite}(w) \land \text{iswrite}(w') \land \text{addr}(w) = \text{addr}(w') \land D(w) < D(w')$
Check the candidate execution well-formedness properties hold
...the $w$ rf $r$ implies $\text{value}(r) = \text{value}(w)$ condition essentially checks correctness of the rf characterisation

For acyclic ob, check each $(e, e')$ in $\text{po}\backslash([W];\text{po};[R]) | \text{rfe} | \text{coe} | \text{fre}$ 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 $\text{pos}|\text{rf}|\text{co}|\text{fr}$, 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:

\[
\begin{align*}
  w & \mapsto [] \\
  r & \mapsto [r] \text{ if } r \text{ reads from memory} \\
  & \mapsto [\ ] \text{ if } r \text{ reads from its thread's buffer} \\
  a:t:D_w x = v & \mapsto [w]@[r | r \text{ reads from } w \text{ via buffer, ordered by } <] \\
\end{align*}
\]

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] (* add W/R pairs separated in po by an MFENCE *)
    | [W]; po; [R & X]          (* add W/R pairs where at least one is from an atomic RMW *)
    | [W & X]; po; [R]          (* ...X identifies such accesses *)

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

(* Internal visibility requirement *)
acyclic pos | fr | co | rf as internal (* coherence check *)

(* Atomicity requirement *)      (* no fre;coe between the read and write of an atomic RMW *)
empty rmw & (fre;coe) as atomic   (* rmw relates the reads and writes of each atomic RMW instruction*)

(* External visibility requirement *)
acyclic ob                      (* external check *)
Summary of axiomatic-model sets and relations

The data of a candidate pre-execution:

- a set \( E \) of events
- \( \text{po} \subseteq E \times E \), program-order

The data of a candidate execution witness:

- \( \text{rf} \subseteq W \times R \), reads-from
- \( \text{co} \subseteq W \times W \), coherence

Subsets of \( E \):

- \( R \) all read events
- \( W \) all write events
- \( \text{MFENCE} \) all mfence events
- \( X \) all locked-instruction accesses

Derived relations, generic:

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

Derived relations, specific to x86 model:

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

Contents 2.6 x86: x86-TSO axiomatic model 224
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.
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.\textit{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.\textit{x86-TSO-ax:} does the order of “reordered” match \textit{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” \textit{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 $\text{po|rf|co|fr}$

(“critical cycles” [39])
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"
```

---

**SB**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W x = 1</td>
<td>c: W y = 1</td>
</tr>
<tr>
<td>po</td>
<td>fre</td>
</tr>
<tr>
<td>fre</td>
<td>po</td>
</tr>
<tr>
<td>b: R y = 0</td>
<td>d: R x = 0</td>
</tr>
</tbody>
</table>

**Documentation:** [http://diy.inria.fr/doc/gen.html](http://diy.inria.fr/doc/gen.html)
For small tests, we can be exhaustive, in various ways

e.g. the earlier coherence tests

**CoRW1**
Thread 0
a:R \(x=1\)
\(\text{rf}\)
\(\text{po}\)
b:W \(x=1\)

**CoWW**
Thread 0
a:W \(x=1\)
\(\text{po}\)
b:W \(x=2\)

**CoWR0**
Thread 0
a:W \(x=1\)
\(\text{co}\)
b:W \(x=2\)

**CoRR**
Thread 0
a:W \(x=1\)
\(\text{co}\)
b:R \(x=1\)
\(\text{rf}\)
c:R \(x=0\)

**CoRW2**
Thread 0
a:W \(x=1\)
\(\text{rf}\)
b:R \(x=1\)
\(\text{co}\)
c:W \(x=2\)
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:

<table>
<thead>
<tr>
<th>SB</th>
<th>MP</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a: W x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c: W y = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f r e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b: R y = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d: R x = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>2+2W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a: W x = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b: W y = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f r e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c: W y = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d: R x = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Thread 1 |
| a: R x = 1 |
| p o |
| b: W y = 1 |
| r f |
| c: R y = 1 |
| p o |
| d: W x = 1 |
| r f |

| Thread 0 |
| a: W x = 2 |
| p o |
| b: W y = 1 |
| c o e |
| d: R x = 0 |
| r f |

| Thread 1 |
| a: W x = 2 |
| p o |
| b: W y = 1 |
| c: W y = 2 |
| c o e |
| d: W x = 1 |
| c o e |
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

```bash
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

Contents
3 Validating models: 241
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*

<table>
<thead>
<tr>
<th>Kind</th>
<th>run-hw.log</th>
<th>run-rmem.log</th>
<th>run-herd.log</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+2W</td>
<td>[x=1; y=1;] == ==</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>[x=1; y=2;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[x=2; y=1;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LB</td>
<td>[0:rax=0; 1:rax=0;] == ==</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>[0:rax=0; 1:rax=1;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0:rax=1; 1:rax=0;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP</td>
<td>[1:rax=0; 1:rbx=0;] == ==</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>[1:rax=0; 1:rbx=1;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[1:rax=1; 1:rbx=1;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB</td>
<td>[0:rax=0; 1:rax=0;] == ==</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ok</td>
<td>[0:rax=0; 1:rax=1;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0:rax=1; 1:rax=0;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0;rax=1; 1:rax=1;]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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 \texttt{ob}), remove it from the \texttt{-safe} list and add it to \texttt{-relax}, then \texttt{diy7} will by default generate cycles of exactly one relaxed edge and some safe edges.

\texttt{x86-rfi.conf}

\begin{verbatim}
#rfi x86 conf file
-arch X86
-nprocs 4
-size 6
-name rfi
-safe PosR* PodR* PodWW PosWW Rfe Wse Fre FencesWR FencedWR
-relax Rfi
\end{verbatim}

\texttt{x86-podwr.conf}

\begin{verbatim}
#podwr x86 conf file
-arch X86
-nprocs 4
-size 6
-name podwr
-safe Fre
-relax PodWR
\end{verbatim}

From \url{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
Armv8-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.


- 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](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:

<table>
<thead>
<tr>
<th>Year</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>A.a</td>
<td>Armv8.0-A (first non-confidential beta)</td>
</tr>
<tr>
<td>2016</td>
<td>A.k</td>
<td>Armv8.0-A (EAC)</td>
</tr>
<tr>
<td>2017</td>
<td>B.a</td>
<td>Armv8.1-A (EAC), Armv8.2-A (Beta) (simplification to MCA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>2020</td>
<td>F.c</td>
<td>Armv8.6-A (initial EAC)</td>
</tr>
</tbody>
</table>
IBM Power architecture

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

<table>
<thead>
<tr>
<th>Processor</th>
<th>Architecture</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER5</td>
<td>Power ISA 2.03</td>
<td>2004</td>
</tr>
<tr>
<td>POWER6</td>
<td>Power ISA 2.03</td>
<td>2007</td>
</tr>
<tr>
<td>POWER7</td>
<td>Power ISA 2.06</td>
<td>2010</td>
</tr>
<tr>
<td>POWER8</td>
<td>Power ISA 2.07</td>
<td>2014</td>
</tr>
<tr>
<td>POWER9</td>
<td>Power ISA 3.0B</td>
<td>2017</td>
</tr>
<tr>
<td>POWER10</td>
<td></td>
<td>2021?</td>
</tr>
</tbody>
</table>

POWER10: 240 hw threads/socket
POWER 8: up to 192 cores, each with up to 8 h/w threads [https://en.wikipedia.org/wiki/POWER8](https://en.wikipedia.org/wiki/POWER8)
Power7: IBM’s Next-Generation Server Processor Kalla, Sinharoy, Starke, Floyd
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 [36]
- The RISC-V Instruction Set Manual Volume II: Privileged Architecture [37]
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 describe 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.
Coherence
Still all forbidden
Out-of-order accesses
Out-of-order pod WW and pod RR: MP (Message Passing)

MP

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W x=1</td>
<td>c: R y=1</td>
</tr>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>fr</td>
<td>rf</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: W y=1</td>
<td>d: R x=0</td>
</tr>
</tbody>
</table>

Allowed:

- 1: X0=1; 1: X2=0;
- Arm: YYYYY YYYYY
- Power: Y RISC-V:N

Initial state:

- 0: X3=y; 0: X1=x;
- 1: X3=x; 1: X1=y;

MP AArch64

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV W0, #1</td>
<td>LDR W0, [X1] //c</td>
</tr>
<tr>
<td>STR W0, [X1] //a</td>
<td>LDR W2, [X3] //d</td>
</tr>
<tr>
<td>MOV W2, #1</td>
<td></td>
</tr>
<tr>
<td>STR W2, [X3] //b</td>
<td></td>
</tr>
</tbody>
</table>

Allowed:

- 1: X0=1; 1: X2=0;
Out-of-order pod WW and pod RR: MP (Message Passing)

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).
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 513. Key: Arm: abcdefghij 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

<table>
<thead>
<tr>
<th>model</th>
<th>experimental observation</th>
<th>conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed</td>
<td>Y</td>
<td>ok</td>
</tr>
<tr>
<td>Allowed</td>
<td>N</td>
<td>ok, but model is looser than hardware (or testing not aggressive)</td>
</tr>
<tr>
<td>Forbidden</td>
<td>Y</td>
<td>model not sound w.r.t. hardware (or hardware bug)</td>
</tr>
<tr>
<td>Forbidden</td>
<td>N</td>
<td>ok</td>
</tr>
</tbody>
</table>
Out-of-order pod WR: SB ("Store Buffering")

**SB**

- **Thread 0**
  - a: $Wx = 1$
  - b: $Ry = 0$

- **Thread 1**
  - c: $Wy = 1$
  - d: $Rx = 0$

**Allowed**

- $Wx = 1$
- $Wy = 1$
- $Rx = 0$
- $Ry = 0$

**SB AArch64**

Initial state:

- $0: X3 = y$
- $0: X1 = x$
- $1: X3 = x$
- $1: X1 = y$

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV W0, #1</td>
<td>MOV W0, #1</td>
</tr>
<tr>
<td>STR W0, [X1]</td>
<td>STR W0, [X1]</td>
</tr>
<tr>
<td>LDR W2, [X3]</td>
<td>LDR W2, [X3]</td>
</tr>
</tbody>
</table>

Allowed:

- $0: X2 = 0$
- $1: X2 = 0$

**Arm:** YYYYY YYYYY

**Power:** Y RISC-V: N

**Contents**

4.1.2 **Armv8-A, IBM Power, and RISC-V**: Phenomena: Out-of-order accesses
Out-of-order pod WR: SB (“Store Buffering”)

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

### SB and Allowed

<table>
<thead>
<tr>
<th>SB</th>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>a: W x=1</td>
<td>c: W y=1</td>
</tr>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>rf</td>
<td>fre</td>
</tr>
<tr>
<td>b: R y=0</td>
<td>d: R x=0</td>
</tr>
<tr>
<td>rf</td>
<td>rf</td>
</tr>
</tbody>
</table>

### Initial State

Initial state: 0: X3=y; 0: X1=x; 1: X3=x; 1: X1=y;

### SB AArch64

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV W0,#1</td>
<td>MOV W0,#1</td>
</tr>
<tr>
<td>STR W0,[X1]</td>
<td>STR W0,[X1]</td>
</tr>
<tr>
<td>LDR W2,[X3]</td>
<td>LDR W2,[X3]</td>
</tr>
</tbody>
</table>

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

### Microarchitecture

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

### Contents

4.1.2 Armv8-A, IBM Power, and RISC-V: Phenomena: Out-of-order accesses
## Out-of-order pod RW: LB ("Load Buffering")

<table>
<thead>
<tr>
<th>LB</th>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>a: R\text{X}=1 &amp; po &amp; c: R\text{Y}=1 &amp; po</td>
<td></td>
</tr>
<tr>
<td>b: W\text{Y}=1 &amp; \text{rfe} &amp; d: W\text{X}=1 &amp; \text{rfe}</td>
<td></td>
</tr>
</tbody>
</table>

### AArch64

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR W0, [X1] //a</td>
<td>LDR W0, [X1] //c</td>
</tr>
<tr>
<td>MOV W2, #1</td>
<td>MOV W2, #1</td>
</tr>
<tr>
<td>STR W2, [X3] //b</td>
<td>STR W2, [X3] //d</td>
</tr>
</tbody>
</table>

**Allowed:** 0: X0=1; 1: X0=1;

**Arm:** NNNNN NNNNN

**Power:** N RISC-V: N

---

### Contents

4.1.2 **Armv8-A, IBM Power, and RISC-V:** Phenomena: Out-of-order accesses
Out-of-order pod RW: LB ("Load Buffering")

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

Contents 4.1.2 Armv8-A, IBM Power, and RISC-V: Phenomena: Out-of-order accesses
Out-of-order pod RW: LB ("Load Buffering")

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.

Contents 4.1.2 Armv8-A, IBM Power, and RISC-V: Phenomena: Out-of-order accesses
Out-of-order pod WW again: 2+2W

<table>
<thead>
<tr>
<th>2+2W</th>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>a: \text{W} x=2</td>
<td>c: \text{W} y=2</td>
</tr>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>b: \text{W} y=1</td>
<td>d: \text{W} x=1</td>
</tr>
</tbody>
</table>

2+2W AArch64

Initial state: 0: \text{X3}=y; 0: \text{X1}=x;
1: \text{X3}=x; 1: \text{X1}=y;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV \text{W0}, #2</td>
<td>MOV \text{W0}, #2</td>
</tr>
<tr>
<td>STR \text{W0}, [\text{X1}]</td>
<td>STR \text{W0}, [\text{X1}]</td>
</tr>
<tr>
<td>MOV \text{W2}, #1</td>
<td>MOV \text{W2}, #1</td>
</tr>
<tr>
<td>STR \text{W2}, [\text{X3}]</td>
<td>STR \text{W2}, [\text{X3}]</td>
</tr>
</tbody>
</table>

Allowed: \text{y}=2; \text{X}=2;

Arm: YYYY YYYYY
YNYYY NY
Power: RISC-V: N
Out-of-order pod WW again: 2+2W

### 2+2W

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W x = 2</td>
<td>a: W x = 2</td>
</tr>
<tr>
<td>b: W y = 1</td>
<td>c: W y = 2</td>
</tr>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>coe</td>
<td>coe</td>
</tr>
<tr>
<td>d: W x = 1</td>
<td></td>
</tr>
</tbody>
</table>

**Allowed**:

- y = 2; x = 2;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV W0, #2</td>
<td>MOV W0, #2</td>
</tr>
<tr>
<td>STR W0, [X1] //a</td>
<td>STR W0, [X1] //c</td>
</tr>
<tr>
<td>MOV W2, #1</td>
<td>MOV W2, #1</td>
</tr>
<tr>
<td>STR W2, [X3] //b</td>
<td>STR W2, [X3] //d</td>
</tr>
</tbody>
</table>

**Initial state**:

- 0: X3 = y; 0: X1 = x;
- 1: X3 = x; 1: X1 = y;

**AArch64**

**Allowed**:

- y = 2; x = 2;

**Contents**

- Armv8-A, IBM Power, and RISC-V: Phenomena: Out-of-order accesses

**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+fens+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.

Contents 4.1.3 Armv8-A, IBM Power, and RISC-V: Phenomena: Barriers
Weaker Barriers

Enforcing ordering can be expensive, especially write-to-read ordering, so each architecture also provides various weaker barriers:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armv8-A</td>
<td>dmb ld</td>
<td>read-to-read and read-to-write</td>
</tr>
<tr>
<td></td>
<td>dmb st</td>
<td>write-to-write</td>
</tr>
<tr>
<td>Power</td>
<td>lwsync</td>
<td>read-to-read, write-to-write, and read-to-write</td>
</tr>
<tr>
<td></td>
<td>eieio</td>
<td>write-to-write</td>
</tr>
<tr>
<td>RISC-V</td>
<td>fence pred, succ</td>
<td>$\text{pred, succ} \subseteq_{\text{nonempty}} {r, w}$</td>
</tr>
</tbody>
</table>

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

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.

| Initial state: x=0; y=z; z=2; |
|---|---|
| Thread 0 | Thread 1 |
| x=1; | r1=y; |
| y=&x; | r2=*r1; |
| Forbidden: 1:r1=y; 1:r2=0; |

---

**MP+fen+addr.real Forbidden**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:W x=1</td>
<td>d:R y=x</td>
</tr>
<tr>
<td>c:W y=x</td>
<td>e:R x=0</td>
</tr>
</tbody>
</table>

**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;**

**MP+dmb.sy+addr.real AArch64**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR X0, [X1] //a LDR X2, [X1] //d</td>
<td></td>
</tr>
<tr>
<td>DMB SY //b LDR X3, [X2] //e</td>
<td></td>
</tr>
<tr>
<td>STR X1, [X3] //c</td>
<td></td>
</tr>
</tbody>
</table>

| Forbidden: 1:X2=x; 1:X3=0; |
Enforcing order with dependencies: natural vs artificial

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.
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

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

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 \( r1 \neq 1 \) to after the second read. (Some ambiguity in Arm, [34, B2.3.2]?)
Enforcing order with dependencies: read-to-read ctrliflen dependencies

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

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:

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

<table>
<thead>
<tr>
<th>Initial state: x=0; y=0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>x=1;</td>
</tr>
<tr>
<td>y=1;</td>
</tr>
</tbody>
</table>

Forbidden: 2:r3=0;

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

**WRC x86**

<table>
<thead>
<tr>
<th>Initial state: 1:rax=0; 2:rax=0; 2:rbx=0; y=0; x=0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>movq $1, (x) //a</td>
</tr>
<tr>
<td>movq $1, (y) //c</td>
</tr>
</tbody>
</table>

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:

```
li r1,1
stw r1,0(r2)

Thread 0
lwz r1,0(r2)
R x=1

a: W x=1

Thread 1
lwz r1,0(r2)
R y=1

b: R x=1

Thread 2
lwz r1,0(r2)
R x=0
d: R y=1
c: W y=1
e: R x=0

d: R y=1
```

- IBM POWER: 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 [21, Pulte, Flur, Deacon,...].
Cumulative barriers

In a non-MCA architecture, e.g. current Power, one needs *cumulative* barriers to support iterated message-passing:

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 and f are concerned.
Cumulative barriers, on the right

Cumulative barriers also ensure that chains of reads-from and dependency edges after such a barrier are respected:

Thread 0
- a: W x = 1
- sync
- c: W y = 1

Thread 1
- d: R y = 1
- data
- e: W z = 1

Thread 2
- f: R z = 1
- addr
- g: R x = 0

 ISA2+sync+data+addr Power
Initial state: 0: r4 = y; 0: r2 = x; 1: r4 = z; 1: r2 = y;
2: r5 = x; 2: r2 = z;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>li r1,1</td>
<td>lwz r1,0(r2) //d</td>
<td>lwz r1,0(r2) //f</td>
</tr>
<tr>
<td>stw r1,0(r2) //a</td>
<td>xor r3, r1, r1</td>
<td>xor r3, r1, r1</td>
</tr>
<tr>
<td>sync //b</td>
<td>addi r3, r3, 1</td>
<td>lwzx r4, r3, r5 //g</td>
</tr>
<tr>
<td>li r3,1</td>
<td>stw r3, 0(r4) //e</td>
<td></td>
</tr>
<tr>
<td>stw r3, 0(r4) //c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
A strong cumulative barrier is also needed to forbid IRIW in a non-MCA architecture:

\begin{verbatim}
Thread 0
   a: W x=1
   rf
   b: R x=1
   sync
   d: R y=0

Thread 1
   li r1,1
   stw r1,0(r2)
   W y=1
   e: R y=1
   rf
   f: sync
   lwz r3,0(r4)
   R x=0

Thread 2
   li r1,1
   stw r1,0(r2)
   W x=1
   a:
   rf
   b: sync
   lwz r3,0(r4)
   R y=0
d:

Thread 3
   li r1,1
   stw r1,0(r2)
   W y=1
   e:
   rf
   f: sync
   lwz r3,0(r4)
   R x=0

Initial state: 0:r2=x; 1:r4=y; 1:r2=x; 2:r2=y; 3:r4=x; 3:r2=y;
IRIW+syncs Power
Forbidden: 1:r1=1; 1:r3=0; 3:r1=1; 3:r3=0;
\end{verbatim}

(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

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.

\[
\text{STR <Xt>, [<Xn|SP>], #<simm>} \quad \text{(post-index)}
\]
\[
\text{STR <Xt>, [<Xn|SP>, #<simm>]!} \quad \text{(pre-index)}
\]

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;

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”:

```
MOV X0,#1
STR X0,[X1]
W x=1
a:
LDR X2,[X1]
R x=1
b:
EOR X3,X2,X2
LDR X4,[X5,X3]
R y=0
c:
Thread 0
po
addr
MOV X0,#1
STR X0,[X1]
W y=1
d:
LDR X2,[X1]
R y=1
e:
EOR X3,X2,X2
LDR X4,[X5,X3]
R x=0
f:
Thread 1
po
rf
rf
rf
fr
```

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.

Writtes 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.

(PPOCA and PPOAA are nicknames for MP+fen+ctrl-rfi-addr and MP+fen+addr-rfi-addr)
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 an 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.

<table>
<thead>
<tr>
<th>POWER</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>PowerG5</td>
</tr>
<tr>
<td>LB+addrs+WW Forbid</td>
<td>0/30G</td>
</tr>
<tr>
<td>LB+datas+WW Allow</td>
<td>0/30G</td>
</tr>
<tr>
<td>LB+addrs+RW Forbid</td>
<td>0/3.6G</td>
</tr>
</tbody>
</table>

*Contents 4.1.6 Armv8-A, IBM Power, and RISC-V: Phenomena: Further thread-local subtleties*
Intra-instruction ordering of address and data inputs to a write

To let the later writes \((c,f)\) in \(LB+\text{datas}+\text{WW}\) 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.r.t. that too.)

\[
\text{STR } <Xt>,[<Xn|SP>],#<\text{simm}> \quad \text{STR } <Xt>,[<Xn|SP>,#<\text{simm}>]!
\]

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)

<table>
<thead>
<tr>
<th></th>
<th>62</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>DS</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{EA} \leftarrow (\text{RA}) + \text{EXTS}(\text{DS} || 0b00) \\
\text{MEM(\text{EA, 8})} \leftarrow (\text{RS}) \\
\text{RA} \leftarrow \text{EA}
\]
Satisfying reads from the same write: RSW and RDW

Coherence suggests that reads from the same address must be satisfied 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 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

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.
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-A and RISC-V are MCA, so no variants of IRIW+addrs are allowed there.

It is moreover highly desirable for an architecture specification to be symmetric w.r.t. permutation of threads.
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 [35, 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 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 [21, Pulte, Flur, Deacon, et al.] for more details, and [18, Flur et al.] for discussion of Armv8 release/acquire in the previous non-MCA architecture

Together with the Arm architecture reference manual [34, 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
- **Power**: load and reserve / store conditional
- **RISC-V**: load-reserved / store-conditional

LDXR / STXR
lwarx / stwcx.
LR.D / SC.D
LL/SC atomic increment

Here are two concurrent increments of x, expressed with exclusives.

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 [21, Pulte, Flur, Deacon, et al.] (especially its supplementary material
https://www.cl.cam.ac.uk/~pes20/armv8-mca/), and [18, Flur et al.] for Armv8-A
load-exclusive/store-exclusives.

Together with the vendor manuals:

- Power: [35, §1.7.4 Atomic Update]
- Arm: [34, Ch.B2 The AArch64 Application Level Memory Model]
- RISC-V: [36, 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
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**  
Thread 0  
a: W x = 0x1110  
Thread 1  
b: R x + 1 = 0x11  
c: R x = 0

**Allowed**

**MP+si+po**  
AArch64

**Initial state:**  
0: X1 = 0x1110; 0: X0 = x;  
1: X0 = x; x = 0x0;

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRH W1, [X0] //a</td>
<td>LDRB W1, [X0, #1] //b</td>
</tr>
<tr>
<td>LDRB W2, [X0] //c</td>
<td></td>
</tr>
</tbody>
</table>

**Allowed:**  
1: X1 = 0x11; 1: X2 = 0x0;

---

Contents  
4.2.4 Armv8-A, IBM Power, and RISC-V: More features: Mixed-size
Mixed-size: further details

See [20, 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
Instruction-set architecture (ISA)

- **ARMv8-A**: Historically only pseudocode. Arm transitioned internally to mechanised ASL [40, 41, 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)
Custom language for expressing the sequential behaviour of instructions (including decode, address translation, etc.) \cite{Armstrong:2013,Gray:2013}

- 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...
Contents 4.3 Armv8-A, IBM Power, and RISC-V: ISA semantics 334
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)
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 [42]

Contents

4.3.1 Armv8-A, IBM Power, and RISC-V: ISA semantics: Integrating ISA and axiomatic models 338
Armv8-A/RISC-V operational model
For more details, see [21, Pulte, Flur, Deacon, et al.] (especially its supplementary material https://www.cl.cam.ac.uk/~pes20/armv8-mca/), together with [22, 20, 18, 12, 8]

Together with the RISC-V manual:

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.

- 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.
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 make progress independently.
- 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.
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+f\text{en}+\text{ctrl}**
(with “real” control dependency)

```
Thread 0
a: W x=1
fen

c: W y=1

Thread 1
d: R y=1
ctrl

e: R x=0
```

**Allowed.** The barrier orders the writes, but the control dependency is weak: e can be speculatively fetched and satisfied early (rmem web UI).
How do instructions work?

Each instruction is specified as an imperative program.

For example:

```sail
function clause execute ( LoadRegister(n,t,m,acctype,memop, ...) ) = {
(\&\&|64) offset := ExtendReg(m, extend_type, shift);
(\&\&|64) address := 0;
(\&\&|:\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
      ...
```
How do instructions work? Each instruction is specified as an imperative Sail program. For example:

```sail
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 *)

Contents

4.4 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V operational model
Instruction instance states

- **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)

```haskell
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 (\(\text{Barrier}(\text{barrier\_kind}, \text{next\_state}')\)) can be committed if:

1. all po-previous conditional branch instructions are finished;
2. \((BO)\) if \(i\) is a \(\text{dmb sy}\) 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
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 \((\text{Write} \_ \text{reg}(\text{reg\_name}, \text{reg\_value}, \text{next\_state}'))\) can do the register write.
Register write

Action:

1. record \texttt{reg\_name} with \texttt{reg\_value} and \texttt{write\_deps} in \texttt{i.reg\_writes}; and
2. update the state of \texttt{i} to Plain \texttt{next\_state'}.

where \texttt{write\_deps} is the set of all \texttt{read\_sources} from \texttt{i.reg\_reads} . . .

\texttt{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 \( \text{Read}_\text{reg}(\text{reg}_\text{name}, \text{read}_\text{cont}) \) 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.

![Diagram showing the sequence of instruction instances](image)

- "does not write \text{reg}_\text{name}"

Contents 4.4 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V operational model 368
Register read

Let \textit{read\_source} be the write to \textit{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 \textit{reg\_value} be its value.

\textbf{Action:}

1. Record \textit{reg\_name}, \textit{read\_source}, and \textit{reg\_value} in \(i\\text{.reg\_reads}\); and
2. update the state of \(i\) to Plain (\textit{read\_cont(reg\_value)}).

\begin{itemize}
\item \texttt{does not write reg\_name}
\end{itemize}
Example: address dependencies

Forbidden. The barrier orders the writes, the address dependency prevents executing e before d (rmem web UI).

Contents 4.4 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V operational model
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.

---

**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. . . .
Action:

1. record the instruction as finished, i.e., set \( \text{finished} \) to \( \text{true} \); 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 \( \text{instruction\_tree} \).
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 \( \text{write_cont} \), \( i \) 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 \( \text{dmb sy} \) 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 \( \text{write_cont} \) 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. \((\text{CO})\) 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.

CoWR

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W x=1 co</td>
<td>c: W x=2</td>
</tr>
<tr>
<td>fr po</td>
<td>rf</td>
</tr>
</tbody>
</table>

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 $c$. 

Contents 4.4 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V operational model
Write forwarding on a speculative branch

Contents 4.4 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V operational model 383
Write forwarding on a speculative branch

PPOCA

Thread 0
a: W x=1
fen

c: W y=1

Thread 1
d: R y=1

f: R z=1

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.

![Diagram](image-url)
Write forwarding on a speculative branch

PPOCA allowed. (rmem web UI)
Write forwarding on a speculative branch

**PPOCA**

- **Thread 0**
  - a: \( Wx = 1 \)
  - c: \( Wy = 1 \)

- **Thread 1**
  - d: \( Ry = 1 \)
  - e: \( Wz = 1 \)

**PPOAA**

- **Thread 0**
  - a: \( Wx = 1 \)

- **Thread 1**
  - d: \( Ry = 1 \)

---

**PPOCA allowed.** (rmem web UI)

**PPOAA forbidden.**
Armv8-A/RISC-V axiomatic model
For more details, see [21, 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: [34, Ch.B2 The AArch64 Application Level Memory Model]
(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)

\begin{align*}
\text{acyclic} & \ pos \mid fr \mid co \mid rf \quad (* \text{coherence check} *) \\
\text{let obs} &= rfe \mid fre \mid coe \quad (* \text{Observed-by} *) \\
\text{let dob} &= addr \mid data \mid \text{ctrl}; [W] \\
&\quad | \text{addr}; po; [W] \\
&\quad | (\text{ctrl} \mid data); \text{coi} \quad (* \text{Think ‘coi’ (globally equivalent)} *) \\
&\quad | (addr \mid data); rfi \\
&\quad \ldots \\
\text{let bob} &= po; [dmb.sy]; po \quad (* \text{Barrier-ordered-before} *) \\
&\quad \ldots \\
\text{let ob} &= obs \mid dob \mid aob \mid bob \quad (* \text{Ordered-before} *) \\
\text{acyclic} & \ ob \quad (* \text{external check} *)
\end{align*}
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

Forbidden. Each edge of the cycle is included in ob.
Example: speculative execution

\[
\begin{align*}
\text{Thread 0} & \\
a: W x = 1 & \\
fen & \\
c: W y = 1 & \\
\end{align*}
\]

\[
\begin{align*}
\text{Thread 1} & \\
d: R y = 1 & \\
\text{ctrl} & \\
e: R x = 0 & \\
\end{align*}
\]

\[
\text{acyclic} \ pos \mid fr \mid co \mid rf
\]

\[
\text{let} \ obs = rfe \mid fre \mid coe
\]

\[
\text{let} \ dob = \text{addr} \mid \text{data} \\
\mid \text{ctrl}; \ [W] \\
\mid \text{addr}; \ po; \ [W] \\
\mid (\text{ctrl} \mid \text{data}); \ coi \\
\mid (\text{addr} \mid \text{data}); \ rfi \\
\ldots
\]

\[
\text{let} \ bob = \text{po}; \ [\text{dmb.sy}]; \ po \\
\ldots
\]

\[
\text{let} \ ob = obs \mid dob \mid aob \mid bob
\]

\[
\text{acyclic} \ ob
\]

\textbf{Allowed.} The edges form a cycle, but \text{ctrl};[R] to read events is not in \text{ob}
Write forwarding from an unknown-address write

Forbidden. ob includes addr;rfi: forwarding is only possible when the address is determined

Contents 4.5 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V axiomatic model
Write forwarding on a speculative path

**PPOCA**
- Thread 0
  - a: \( Wx = 1 \)
  - c: \( Wy = 1 \)
  - f: \( Rz = 1 \)
- Thread 1
  - d: \( Ry = 1 \)
  - e: \( Wz = 1 \)
  - g: \( Rx = 0 \)

\[
\text{let obs} = \text{rfe} | \text{fre} | \text{coe} \\
\text{let dob} = \text{addr} | \text{data} \\
| \text{ctrl}; [\text{W}] \\
| \text{addr}; \text{po}; [\text{W}] \\
| (\text{ctrl} | \text{data}); \text{coi} \\
| (\text{addr} | \text{data}); \text{rfi} \\
\text{...} \\
\text{let bob} = \text{po}; [\text{dmb.sy}]; \text{po} \\
\text{...} \\
\text{let ob} = \text{obs} | \text{dob} | \text{aob} | \text{bob} \\
\text{acyclic ob}
\]

**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:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>if (y == 1)</td>
</tr>
<tr>
<td>y = 1</td>
<td>print x</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>int r1 = x</td>
</tr>
<tr>
<td>y = 1</td>
<td>if (y == 1)</td>
</tr>
<tr>
<td></td>
<td>print x</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>int r1 = x</td>
</tr>
<tr>
<td>y = 1</td>
<td>if (y == 1)</td>
</tr>
<tr>
<td></td>
<td>print x</td>
</tr>
</tbody>
</table>

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

\[
p\text{rint } x \quad \Longrightarrow \quad \text{print } r1
\]
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:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>int r1 = x</td>
</tr>
<tr>
<td>y = 1</td>
<td>if (y == 1)</td>
</tr>
<tr>
<td></td>
<td>print r1</td>
</tr>
</tbody>
</table>

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

\[
\text{print } x \quad \Rightarrow \quad \text{print } r1
\]

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 Analysis/Pattern Precision Eliminator
- basic-as: Basic Alias Analysis (stateless A A eval)
- basiccg: Basic CallGraph Construction
- count-as: Count Alias Analysis Query Responses
- da: Dependence Analysis
- debug-as: A A 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 (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-predom: Print postdominance tree of function to “dot” file

globaledit-as: Simple mod/ref analysis for globals

intcount: Counts the various types of Instructions
intervals: Interval Function Construction
iv-users: Induction Variable Users
lazy-value-info: Lazy Value Information Analysis
libcall-as: LibCall Alias Analysis
lit: Statically lint-checks LLVM IR
loops: Natural Loop Information
mdnode: Memory Dependence Analysis
module-decoders: 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-cfg-sccs: Print SCCs of the Call Graph
print-cfg: Print CFGs of each function
print-dom-info: Print Dom在家 Module
print-indentifiers: Print internal fn callsites passed constants
print-function: Print function to stderr
print-module: Print module to “dot” file
print-projections: Print the projections of the call graph
print-remat: Print rematerialization pass info
print-sccp: Print SCCs of each function
print-sroa: Print SROA pass info
print-reg2mem: Print register to memory pass info
print-reg2store: Print register to store pass info
print-reassociate: Print reassociate pass info
print-trylambda: Print trylambda pass info
print-tribute-activities: Print tribute pass info
print-tribute-dominance: Print tribute pass info
print-use: Print use pass info
print-use-dominator: Print use dominance pass info

Tree SSA passes
Remove useless statements
OpenMP lowering
OpenMP expansion
Loop non-atomic
Lower exception handling control flow
Build the control flow graph
Find all referenced variables
Enter static single assignment form
Warp for unitialized variables
Dead code elimination
Dominator optimization
Forward propagation of single-use variables
Copy Renaming
PHI node optimizations
May-alias optimization
Profiling
GCC
IPA passes
IPA i128/langle data
IPA remove symbols
IPA OpenACC
IPA points-to analysis
IPA OpenACC kernels
Target clone
IPA auto profile
IPA i128/long data
IPA free function summary
IPA increase TLS
IPA whole program visibility
IPA profile
IPA i128/short data
IPA global mod/ref analysis
IPA scalar replacement of aggregates
IPA constructor/destructor merge
IPA function summary
IPA inline
IPA i128/const analysis
IPA free function summary
IPA reference
IPA i128/short data
IPA i128/comasts
Materialize all clones
IPA points-to analysis
OpenMP simd clone

Tree SSA passes
Remove useless statements
OpenMP lowering
OpenMP expansion
Loop non-atomic
Lower exception handling control flow
Build the control flow graph
Find all referenced variables
Enter static single assignment form
Warp for unitialized variables
Dead code elimination
Dominator optimization
Forward propagation of single-use variables
Copy Renaming
PHI node optimizations
May-alias optimization
Profiling
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 scheduling
Mode switching optimization
Mode scheduling
Instruction scheduling
Register allocation
The integrated register allocator (IR)
Relaxing
Basic block reordering
Variable tracking
Delayed branch scheduling
Branch shortening
Register-to-stack conversion
Final

Contents
5.1 Programming language concurrency: Introduction
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 [43, Ševčík].
Defining PL Memory Models

Option 1: Don’t. No Concurrency

Tempting... but poor match for current practice
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).
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)
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. [44, 45]; Sarita Adve & Mark Hill [46, 47]
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,...)
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 [48, Manson, Pugh, Adve]
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 [48, 49] 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:

\[
\begin{array}{c|c}
\text{x = y = 0} & \text{HotSpot optimisation} \\
\hline
r1=x & r2=y \\
y=r1 & x=(r2==1)?y:1 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{x = y = 0} & \\
\hline
r1=x & x=1 \\
y=r1 & r2=y \\
\end{array}
\]

[49, 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

[50, 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.

- `memory_order_seq_cst` (SC semantics among themselves)
- `memory_order_release/memory_order_acquire` (release/acquire semantics for message-passing)
- `memory_order_release/memory_order_consume(deprecated)` (was supposed to expose dependency guarantees in C/C++)
- `memory_order_relaxed` (implementable with plain machine loads and stores)
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:
  ```c
  _Atomic(Node *) top;
  ```
- or annotate the accesses with a memory order:
  ```c
  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:
  ```c
  x.store(v, memory_order_release)
  r = x.load(memory_order_acquire)
  ```

Contents 5.3 Programming language concurrency: C/C++11
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 [23, 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, 26, 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:

- $\textbf{sb}$ sequenced-before (like po program order, but can be partial)
- $\textbf{asw}$ additional synchronizes with (synchronisation from thread creation etc.)

Candidate execution witness:

- $\textbf{rf}$ – reads-from
- $\textbf{mo}$ – modification order (like co coherence, but over atomic writes only)
- $\textbf{sc}$ – 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
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 single threaded program
A data race

```c
int y, x = 2;
x = 3; // | y = (x==3);
```

![Diagram](image)
A data race

```c
int y, x = 2;
x = 3;       | y = (x==3);
```

![Diagram of data race](image)
Simple concurrency: Decker’s example and SC

```c
atomic_int x = 0;
atomic_int y = 0;
x.store(1, seq_cst);
y.load(seq_cst);
y.store(1, seq_cst);
x.load(seq_cst);
```
Simple concurrency: Decker’s example and SC

```c
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);
y.load(seq_cst);
y.store(1, seq_cst);
x.load(seq_cst);
```

\[c:W_{sc} \ y=1\]
\[d:R_{sc} \ x=0\]
\[e:W_{sc} \ x=1\]
\[f:R_{sc} \ y=0\]
Simple concurrency: Decker’s example and SC

```c
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);
y.load(seq_cst);
y.store(1, seq_cst);
x.load(seq_cst);
```

c:W\_sc y=1  

\[\text{FORBIDDEN}\]

\[\text{FORBIDDEN}\]

d:R\_sc x=0  

\[\text{FORBIDDEN}\]

\[\text{FORBIDDEN}\]

e:W\_sc x=1  

f:R\_sc y=0
Simple concurrency: Decker’s example and SC

```c
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);
y.load(seq_cst);
y.store(1, seq_cst);
x.load(seq_cst);
```

```
c: W_{sc} y = 1
d: R_{sc} x = 0
e: W_{sc} x = 1
f: R_{sc} y = 1
```
Expert concurrency: The release-acquire idiom

// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;

\[
a:W_{na} \ x=1 \\
b:W_{rel} \ y=1 \\
c:R_{acq} \ y=1 \\
d:R_{na} \ x=1 \\
\]

sb
rf
sb
Expert concurrency: The release-acquire idiom

// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;

\[
\begin{align*}
\text{a:} & \text{W}_\text{na} x=1 \\
\text{sb} & \\
\text{b:} & \text{W}_\text{rel} y=1 \\
\text{sw} & \\
\text{c:} & \text{R}_\text{acq} y=1 \\
\text{sb} & \\
\text{d:} & \text{R}_\text{na} x=1
\end{align*}
\]
Expert concurrency: The release-acquire idiom

// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;

\[
\begin{align*}
a: & W_{na} \ x = 1 \\
b: & W_{rel} \ y = 1 \\
c: & R_{acq} \ y = 1 \\
d: & R_{na} \ x = 1
\end{align*}
\]
Expert concurrency: The release-acquire idiom

// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;

\[
\text{simple-happens-before} \rightarrow = \left( \text{sequenced-before} \rightarrow \cup \text{synchronizes-with} \rightarrow \right)^+ \]
Unlocks and locks synchronise too:

```c
int x, r;
mutex m;
m.lock();
x = ... m.lock();
m.unlock();
r = x;
```
Unlocks and locks synchronise too:

```c
int x, r;
mutex m;
m.lock();
x = ...;  // Critical Section
m.unlock();
m.lock();
r = x;
```

Diagram:

- `c: L mutex`
- `h: L mutex`
- `d: W_{na} x=1`
- `i: R_{na} x=1`
Locks and unlocks

Unlocks and locks synchronise too:

```c
int x, r;
mutex m;
m.lock();
x = ...  // x = ...
m.unlock();

m.lock();
r = x;
```

```
\begin{tikzpicture}
  \node (c) at (0,0) {c:L mutex};
  \node (d) at (0,-2) {d:W_{na} x=1};
  \node (f) at (0,-4) {f:U mutex};
  \node (h) at (3,0) {h:L mutex};
  \node (i) at (3,-2) {i:R_{na} x=1};

  \draw[->] (c) -- (d);
  \draw[->] (d) -- (f);
  \draw[->] (i) -- (h);

  \node[below] at (c) {sb};
  \node[below] at (d) {sb}
  \node[below] at (f) {sb}
  \node[below] at (h) {sb}

  \node[below] at (i) {sb}

  \node[above] at (c) {x=1};
  \node[above] at (f) {x=1};
  \node[above] at (i) {x=1};

  \node[above] at (h) {x=1};

  \node[below] at (c) {sc};
\end{tikzpicture}
```
Locks and unlocks

Unlocks and locks synchronise too:

```c
int x, r;
mutex m;

m.lock();
x = ...

m.unlock();
m.lock();
r = x;
```

![Diagram showing mutex lock and unlock operations]
Locks and unlocks

Unlocks and locks synchronise too:

```c
int x, r;
mutex m;
m.lock();
x = ...
m.unlock();
m.lock();
r = x;
```

```
c:L mutex   h:L mutex
```

```
sb
sb           sb
```

```
d:W_{na} x=1  i:R_{na} x=1
```

```
f:U mutex
```

```
hb
```

```c
m.lock();
r = x;
```
Locks and unlocks

Unlocks and locks synchronise too:

```c
int x, r;
mutex m;
m.lock();
x = ...
m.unlock();
m.lock();
r = x;
```

![Diagram showing mutex synchronization with lock and unlock operations]
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

```c
int y, x = 2;
x = 3;        \quad \mid y = (x==3);
```

```plaintext
A data race

int y, x = 2;
x = 3;        \quad \mid y = (x==3);
```
Data race definition

let data_races actions hb =
{ (a, b) | ∀ a∈actions b∈actions |
¬ (a = b) ∧
same_location a b ∧
(is_write a ∨ is_write b) ∧
¬ (same_thread a b) ∧
¬ (is_atomic_action a ∧ is_atomic_action b) ∧
¬ ((a, b) ∈ hb ∨ (b, a) ∈ hb) }

A program with a data race has undefined behaviour.
Relaxed writes: load buffering

```plaintext
x.load(relaxed);
y.store(1, relaxed);
y.load(relaxed);
x.store(1, relaxed);
c:Rrlx x=1
d:Wrlx y=1
e:Rrlx y=1
f:Wrlx x=1
sb
rf
sb
rf
```

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);

\[ c:Wrlx \ x=1 \quad d:Wrlx \ y=1 \quad e:Rrlx \ x=1 \quad g:Rrlx \ y=1 \]
\[ f:Rrlx \ y=0 \quad h:Rrlx \ x=0 \]
Expert concurrency: fences avoid excess synchronisation

// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(acquire));
r = x;
Expert concurrency: fences avoid excess synchronisation

// sender
x = ...
y.store(1, release);
// receiver
while (0 == y.load(acquire));
r = x;

// 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;

d:Wrel y=1
f:Facq
r:Rrlx y=1
c:Wna x=1
g:Rna x=1
Expert concurrency: The fenced release-acquire idiom

// sender
x = ... 
y.store(1, release);

// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;

d:W_{rel} y=1  
\downarrow \hspace{1cm} \downarrow
f:R_{acq}
\downarrow
sb

\rightarrow

\textcolor{red}{rf}
\rightarrow

c:W_{na} x=1 
\downarrow
sb

\rightarrow

e:R_{rlx} y=1 
\downarrow
sb

\rightarrow

g:R_{na} x=1
Expert concurrency: The fenced release-acquire idiom

// sender
x = ...
y.store(1, release);

// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;

c:W_{na} x=1
d:W_{rel} y=1
e:R_{rlx} y=1
g:R_{na} x=1
Modication order is a per-location total order over atomic writes of any memory order.

```c
x.store(1, relaxed);
x.store(2, relaxed);
x.load(relaxed);
x.load(relaxed);
```
Modification order is a per-location total order over atomic writes of any memory order.

```
x.store(1, relaxed);
x.store(2, relaxed);
x.load(relaxed);
```

```
b:W_{rlx} x=1  
  sb 
  c:W_{rlx} x=2  
  sb  
  d:R_{rlx} x=1  
  rf  
  e:R_{rlx} x=2  
  rf
```
**Modification order** is a per-location total order over atomic writes of any memory order.

```plaintext
x.store(1, relaxed);
| x.load(relaxed);
```
```plaintext
x.store(2, relaxed);
| x.load(relaxed);
```

```
b:W_{rlx} x=1
```
```
c:W_{rlx} x=2
```
```
d:R_{rlx} x=1
```
```
e:R_{rlx} x=2
```

mo
rf
sb
Coherence and atomic reads

All forbidden!

Atomics cannot read from later writes in happens before.
A successful \texttt{compare\_exchange} is a read-modify-write.

Read-modify-writes read the last write in \texttt{mo}:

\begin{verbatim}
x.store(1, relaxed);
x.store(2, relaxed);
x.store(4, relaxed);
\texttt{compare\_exchange(\&x, 2, 3, relaxed, relaxed);}
\end{verbatim}
Read-modify-writes

A successful compare_exchange is a read-modify-write.

Read-modify-writes read the last write in $x$:

\[
x\text{.store}(1, \text{relaxed}); \quad \text{compare_exchange}(&x, 2, 3, \text{relaxed}, \text{relaxed}); \\
x\text{.store}(2, \text{relaxed}); \\
x\text{.store}(4, \text{relaxed});
\]

\[a:W_{rlx} x=1 \quad d:RMW_{rlx} x=2/3\]

\[sb\]

\[b:W_{rlx} x=2\]

\[sb\]

\[c:W_{rlx} x=4\]
Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in `mo`:

```c
x.store(1, relaxed);
x.store(2, relaxed);
x.store(4, relaxed);
compare_exchange(&x, 2, 3, relaxed, relaxed);
```
A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```c
x.store(1, relaxed);
x.store(2, relaxed);
x.store(4, relaxed);

compare_exchange(&x, 2, 3, relaxed, relaxed);
```

```
a: W_{rlx} x=1
b: W_{rlx} x=2
c: W_{rlx} x=4
d: RMW_{rlx} x=2/3
```
Very expert concurrency: consume

Weaker than acquire

Stronger than relaxed

Non-transitive happens before! (only fully transitive through data dependence, dd)
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)
C1x and C++11 support many modes of programming:
  * sequential
C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
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
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 [51], a typed specification language which can be translated into OCaml and multiple provers.
The full model

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]

\[ \{(a, b) \in r | a \neq b = (a, b) \notin r \} \]

\[ \{(a, b) = (a, b) \} \]

\[ \{(a, b) \} \]
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/
5.3.1 Programming language concurrency: C/C++11: C/C++11 models and tooling
Rephrased in relational algebra, in .cat, and improved in various ways:

▶ Overhauling SC atomics in C11 and OpenCL. Batty, Donaldson, Wickerson. [52]. Supplementary material: http://multicore.doc.ic.ac.uk/overhauling/

Usable in herd, for examples in a small C-like language
C11 cat from [52, Batty, Donaldson, Wickerson], adapted by Lau for [53]

```plaintext
let sb = po | I \ (M \ I)
let no = co
let cacq = [ACQ | (SC & (R | F)) | ACQ,REL]
let crel = [REL | (SC & (W | F)) | ACQ,REL]
let fr = rf.inv ; no
let fsb = [F] ; sb
let sbf = sb ; [F]
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] ; sb ? ; 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
let s1_prime = [SC] ; sc_clk_imm ; hb
let s2_prime_head = [SC] ; sc_clk ; fsb?
let s2_prime_tail = mo ; sbf?
let s2_prime = [SC] ; s2_prime_head ; s2_prime_tail
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
let s4_prime = [SC] ; sc_clk_imm ; rf_inv ; [W]
let s5_prime = [SC] ; sc_clk ; fr
let s6_prime = [SC] ; sc_clk ; fr ; sbf
let s7_prime_head = [SC] ; sc_clk
let s7_prime_tail = rf ; sbf
let s7_prime = [SC] ; s7_prime_head ; s7_prime_tail
let __bmc_hb = hb
let s_prime = [SC] ; sc_clk_imm ; hb
```

//Modified from:
//C11.cat w/o locks, consume output addr
output data

// Modified from:
// C11.cat w/o locks, consume output data

```plaintext
let sb = po | I \ (M \ I)
let no = co
let cacq = [ACQ | (SC & (R | F)) | ACQ,REL]
let crel = [REL | (SC & (W | F)) | ACQ,REL]
let fr = rf.inv ; no
let fsb = [F] ; sb
let sbf = sb ; [F]
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] ; sb ? ; 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
let s1_prime = [SC] ; sc_clk_imm ; hb
let s2_prime_head = [SC] ; sc_clk ; fsb?
let s2_prime_tail = mo ; sbf?
let s2_prime = [SC] ; s2_prime_head ; s2_prime_tail
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
let s4_prime = [SC] ; sc_clk_imm ; rf_inv ; [W]
let s5_prime = [SC] ; sc_clk ; fr
let s6_prime = [SC] ; sc_clk ; fr ; sbf
let s7_prime_head = [SC] ; sc_clk
let s7_prime_tail = rf ; sbf
let s7_prime = [SC] ; s7_prime_head ; s7_prime_tail
let __bmc_hb = hb
```
Cerberus BMC

- Cerberus-BMC: a Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. Lau, Gomes, Memarian, Pichon-Pharabod, Sewell. [53]

Integrates the Cerberus semantics for a substantial part of C [54, 55, 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
// MP+na-rel+acq-na
// Message Passing, of data held in non-atomic x, with
// release/acquire synchronisation on y.
// If the value of r1 is 1, then the value of r2 should also
// be 1.
// An exhaustive execution of this program should therefore
// return the value 1 and 2, but not 0.
#include <stdatomic.h>
int main() {
    int x = 0;
    Atomic<int> y = 6;
    int r1, r2;
    {-
        x = 1;
        atomic_store_explicit(&y, 1, memory_order_release);
    }||{
        r1 = atomic_load_explicit(&y, memory_order_acquire);
        if (r1 == 1)
            r2 = x;
        else
            r2 = 2;
    } assert((r1 == 1 && r2 == 0));
    return r1 + 2 * r2;
}
// RC11 .cat file without fences
// adapted 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 = sw_prime | asw
let hb = (sb | sw)+
let mo = co
let fr = (rf_invs ; 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 ; sb ; sb & loc ; mo | fr
let psc_base = ([SC] | fhb) ; scb ; ([SC] | hbf)
let psc = [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
Can we compile to x86?

<table>
<thead>
<tr>
<th>Operation</th>
<th>x86 Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load(non-seq_cst)</td>
<td>mov</td>
</tr>
<tr>
<td>load(seq_cst)</td>
<td>lock xadd(0)</td>
</tr>
<tr>
<td>store(non-seq_cst)</td>
<td>mov</td>
</tr>
<tr>
<td>store(seq_cst)</td>
<td>lock xchg</td>
</tr>
<tr>
<td>fence(non-seq_cst)</td>
<td>no-op</td>
</tr>
</tbody>
</table>

x86-TSO is stronger and simpler.
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 [58, Podkopaev, Lahav, Vafeiadis]
**Mappings**

Compilation from C/C++11 involves mapping each synchronisation operation to hardware and restricting compiler optimisations across these.

<table>
<thead>
<tr>
<th>C/C++11 operation</th>
<th>x86</th>
<th>Armv8-A AArch64</th>
<th>Power</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Relaxed</td>
<td>mov</td>
<td>ldr</td>
<td>ld</td>
<td></td>
</tr>
<tr>
<td>Store Relaxed</td>
<td>mov</td>
<td>str</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td>Load Acquire</td>
<td>mov</td>
<td>ldar(^2)</td>
<td>ld;cmp;bc;isync</td>
<td></td>
</tr>
<tr>
<td>Store Release</td>
<td>mov</td>
<td>stlr</td>
<td>lwsync;st</td>
<td></td>
</tr>
<tr>
<td>Load Seq_Cst</td>
<td>mov</td>
<td>ldar(^3)</td>
<td>sync;ld;cmp;bc;isync(^4)</td>
<td></td>
</tr>
<tr>
<td>Store Seq_Cst</td>
<td>xchg(^1)</td>
<td>stlr(^3)</td>
<td>sync;st(^4)</td>
<td></td>
</tr>
<tr>
<td>Acquire fence</td>
<td>nothing</td>
<td>dmb ld</td>
<td>lwsync</td>
<td></td>
</tr>
<tr>
<td>Release fence</td>
<td>nothing</td>
<td>dmb</td>
<td>lwsync</td>
<td></td>
</tr>
<tr>
<td>Acq_Rel fence</td>
<td>nothing</td>
<td>dmb</td>
<td>lwsync</td>
<td></td>
</tr>
<tr>
<td>Seq_Cst fence</td>
<td>mfence</td>
<td>dmb</td>
<td>hwsync</td>
<td></td>
</tr>
</tbody>
</table>

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.

---

Contents 5.3.2 Programming language concurrency: C/C++11: Mappings from C/C++11 to hardware
C/C++11 operational model

proved equivalent to that axiomatic model, in Isabelle [19, Nienhuis et al.]
C/C++11 after 2011

- 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. [59]
- Outlawing ghosts: avoiding out-of-thin-air results. Boehm, Demsky. [60]
- The Problem of Programming Language Concurrency Semantics. Batty, Memarian, Nienhuis, Pichon-Pharabod, Sewell. [17]
- Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. Vafeiadis, Balabonski, Chakraborty, Morisset, Zappa Nardelli. [61]
- Overhauling SC atomics in C11 and OpenCL. Batty, Donaldson, Wickerson. [52]
- An operational semantics for C/C++11 concurrency. Nienhuis, Memarian, Sewell. [19]
- Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. Manerkar, Trippel, Lustig, Pellauer, Martonosi. [62]
- Repairing sequential consistency in C/C++11. Lahav, Vafeiadis, Kang, Hur, Dreyer. [63]
- Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. Flur, Sarkar, Pulte, Nienhuis, Maranget, Gray, Sezgin, Batty, Sewell. [20]
- Bridging the gap between programming languages and hardware weak memory models. Podkopaev, Lahav, Vafeiadis. [58]
- Cerberus-BMC: a Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. Lau, Gomes, Memarian, Pichon-Pharabod, Sewell. [53]
- P0668R5: Revising the C++ memory model. Boehm, Giroux, Vafeiadis. [56]
- P0982R1: Weaken Release Sequences. Boehm, Giroux, Vafeiadis. [57]
- ...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 [64, 23.9p9]:

[Note: The requirements do allow \( r_1 == r_2 == 42 \) in the following example, with \( x \) and \( y \) initially zero:

\[
\begin{align*}
\text{LB+ctrldata+ctrl-single} \\
\text{r1 = load}_{rlx}(x); & \quad \text{r2 = load}_{rlx}(y); \\
\text{if } (r1 == 42) & \quad \text{if } (r2 == 42) \\
\text{store}_{rlx}(y,r1) & \quad \text{store}_{rlx}(x,42)
\end{align*}
\]

\( a:R_{rlx}x = 42 \) \quad \( b:R_{rlx}y = 42 \)

\( sb \downarrow cd,dd \quad rf \quad rf \quad sb \downarrow cd \)

\( c:W_{rlx}y = 42 \) \quad \( d:W_{rlx}x = 42 \)

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:

\[
\begin{align*}
\text{LB+ctlrdata+ctrl-single} \\
\text{r1} &= \text{load}_{rlx}(x); \quad \text{r2} = \text{load}_{rlx}(y); \\
\text{if (r1 == 42)} &\quad \text{if (r2 == 42)} \\
\text{store}_{rlx}(y, r1) &\quad \text{store}_{rlx}(x, 42)
\end{align*}
\]

\[
\begin{align*}
\text{a:R}_{rlx}x &= 42 & \text{b:R}_{rlx}y &= 42 \\
\text{sb} &\downarrow \text{rf} & \text{sb} &\downarrow \text{rf} \\
\text{cd,dd} &\text{rf} & \text{cd} &\text{rf} \\
\text{c:W}_{rlx}y &= 42 & \text{d:W}_{rlx}x &= 42
\end{align*}
\]

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. [17] observe that this cannot be solved with any per-candidate-execution model that uses the C/C++11 notion of candidate execution. Consider:

```
LB+ctrlndata+ctrl-double

r1 = load_{rlx}(x);
if (r1 == 42)
    store_{rlx}(y,r1)
else
    store_{rlx}(x,42)

r2 = load_{rlx}(y);
if (r2 == 42)
    store_{rlx}(x,42)
```

Compilers will optimise the second thread's conditional, removing the control dependency, to:

```
r1 = load_{rlx}(x);
if (r1 == 42)
    store_{rlx}(x,42)
```

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+ctrlndata+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. [65]

- **Option 8b**: Explaining Relaxed Memory Models with Program Transformations. Lahav, Vafeiadis. [66]

- **Option 8c**: forbid load-to-store reordering, making \( rf \cup sb \) acyclic [67, 60, 61, 63]

- **Option 8d**: Promising 2.0: global optimizations in relaxed memory concurrency. Lee, Cho, Podkopaev, Chakraborty, Hur, Lahav, Vafeiadis [68]

- **Option 8e**: Modular Relaxed Dependencies in Weak Memory Concurrency. Paviotti, Cooksey, Paradis, Wright, Owens, Batty. [69]

- **Option 8f**: Pomsets with Preconditions: A Simple Model of Relaxed Memory. Jagadeesan, Jeffrey, Riely [70]

- ...? 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](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. [71]

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.r.t. compiler optimisations
GPU concurrency

- GPU Concurrency: Weak Behaviours and Programming Assumptions. Alglave, Batty, Donaldson, Gopalakrishnan, Ketema, Poetzl, Sorensen, Wickerson. [72]
- Remote-scope promotion: clarified, rectified, and verified. Wickerson, Batty, Beckmann, Donaldson. [73]
- Overhauling SC atomics in C11 and OpenCL. Batty, Donaldson, Wickerson. [52].
- Exposing errors related to weak memory in GPU applications. Sorensen, Donaldson. [74]
- Portable inter-workgroup barrier synchronisation for GPUs. Sorensen, Donaldson, Batty, Gopalakrishnan, Rakamaric. [75]
Option 11: broadly follow C/C++

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. [76]
- Weakening WebAssembly. Watt, Rossberg, Pichon-Pharabod. [77]
“local data race freedom”

- Bounding data races in space and time. Dolan, Sivaramakrishnan, Madhavapeddy. [78]
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+concurrency in theorem provers)

▶ “system” fragment – what you need in addition for OS kernels and hypervisors: instruction fetch, exceptions, virtual memory. Ongoing – e.g. [24, 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

<table>
<thead>
<tr>
<th>Status</th>
<th>Total</th>
<th>i7-8665U</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+1W</td>
<td>Allow</td>
<td>—</td>
</tr>
<tr>
<td>2+2W</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>CoRR</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>CoRW1</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>CoRW2</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>CoWR0</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>CoWW</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>INC</td>
<td>Allow</td>
<td>298/100M</td>
</tr>
<tr>
<td>IRIW</td>
<td>Forbid</td>
<td>0/100M</td>
</tr>
<tr>
<td>LB</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>LockINC</td>
<td>Forbid</td>
<td>0/100M</td>
</tr>
<tr>
<td>MP</td>
<td>Forbid</td>
<td>0/100M</td>
</tr>
<tr>
<td>R</td>
<td>Allow</td>
<td>—</td>
</tr>
<tr>
<td>S</td>
<td>Forbid</td>
<td>—</td>
</tr>
<tr>
<td>SB</td>
<td>Allow</td>
<td>171/100M</td>
</tr>
<tr>
<td>SB+mfences</td>
<td>Forbid</td>
<td>0/100M</td>
</tr>
<tr>
<td>SB+rfi-pos</td>
<td>Allow</td>
<td>320/100M</td>
</tr>
<tr>
<td>WRC</td>
<td>Forbid</td>
<td>0/100M</td>
</tr>
</tbody>
</table>
AArch64 Experimental Results

Status

Total

ec2-a1 (a)

BCM2711 (b)

h955 (c)

AMD (d)

Juno (e)

Kirin6220 (f)

HelioG25 (g)

S905 (h)

1.77M/140M
0/140M
0/140M

0/140M
0/140M

248M/3.99G
0/3.76G
0/6.60G

0/3.99G
0/3.72G

40.9M/300M
0/300M
0/300M

0/300M
0/300M

26.3M/260M
0/260M
0/260M

0/260M
0/260M

31.3M/312M
0/312M
0/312M

0/312M
0/312M

1.46M/24.0M
0/24.0M
—

0/24.0M
—

126M/4.56G
0/4.56G
0/6.40G

0/4.56G
0/4.54G

0/312M
—
0/312M
0/312M
0/312M

—
—
—
—
0/24.0M

—
16.7M/260M
0/260M
—
—

—
14.2M/312M
0/312M
—
—

—
39.4k/24.0M
—
—
—

—
61.2M/4.56G
0/4.80G

0/1.08G
—

—
4.03M/198M
—

0/194M
—

—
23.5M/3.35G
0/1.86G
33.5k/3.31G
—

—
512k/1.85G
0/1.74G
216k/1.20G
—

—
14.9M/8.15G
0/6.44G
5285/560M
—

—
73.5M/2.02G
0/3.22G
—
—

300k/300M
0/300M

0/300M
726k/300M
759k/300M

829k/260M
0/260M

0/260M
1.85M/260M
1.73M/260M

838k/312M
0/312M

0/312M
1.27M/312M
1.17M/312M

—
—
—
4435/24.0M
—

9.47M/6.40G
0/4.56G

0/4.56G
6.95M/4.56G
15.9M/6.40G

6.65M/2.59G
0/194M

0/194M
587k/194M
8.16M/2.59G

1804/3.75G
0/3.35G

0/3.35G
12.1k/3.35G
14.5k/3.75G

76.0k/1.74G
0/1.75G

0/1.75G
179k/1.81G
142k/1.74G

80.6k/6.44G
0/8.13G

0/8.13G
900k/8.14G
1.02M/11.1G

276k/3.22G
0/2.02G

0/2.02G
335k/2.02G
545k/3.22G

0/3.76G
60.2M/3.18G
0/3.18G
—
—

0/300M
4.59M/300M
0/300M
—
—

0/260M
7.09M/260M
0/260M
—
—

0/312M
7.45M/312M
0/312M
—
—

0/24.0M
14.1k/24.0M
—
—
—

0/4.56G
34.1M/4.56G
0/4.56G
—
—

0/194M
3.20M/198M
0/198M
—
—

0/3.35G
544k/3.35G
0/3.35G
—
—

0/1.81G
2917/1.85G
0/1.85G
—
—

0/8.87G
0/4.95G
0/4.95G
—
—

0/140M
156k/140M
0/70.0M
204k/140M
25.3M/140M

0/6.60G
1.65M/6.60G
0/3.29G
3.54M/6.60G
1.03G/4.06G

0/300M
232k/300M
0/200M
454k/300M
88.4M/300M

0/260M
257k/260M
0/160M
571k/260M
92.7M/260M

0/312M

0/312M
0/162M

0/312M
180M/312M

—
—
—
—
20.8M/24.0M

0/6.40G

0/6.40G
0/3.20G

0/6.40G
1.15G/6.16G

0/2.59G

0/2.59G
0/1.30G

0/2.59G
44.5M/198M

0/3.75G
14.2k/3.75G
0/3.51G
20.5k/3.75G
137M/3.35G

0/1.74G
74.9k/1.74G
—
7322/1.74G
11.1M/1.83G

0/140M
—
0/140M
0/70.0M

0/3.76G
—
0/4.45G
0/1.85G

0/300M
—
—
0/200M

0/260M
—
—
0/160M

0/312M
—
—
0/162M

0/24.0M
—
—
—

0/4.56G
—
0/4.66G
0/2.27G

0/194M
—
0/194M
0/97.0M

0/3.35G
—
0/3.35G
0/3.31G

0/1.80G
—
0/1.74G
—

Allow
Forbid
Forbid
Allow
Forbid

LB+ctrls
LB+data.reals
LB+datas
LB+datas+WW
LB+dmb.sys

Forbid
Forbid
Forbid
Allow
Forbid

0/38.0G
—
0/42.6G
16.6M/38.0G
0/40.1G

llsc-inc
MP
MP+dmb.sy+addr
MP+dmb.sy+addr-po
MP+dmb.sy+addr.real

Forbid
Allow
Forbid
Allow
Forbid

—
675M/43.0G
0/38.4G
7.51M/17.4G
—

—
68.3M/3.32G
—

0/3.32G
—

—
2.51M/140M
—
344k/140M
—

—
153M/3.99G
0/6.60G
1.41M/610M
—

—
40.9M/300M
0/300M
—
—

MP+dmb.sy+ctrl
MP+dmb.sy+ctrlisb
MP+dmb.sy+fri-rfi-ctrlisb
MP+dmb.sy+po
MP+dmb.sy+rs

Allow
Forbid
Allow
Allow
Allow

52.7M/48.6G
0/42.6G
1/42.6G
69.8M/42.7G
94.3M/58.9G

2.05M/3.32G
0/3.32G

0/3.32G
4.05M/3.32G
4.05M/3.32G

225k/140M
0/140M

0/140M
454k/140M
466k/140M

22.7M/6.60G
0/3.74G

0/3.74G
31.2M/3.76G
41.8M/6.60G

MP+dmb.sys
MP+po+dmb.sy
MP+popl+poap
MP+rfi-addr+addr
MP+si+po

Forbid
Allow
Forbid
Allow
Allow

0/44.0G
173M/38.9G
0/38.9G
—
—

0/3.32G
3.04M/3.32G
0/3.32G
—
—

0/140M
306k/140M
0/140M
—
—

PPOAA
PPOCA
RDW
RSW
SB

Forbid
Allow
Forbid
Allow
Allow

0/58.9G
6.26M/58.9G
0/31.9G
13.0M/58.9G
6.94G/44.6G

0/3.32G
940k/3.32G
0/1.97G
2.08M/3.32G
402M/3.32G

SB+dmb.sys
SB+rfi-addrs
S+dmb.sy+data-wsi
WRC+addrs

Forbid
Allow
Forbid
Forbid

0/44.0G
—
0/28.1G
0/21.6G

0/3.32G
—
0/3.32G
0/1.97G

Contents

950M/42.9G
0/44.0G
3.32G/47.4G
18.3M/42.9G
0/38.0G

39.9M/3.32G
0/3.32G
3.32G/3.32G

0/3.32G
0/3.32G

2+2W
2+2W+dmb.sys
CoWR
LB
LB+addrs+WW

?

?



0/3.32G
—
0/3.32G
0/3.32G
0/3.32G



0/140M
—
0/140M
0/140M
0/140M



0/3.72G
—
0/3.74G
0/3.72G
0/3.18G



0/300M
—
0/300M
0/300M
0/300M



0/260M
—
0/260M
0/260M
0/260M



7 Appendix: Selected Experimental Results:



0/4.54G
—
0/4.56G
0/4.54G
0/4.56G

Snapdragon425 (i) a10x-fusion (j)
8.27M/198M
0/194M
0/2.59G

0/198M
0/194M



0/194M
—
0/194M
0/194M
0/194M

29.2M/3.35G
0/3.35G
0/3.75G

0/3.35G
0/3.35G



0/3.35G
—
0/3.35G
0/3.35G
0/3.35G

iphone7 (k)
156k/1.85G
0/1.81G
0/1.74G

0/1.82G
0/1.74G



0/1.74G
—
0/1.75G
0/1.74G
0/1.80G

ipadair2 (l)


0/8.12G
0/8.87G
0/5.84G

0/8.15G
0/5.84G



0/5.84G
—
0/8.13G
0/5.84G
0/5.67G

APM883208 (m)

Cavium (n)

Exynos9 (o)

164M/2.02G
0/2.02G
0/3.22G

0/2.02G
0/2.02G

12.3k/773M
0/773M
0/1.37G

0/773M
0/761M

87.8M/3.16G
0/3.16G
0/3.16G

0/3.16G
0/3.16G



0/2.02G
—
0/2.02G
0/2.02G
0/2.02G



0/761M
—
0/773M
0/761M
0/695M

—
571/773M
0/1.37G

0/683M
—


0/3.16G
—
0/3.16G
0/3.16G
0/3.16G

—
64.7M/3.16G
0/2.52G
4.48M/3.16G
—

0/1.37G

8.61M/3.16G
0/3.16G

0/3.16G
19.5M/3.16G
16.8M/3.16G

0/2.02G
19.5M/2.02G
0/2.02G
—
—

0/773M
153/695M
0/695M
—
—

0/3.16G
31.5M/3.16G
0/3.16G
—
—

0/11.1G
184k/11.1G
0/10.6G
564k/11.1G
405M/8.15G

0/3.22G
137k/3.22G
0/1.61G
82.3k/3.22G
1.02G/2.02G

0/1.37G

0/1.37G
0/1.23G

0/1.37G
209M/773M

0/3.16G
2.61M/3.16G
0/1.64G
4.66M/3.16G
577M/3.16G

0/8.87G
—
0/560M
0/5.34G

0/2.02G
—
—
0/1.01G

0/773M
—
0/693M
0/619M

0/3.16G
—
0/3.16G
0/1.64G



0/1.37G
0/761M




0/761M

0/773M


nexus9 (p)


openq820 (q)

0/4.51G
0/5.11G
0/2.47G

0/4.51G
0/2.47G

145M/6.06G
0/6.06G
0/5.96G
18.3M/6.06G
0/5.88G

0/2.47G
—
0/4.51G
0/2.47G
0/5.11G

0/5.88G
—
0/6.06G
16.6M/5.88G
0/6.06G

—
0/4.51G
0/3.07G
—
—

—
138M/6.06G
0/5.96G
1.01M/3.12G
—

0/3.07G
0/4.51G

0/8.71G

618k/5.96G
0/6.06G
1/6.06G
1.83M/6.06G
1.66M/5.96G

0/5.11G
0/4.51G
0/4.51G
—
—

0/6.06G
1.03M/6.06G
0/6.06G
—
—

0/8.71G
0/8.71G
—
0/4.51G

0/5.96G
223/5.96G
0/2.98G
814k/5.96G
1.55G/6.06G

0/5.11G
—
—
—

0/6.06G
—
0/5.88G
0/2.94G








0/4.51G

0/4.51G







0/8.71G


514


## Power Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>Status</th>
<th>Total</th>
<th>bim</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+2W</td>
<td>Allow</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>IRIW+syncs</td>
<td>Forbid</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ISA2+sync+data+addr</td>
<td>Forbid</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LB</td>
<td>Allow</td>
<td>0/160M</td>
<td>0/160M</td>
</tr>
<tr>
<td>LB+ctrls</td>
<td>Forbid</td>
<td>0/160M</td>
<td>0/160M</td>
</tr>
<tr>
<td>LB+datas</td>
<td>Allow</td>
<td>160M/160M</td>
<td>160M/160M</td>
</tr>
<tr>
<td>MP</td>
<td>Allow</td>
<td>371k/160M</td>
<td>371k/160M</td>
</tr>
<tr>
<td>MP+eieio+addr</td>
<td>Forbid</td>
<td>*160M/160M</td>
<td>*160M/160M</td>
</tr>
<tr>
<td>MP+sync+addr</td>
<td>Allow</td>
<td>160M/160M</td>
<td>160M/160M</td>
</tr>
<tr>
<td>MP+sync+ctrl</td>
<td>Allow</td>
<td>1242/160M</td>
<td>1242/160M</td>
</tr>
<tr>
<td>MP+sync+ctrlisync</td>
<td>Allow</td>
<td>160M/160M</td>
<td>160M/160M</td>
</tr>
<tr>
<td>MP+sync+rs</td>
<td>Allow</td>
<td>2064/160M</td>
<td>2064/160M</td>
</tr>
<tr>
<td>SB</td>
<td>Allow</td>
<td>702k/160M</td>
<td>702k/160M</td>
</tr>
<tr>
<td>WRC+addr</td>
<td>Allow</td>
<td>103/100M</td>
<td>103/100M</td>
</tr>
<tr>
<td>WRC+eieio+addr</td>
<td>Allow</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>WRC+sync+addr</td>
<td>Forbid</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
## RISC-V Experimental Results

<table>
<thead>
<tr>
<th>Status</th>
<th>Total</th>
<th>hifiveu540</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+2W</td>
<td>Allow</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>LB</td>
<td>Allow</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>LB+ctrls</td>
<td>Forbid</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>LB+datas</td>
<td>Forbid</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>MP</td>
<td>Allow</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>MP+fence.rw.rw+addr</td>
<td>Forbid</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>MP+fence.rw.rw+ctrl</td>
<td>Allow</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>SB</td>
<td>Allow</td>
<td>0/1.20G</td>
</tr>
<tr>
<td>WRC+addrs</td>
<td>Forbid</td>
<td>0/600M</td>
</tr>
</tbody>
</table>
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 built on, or other parallel and recent developments.
Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave.
[pdf].

Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli.
In DAMP 2009: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming.
[pdf].

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell.
In CAV 2010: Proceedings of the 22nd International Conference on Computer Aided Verification, LNCS 6174.
[pdf].

[4] A better x86 memory model: x86-TSO.
Scott Owens, Susmit Sarkar, and Peter Sewell.
In TPHOLs 2009: Proceedings of Theorem Proving in Higher Order Logics, LNCS 5674.
[pdf].

(Research Highlights).
[pdf].

Scott Owens.
[url].

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber.
[pdf].
References

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams.
[project page].
[pdf].

[9] Litmus: running tests against hardware.
Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell.
[pdf].

[pdf].

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell.
[project page].
[pdf].

[12] Synchronising C/C++ and POWER.
Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams.
[project page].
[pdf].

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 CAV 2012: Proceedings of the 24th International Conference on Computer Aided Verification.
[pdf].

[pdf], Draft.
[15] Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory.
Jade Alglave, Luc Maranget, and Michael Tautschnig.
[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.
[pdf].

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell.
In ESOP 2015: Programming Languages and Systems – 24th European Symposium on Programming, European Joint Conferences on Theory and Practice of Software (ETAPS) (London).
[pdf].

[18] Modelling the ARMv8 architecture, operationally: concurrency and ISA.
In POPL 2016: Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA).
[project page].
[pdf].

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell.
[pdf].

[project page].
[pdf].

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell.
[project page].
[pdf].

[22] ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS.
[project page].
[pdf].

[23] Cerberus-BMC tool for exploring the behaviour of small concurrent C test programs with respect to an arbitrary axiomatic concurrency model,
[project page].
[web interface].

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].

Jade Alglave.
http://www0.cs.ucl.ac.uk/staff/J.Alglave/these.pdf.

Mark John Batty.
[pdf].

[27] The Semantics of Multicopy Atomic ARMv8 and RISC-V.

Contents 8 References: 522
References:

[28] A no-thin-air memory model for programming languages.
Jean Pichon-Pharabod.
https://www.repository.cam.ac.uk/handle/1810/274465.

[29] The diy7 tool suite (herdtools), Jade Alglave and Luc Maranget.
diy.inria.fr.
Accessed 2020-10-10.

[github], Accessed 2020-10-10.

https://isla-axiomatic.cl.cam.ac.uk/.
Accessed 2020-10-10.

Downloaded 2020-09-23. 5052 pages.

Downloaded 2020-09-23. 3165 pages.

Downloaded 2020-09-23. 8248 pages.
[35] Power ISA Version 3.0B, IBM. 
Downloaded 2020-09-23. 1258 pages.

Downloaded 2020-09-23. 238 pages.

Downloaded 2020-09-23. 135 pages.

[38] The Power of Processor Consistency. 
Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger.
In SPAA.

[39] Efficient and correct execution of parallel programs that share memory. 
Dennis Shasha and Marc Snir.

[40] Trustworthy specifications of ARM® v8-A and v8-M system level architecture. 
Alastair Reid.
[url].

[41] Who guards the guards? formal validation of the Arm v8-m architecture specification. 
Alastair Reid.
[url].

[42] Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. 
Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell.
In Proc. CAV.

[43] Safe optimisations for shared-memory concurrent programs. 
Jaroslav Sevcík.

Contents  8 References:  524
Memory Consistency Models for Shared Memory Multiprocessors.
Kourosh Gharachorloo.

Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.

Designing Memory Consistency Models for Shared-Memory Multiprocessors.
S. V. Adve.

Weak Ordering – A New Definition.
Sarita V. Adve and Mark D. Hill.

The Java memory model.
Jeremy Manson, William Pugh, and Sarita V. Adve.

On Validity of Program Transformations in the Java Memory Model.
Jaroslav Sevcik and David Aspinall.

Foundations of the C++ concurrency memory model.
Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell.
[project page].
[url].

[52] Overhauling SC atomics in C11 and OpenCL.
Mark Batty, Alastair F. Donaldson, and John Wickerson.

[53] 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.
[pdf].

[54] Into the depths of C: elaborating the de facto standards.

[56] P0668R5: Revising the C++ memory model, Hans-J. Boehm, Olivier Giroux, and Viktor Vafeiadis.

WG21 wg21.link/p0982, November 2018.

[58] Bridging the gap between programming languages and hardware weak memory models.
Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis.
[url].

[59] Compiler testing via a theory of sound optimisations in the C11/C++11 memory model.
Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli.
[url].

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].

[61] Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it.
Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli.
[url].

[62] 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.
[url].

Contents 8 References: 527
[63] Repairing sequential consistency in C/C++11. 
Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 
[url].

[64] Programming Languages — C++. 
P. Becker, editor. 
2011. 

[65] A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. 
Jean Pichon-Pharabod 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].

[66] Explaining Relaxed Memory Models with Program Transformations. 
Ori Lahav and Viktor Vafeiadis. 
[url].

Viktor Vafeiadis and Chinmay Narayan. 
[url].

[68] Promising 2.0: global optimizations in relaxed memory concurrency. 
[url].

[69] Modular Relaxed Dependencies in Weak Memory Concurrency.
Pomsets with Preconditions: A Simple Model of Relaxed Memory.
Radha Jagadeesan, Alan Jeffrey, and James Riely.
In Proceedings of OOPSLA.

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.

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.

Remote-scope promotion: clarified, rectified, and verified.
John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson.

Exposing errors related to weak memory in GPU applications.
Tyler Sorensen and Alastair F. Donaldson.

Portable inter-workgroup barrier synchronisation for GPUs.
Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric.
[76] Repairing and mechanising the JavaScript relaxed memory model.
[url].

[77] Weakening WebAssembly.
Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod.
[url].

[78] Bounding data races in space and time.
Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy.
[url].