Relaxed Memory

Multicore Semantics course notes

Peter Sewell **Shaked Flur**

Working draft of 2026-02-21

Contents

Acknowledgements 6

Reading guide 7
Readership . . . 7
Structure . . . 7
Use as course material . . . 8

1 Introduction 9
1.1 Memory . . . 9
1.2 Out-of-order and speculative uniprocessors . . . 10
1.3 Shared-memory multiprocessors . . . 10
1.4 Sequential consistency . . . 11
1.5 Running the example experimentally, on hardware . . . 12
1.6 Architecture specifications . . . 15
1.7 Programming language compiler effects . . . 20
1.8 Programming language specifications . . . 21
1.9 Status of the models . . . 22
1.10 Exercises . . . 23

I SC, x86, tools, and approach 24

2 x86 basic phenomena 25
2.1 Litmus tests and candidate executions . . . 25
2.2 SB: store buffering? . . . 26
2.3 LB: load request buffering? . . . 27
2.4 MP: message passing? . . . 28
2.5 SB+rfi-pos: write buffers with read-back? . . . 29
2.6 IRIW: independent reads of independent writes? . . . 30
2.7 WRC: write-to-read causality? . . . 31
2.8 SB+mfences: restoring order with fences . . . 32
2.9 Read-modify-write instructions . . . 33
2.10 Synchronising power of locked instructions . . . 34
2.11 Exercises . . . 35

3 x86: some vendor documentation history 36
3.1 pre-IWP (before Aug. 2007) . . . 36
3.2 IWP/AMD3.14/x86-CC . . . 37
3.3 Intel SDM rev. 29–34 (Nov. 2008–Mar. 2010) . . . 38
3.4 AMD APM version 3.15 (Nov. 2009) . . . 39
3.5 Intel SDM rev.80 (June 2023) . . . 39
3.6 AMD APM 4.07 (April 2020) . . . 41

4 x86-TSO: creating a good de facto standard model 43

5 Operational and axiomatic concurrency model definitions 45

6 SC, operationally 47
6.1 An operational SC model . . . 47
6.2 Exercises . . . 50

7 x86-TSO, operationally 51
7.1 An operational x86-TSO model . . . 51
7.2 x86-TSO operational example: SB . . . 53
7.3 x86-TSO operational example: spinlocks . . . 55
7.4 Discussion . . . 62
7.5 Exercises . . . 64

8 Making operational models executable as a test oracle: RMEM 65
8.1 Exercises . . . 67

9 SC, axiomatically 72
9.1 Execution graphs, informally . . . 72
9.2 Execution graphs, formally . . . 73
9.3 Coherence . . . 75
9.4 An axiomatic SC model . . . 83
9.5 Equivalence of the operational and axiomatic SC models . . . 84
9.6 Exercises . . . 87

10 x86-TSO, axiomatically 88
10.1 Coherence in x86-TSO . . . 88
10.2 Local ordering and the external relations . . . 90
10.3 An x86-TSO axiomatic model, without `MFENCE` and LOCK'd instructions . . . 90
10.4 x86-TSO axiomatic examples . . . 91
10.5 Equivalence of the operational and axiomatic x86-TSO models, without `MFENCE` and LOCK'd instructions . . . 91
10.6 Relational algebra Cat notation for axiomatic model definitions . . . 92
10.7 An x86-TSO axiomatic model, with LOCK'd instructions and `MFENCE` . . . 93
10.8 Equivalence of the operational and axiomatic x86-TSO models . . . 94
10.9 Exercises . . . 95

11 Making axiomatic models executable: Herd and Isla 96
11.1 Exercises . . . 100

12 Running tests on hardware: Litmus 101

13 Test families and test generation: Diy 103
13.1 Organising tests . . . 103
13.2 Generating single tests from cycles . . . 105
13.3 Generating families of tests . . . 105

14 Validating the model: why should one believe it? 108
14.1 Sound with respect to existing hardware: experimental validation . . . 108
14.2 Sound with respect to future hardware; loose enough to permit future microarchitectural innovation . . . 110
14.3 Opaque with respect to hardware implementation detail . . . 110

14.4 Complete with respect to hardware . . . 111
14.5 Strong enough for software . . . 111
14.6 Precise and unambiguous . . . 111
14.7 Clear . . . 111
14.8 Executable as a test oracle . . . 111
14.9 Incrementally executable . . . 111
14.10 Mathematically validated . . . 112
14.11 Authoritative . . . 112
14.12 Accurately capturing the architectural intent . . . 112
14.13 Consistency with the de facto standard . . . 112

II Arm-A, IBM Power, and RISC-V 113

15 Introducing Arm-A, IBM Power, and RISC-V relaxed concurrency 114
15.1 Architectures and Implementations . . . 114
15.1.1 Arm-A . . . 115
15.1.2 IBM Power . . . 116
15.1.3 RISC-V . . . 117
15.2 Relaxed behaviour and abstract microarchitecture, informally . . . 117
15.2.1 Microarchitecture optimisations and relaxed architecture specifications 118
15.2.2 The pros and cons of relaxed architecture specifications . . . 119
15.2.3 Abstract microarchitecture – structure . . . 119
15.2.4 Abstract microarchitecture – behaviour . . . 123
15.3 Relaxed architecture design principles and choices . . . 125
15.4 Litmus tests . . . 125
15.4.1 Candidate executions . . . 126

16 Arm-A, IBM Power, and RISC-V phenomena 129
16.1 Non-mixed-size Phenomena . . . 129
16.1.1 Coherence . . . 129
16.1.2 Out-of-order execution . . . 132
16.1.3 Dependencies . . . 136
16.1.4 Speculative execution - branching . . . 142
16.1.5 Instruction Barrier . . . 145
16.1.6 Write forwarding . . . 146
16.1.7 Speculative execution - restarts . . . 147
16.1.8 Satisfy same address reads out-of-order . . . 150
16.1.9 Write forwarding from a non-speculative write . . . 151
16.1.10 Multi-step read satisfaction . . . 153
16.1.11 Detour . . . 156
16.1.12 Write subsumption . . . 156
16.1.13 Symbolic forwarding . . . 160
16.1.14 Multi-Copy Atomicity . . . 161
16.1.15 Atomic Memory Modification . . . 166
16.1.16 Release/Acquire Memory Accesses . . . 169
16.2 Mixed-Size Memory Accesses . . . 172
16.2.1 Reading from Multiple Writes . . . 173
16.2.2 Mixed-size Coherence . . . 174
16.2.3 Single-copy Atomicity . . . 175
16.2.4 Atomicity of register accesses . . . 179
16.2.5 Mixed-size Multi-copy Atomicity . . . 181

16.2.6 Mixed-size write-forwarding 183

III Systems concurrency **187**

IV Reflections, related work, and history **188**

References **189**

Acknowledgements

Many thanks are due to all those who have contributed to the research underlying this text, both in collaboration with the authors and separately, especially (in roughly chronological order) Susmit Sarkar, Francesco Zappa Nardelli, Jade Alglave, Thomas Braibant, Scott Owens, Tom Ridge, Magnus Myreen, Luc Maranget, Jaroslav Ševčík, Anthony Fox, Samin Istiaq, Mark Batty, Kathryn E. Gray, Tjark Weber, Kayvan Memarian, Sela Mador-Haim, Rajeev Alur, Milo M.K. Martin, Gabriel Kerneis, Dominic Mulligan, Ali Sezgin, Kyndylan Nienhuis, Robert M. Norton, Jon French, Alasdair Armstrong, Thomas Bauereiss, Prashanth Mundkur, Mark Wassell, Brian Campbell, Neel Krishnaswami, Ian Stark, Ben Simner, and Thibaut Pérami.

This work would not have been possible without our main industry collaborators: Derek Williams (IBM); Richard Grisenthwaite, Will Deacon, Alastair Reid, and Graeme Barnes (Arm); Hans Boehm, Paul McKenney, and other members of the C++ concurrency group; and Daniel Lustig and other members of the RISC-V concurrency group.

Thanks also to Ori Lahav and Viktor Vafeiadis for discussion of the current models for C/C++, to Paul Durbaba for his 2021 Part III dissertation mechanising the x86-TSO axiomatic/operational correspondence proof, and to many others for discussions, including: Alan Stern, Andy Glew, Anthony Williams, Clark Nelson, Dave Dice, David Christie, Doug Lea, Gil Neiger, Jasmin Blanchette, John Baldwin, John Wickerson, Keir Fraser, Konrad Slind, Lawrence Crowl, Michael Fetterman, Michael Wong, Mike Gordon, Nathan Chong, Nick Maclaren, Paul Loewenstein, Peter Dimov, Raul Silvera, Robert N.M. Watson, Warren Hunt, and William Collier.

This text is partly based on slides for the semantics part of the University of Cambridge MPhil and Part II/III course *Multicore Semantics and Programming*, given by Sewell and Pulte from 2010 to date. Thanks to all the students who have tested the exposition, and to Tim Harris who has given the Concurrent Algorithms part, which is not covered here.

This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee (ERC AdG SAFER, EP/Y035976/1, Sewell). This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC AdG ELVER, grant agreement No 789108, Sewell). This work was partly supported by 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 (Postdoctoral 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 was supported in part by 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.

Reading guide

Readership

Relaxed memory is both technical and pervasive, cutting across many areas and communities. We aim for this text to be broadly useful to all these groups:

- hardware designers and validation engineers, for building processors that conform to a specific concurrency model;
- programming language implementers, for building language implementations that conform to a specific programming language model, above one or more architecture models;
- concurrent programmers, for writing software that is correct above the underlying architecture or language-level model;
- hardware architects and programming language designers, for precisely specifying the allowed behaviour of their hardware architectures and programming languages;
- semantics and verification researchers, for building theory and tools to reason about concurrent hardware and software with respect to these models;
- advanced undergraduate and postgraduate students; and
- those who teach this material.

It cuts across microarchitecture, architecture, programming languages, concurrent algorithms, concurrent programming, and mathematical semantics. Some background in any of these would help, but very few readers will have background in all of them, so we have tried to write in a way that will make sense to anyone.

To precisely define the allowed behaviour of each concurrency model, we use some discrete mathematics that, although simple, may not be familiar for all readers. We include brief introductions to the notation as we go, to make this somewhat self-contained, but it's designed to still make sense if one skips over the more mathematical parts.

Relaxed memory does involve quite a number of subtle phenomena, and not all readers will need to know about all of them. This is designed to serve both as a tutorial, introducing the most important issues first, and as a reference, covering the phenomena and models in detail. We recommend reading the former chapters first, without going into all of the latter, and then going back to those as needed.

Structure

Chapter 1 gives a brief and informal introduction to relaxed-memory concurrency, in architectures and programming languages, and to the role of precise models in defining what concurrent behaviour is allowed by particular abstractions.

We begin the main text in Part I with the simple (but normally unrealistic) sequential consistency model (SC), and the x86 architecture model. x86 has a simpler and stronger model (less relaxed, closer to SC) than the other main architectures we consider, so it provides a good context to see some of the basic phenomena and the techniques and tools we use for modelling and validation, before we get to the complexities of those other architectures.

In Part II we continue with Arm-A, IBM Power, and RISC-V. These are broadly similar to each other, though not identical; they are much more relaxed than x86.

In both Part I and Part II we start with the relaxed-memory phenomena that the architecture exhibits, explaining them informally with litmus-test examples, and discussion of how

the observable behaviour might arise from microarchitectural design choices. We then describe how one can express the architectural intent in general, with precise models that define what behaviour is allowed for arbitrary code. We do this using models in two complementary styles: *abstract-microarchitectural operational models*, that let one explain and understand the behaviour based on intuitions from hardware implementation, and *axiomatic models*, that more concisely (but perhaps less intuitively) define the allowed behaviour. Both are expressed mathematically, but we explain the small amount of required maths along the way, to make this as broadly accessible as possible.

In both these parts we focus just on the "user" architecture, including loads, stores, read-modify-write operations, barriers, and other synchronisation mechanisms, on coherent write-back memory, sufficient for many concurrent algorithms. This excludes all "systems" aspects: other memory types and "non-temporal" operations, self-modifying code and instruction/data-cache maintenance, virtual memory and TLB maintenance, exceptions and interrupts, and device memory. Some of these are now reasonably well-understood for some architectures, and we may cover them in a future revision; others remain open research questions.

For now, the text covers only this "user" architecture-level concurrency. Future versions may also cover programming language concurrency, "systems" architecture-level concurrency, and the history of the subject.

Use as course material

As mentioned, this text is partly based on the semantics part of the University of Cambridge Computer Laboratory course *Multicore Semantics and Programming*. This is both a masters-level course, in the MPhil in Advanced Computer Science, and a 3rd- or 4th-year undergraduate course (Part II/Part III in Cambridge terms). The aim of the semantics part is to give the students a good understanding of the basic architecture-concurrency phenomena and how they are captured in operational and axiomatic models, along with the general issues in programming-language relaxed concurrency and a briefer introduction to the C/C++ model. The slides are available online. The semantics part of the course is 5 two-hour blocks, divided into 10 one-hour sessions roughly as below.

1–2	Chapters 1–5:	introduction and x86 basic phenomena
3–4	Chapters 6–10:	operational and axiomatic models for sequential consistency and x86
5	Chapters 11–14:	validating the models
6	Chapter 16:	Arm-A, IBM Power, and RISC-V – the basic phenomena
7	Chapter 16:	Arm-A, IBM Power, and RISC-V – a sample of further subtleties
8–9		Arm-A, IBM Power, and RISC-V – operational and axiomatic models
10	Chapters **??**–**??**	introduction to programming-language relaxed concurrency

One could also pull out a briefer treatment, e.g. for a one-hour lecture within a computer architecture, concurrent programming, and semantics course.

Chapter 1

Introduction

This text addresses a fundamental question for concurrent programming: *what is the programmer-visible behaviour of memory?* We look at this for several mainstream processor architectures: x86, Arm-A, IBM Power, and RISC-V; and for higher-level programming languages, especially C and C++. We discuss the main phenomena by example, explaining how they arise from microarchitecture and compiler implementation optimisations (though this is not a text on hardware or compiler design); show how one can precisely specify real-world *relaxed memory models*, in operational and axiomatic styles, that define the allowed behaviour of concurrent programs; discuss how models can be validated and implementations tested; and consider the relationships between models. These models give a solid basis for high-performance concurrent programming, criteria for hardware and compiler implementation correctness, and a foundation for semantic reasoning and verification tool building.

This is largely based on a line of research from 2008–2024 by the authors and many others, including key industry colleagues; it builds also on much earlier research. We mention a few key points of this context as we go along, but the main narrative describes the current state of the art, as we understand it, not the historical development. In a future version we aim to include a more detailed survey of the related work.

1.1 Memory

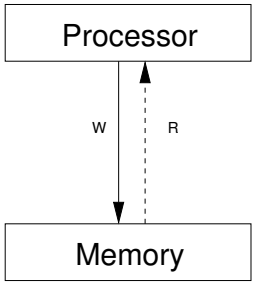
The abstraction of *memory*, and the organisation of computing devices into separate memory and processing components, dates back to the beginning of general-purpose computing. In 1837 Charles Babbage, describing his planned but unrealised Analytical Engine, wrote:

> *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.*
>
> [*On the Mathematical Powers of the Calculating Engine,* Charles Babbage [42]]

Here the store was to be able to hold 1000 numbers, each 40 decimal digits, while the mill was to be able to compute the four arithmetic operations and comparisons, controlled by a program on punched cards. Alan Turing's 1936 mathematical model of general-purpose computing [134], intended to characterise what is in principle computable, had an unbounded *tape*, divided into squares each holding one of a finite number of symbols, an *m-configuration* from a finite set of states, and a finite table determining the steps of the machine. The late 1940s and early 1950s saw the first general-purpose stored-program electronic computers, including Mauchly and Eckert's EDVAC (the design of which is detailed in von Neumann's *First draft of a report*

on the EDVAC [98, 135]), Turing et al's ACE [133, 139], the Manchester Baby [138], and the University of Cambridge EDSAC [137]. Each of these executed instructions sequentially in the order they were given in the program, issuing each required memory access to the memory and waiting for the memory to complete its operation before continuing.

Semantically, these are all straightforward: one can regard memory as an array or sequence of values, with the processor taking a step per instruction, in program order, and each step reading or writing (one location of) memory if necessary. Informally, this can be depicted as below.



(Properly characterising early machines involves many other questions, e.g. whether program and data memory are distinguished, and whether memory addresses must be static in the program or can be dynamically computed, but those are not relevant here.)

1.2 Out-of-order and speculative uniprocessors

Early high-performance machines in the 1950s and 1960s, such as the ILLIAC II, IBM Stretch, CDC 6600, and IBM System/360 Model 91, introduced a range of sophisticated hardware execution optimisations: *pipelined execution*, to simultaneously execute the successive stages of instructions, *superscalar execution*, to simultaneously dispatch multiple instructions to different execution units, *out-of-order execution*, to let program-order-later instructions go ahead when some of their program-order-predecessors are blocked waiting for some resource, such as a memory read, and *speculative execution*, to let instructions go ahead before it is known for sure that they will be reached, with roll-back as needed.

These let the hardware automatically extract some of the instruction-level parallelism (ILP) implicit in the instruction stream, but, at least in that uniprocessor context, they preserved the simple sequential programmer's model shown above: the hardware could do some out-of-order and speculative execution, but it would prevent or roll back anything that violated that model, so the programmer could still assume that instructions were executed one at a time in program order (apart from some imprecise exception cases).

1.3 Shared-memory multiprocessors

Shared-memory *multiprocessors*, with multiple processors that can interact via a shared memory, date back at least to the 1962 Burroughs D825, and to the 1972 IBM System 370/158MP. For many years they remained of relatively niche interest, as continued sequential performance increases came from smaller transistor sizes, increased clock frequencies, and better extraction of instruction-level parallelism by advanced processor designs. In the early 2000s those increases slowed, as they reached power-density, scaling, and ILP limits, and multiprocessors started to become commonplace, e.g. with the 2005 introduction of the Intel Core 2 Duo. Now they are ubiquitous.

Shared-memory concurrency has thus long become mainstream — but understanding, designing, and programming concurrent systems remains very hard. We focus here on the shared-memory abstractions provided by general-purpose processors and programming languages, but

similar issues arise in (and similar techniques and models can be used for) the abstractions provided by GPUs and other accelerators, by supercomputer interconnects, and in datacentre-scale and internet-scale distributed databases: wherever one needs multiple concurrent computation over the same shared data, and the need for performance means that everything cannot be simply synchronised.

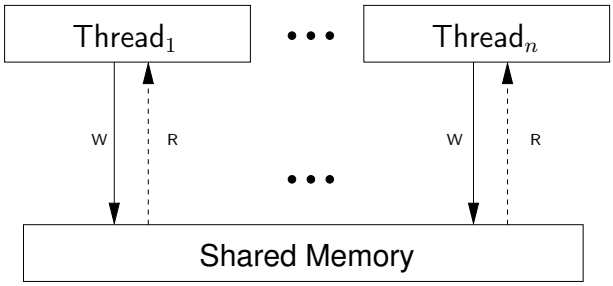
Sometimes shared memory is presented in opposition to message-passing models of concurrency, but in reality the two are tightly intertwined. Modern high-performance interconnects provide a programmer-level shared-memory abstraction above message-passing hardware, and distributed databases provide a shared-memory abstraction above internet message passing.

1.4 Sequential consistency

The most obvious semantics for a shared-memory multiprocessor is *sequential consistency*, as articulated in 1979 by Lamport [86]. A machine has a sequentially consistent (SC) shared memory if:

> *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*

or, in other words, if each processor appears to execute in order, with their effects on memory interleaved in some arbitrary way. One can informally depict the programmer's model for such a machine as below.



Much previous theory research has assumed this SC model, and probably many programmers still do. To the best of our knowledge, the Burroughs D825 did behave like that, in the 1960s, but more recent machines do not. To investigate how modern multiprocessors actually behave, we start with a very simple example, of a shared-memory program intended to enforce mutual exclusion between two critical sections:

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

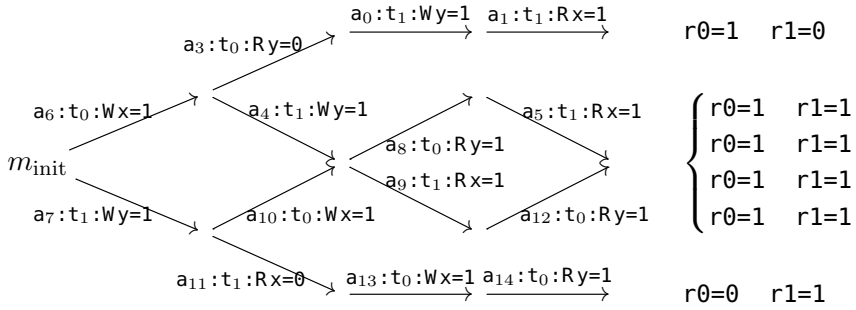
For the moment, we'll use a C-like pseudocode and leave the per-thread semantics implicit, just regarding each thread as an automoton (a labelled transition system) that performs a sequence of memory accesses. This example is already highly simplified with respect to practically useful algorithms for mutual exclusion: it's a one-shot algorithm (not supporting repeated use), it's for exactly two threads, it's inline rather than abstracted into a library, and it may have poor performance and fairness properties – but none of that is relevant right now.

Intuitively, one might think that the algorithm is correct iff the two threads cannot both be executing their critical sections at the same time. However, that is a very intensional property – it's phrased in terms of the internal details of how execution happens, which is not directly observable, rather than the extensional programmer-visible behaviour – and it refers implicitly to a notion of global time, which we will see is problematic. We therefore simplify it still further, replacing the conditionals and their critical-section bodies with just reads of the shared variables:

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

(writing r0 and r1 for thread-local variables) and ask whether a final state with r0=0 and r1=0 is possible, instead of whether the threads can be executing their critical sections at the same time. Small concurrency test cases like these are known as *litmus tests*, as they indicate what behaviour a model or implementation might allow.

For such a simple example, one can enumerate all the possible interleavings in the SC model, with the final states of each. There are six possible SC interleavings – the six possible paths through the transition system below – leading to three distinct final states:



The initial memory state m_{init} has x=0 and y=0, and each memory read or write transition is labelled with a unique event ID a_i, its thread t_j, and its address and value. We elide the details of the intermediate memory states. None of the final states have r0=0 and r1=0, so in the SC model this code does enforce mutual exclusion. In the SC model one can also reason more briefly: because in any SC execution there must be a total order over all the operations, and each thread executes in program order, either the x=1 or y=1 must go first, then because there are no other writes, the other thread's read of that must see that value, so one cannot end up with r0=0 and r1=0.

1.5 Running the example experimentally, on hardware

Now we'll run the example on some modern multiprocessors, to see experimentally what actually happens. To focus on the behaviour of the hardware, rather than the combination of the hardware and some compiler, we'll first rewrite the test in assembly code, to ensure that we know exactly what is being tested. Here are x86 and Arm-A (AArch64) versions of the test:

SB x86

Initial state: 0:rax=0; 1:rax=0; y=0; x=0;	
Thread 0	Thread 1
movq $1, (x) //W x=1 movq (y), %rax //R rax=y	movq $1, (y) //W y=1 movq (x), %rax //R rax=x
Final: 0:rax=0; 1:rax=0;	

SB AArch64

Initial state: 0:X3=y; 0:X1=x; 0:X0=1; 0:X2=0; 1:X3=x; 1:X1=y; 1:X0=1; 1:X2=0; y=0; x=0;	
Thread 0	Thread 1
STR X0,[X1] //W x=1 LDR X2,[X3] //R R2=y	STR X0,[X1] //W y=1 LDR X2,[X3] //R R2=x
Final: 0:X2=0; 1:X2=0;	

Assembly syntax In these architecture-level litmus tests, there is an assembly language program for each hardware thread, which run concurrently. Each per-thread program is a list of assembly instructions, which map fairly straightforwardly onto the machine-code instructions that processors actually execute, and labels that identify program points. For x86, two styles of syntax are in widespread use: *Intel syntax* (used in the Intel manuals) and *AT&T* syntax (dominant in Unix environments). These x86 litmus tests use a variant of the latter: the movq $1,(x)

is a 64-bit (quad-word) store of constant `1` to location `x` and `movq (y),%rax` is a 64-bit load of location `y` into 64-bit register `rax`. Intel syntax and the Arm-A syntax we use have the opposite argument order to that, destination before source. In the Arm-A test, which uses the Arm-A AArch64 64-bit instruction set, `MOV W0,#1` copies the constant `1` into register `R0`. The AArch64 general-purpose registers `R0...R30` can be referred to either as 64-bit registers, using `X0...X30`, or 32-bit registers, using `W0...W30`. These names determine the sizes of stores and loads: the `STR W0,[X1]` stores the low-order 32-bit contents of `R0` into the memory location addressed by `R1`, while the `LDR W2,[X3]` loads a 32-bit value from the memory location addressed by `R3` into `R2`, zero-extending it.

Then we need a test harness, to run the test repeatedly and log the outcomes. We'll use the `litmus7` tool from the `herdtools7` tool suite by Alglave and Maranget [30]. To install `litmus7`:

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

This uses a common format for tests, shared with the other tools of that suite, `diy7` and `herd7`, and with the `rmem` [117] and `isla-axiomatic` [38] tools by Flur et al. and Armstrong et al.; we'll return to all these later. In that format, the x86 version of this test can be written as `SB.litmus`:

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

Here the initial `X86_64` specifies the test architecture, lines 3–6 define the initial state of memory (`x` and `y`) and registers (`0:rax` and `1:rax` for the `rax` register of hardware threads 0 and 1 respectively), and lines 7–9 give a sequence of x86 assembly instructions for two threads, `P0` and `P1`. Line 10 specifies the condition on the final state that we're interested in.

Running that with `litmus7` executes the test in an aggressive test harness: it converts the test to actual assembly, runs many instances of the test (10^6 times by default), iterates in randomised orders over arrays for `x` and `y` instead of scalars, roughly synchronises the start times of the threads in each instance, and so on. The embedded assembly kernel it actually generates for the x86 version of this test is below – one can see that the accesses to `x` and `y` use an indexed addressing mode to allow convenient iteration over arrays of locations.

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

It logs a histogram of the observed final values. For a typical x86 processor we might see something like this, though the exact numbers will vary:

```
$ 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
[...]
```

Here we see that in most executions it appears that Thread 0 runs entirely before Thread 1 or vice versa, and in a few executions (54 out of 10^6) it appears that the two stores happened before the two loads. Those are all allowed by the SC model. But in 14 out of 10^6 executions, we see the non-SC behaviour in which both threads' reads see `0`. This is a rather common non-SC behaviour; to see rarer non-SC behaviours one might need to run tests for $10^9 \ldots 10^{11}$ executions, and tune the litmus test harness with its command-line options.

If we run an Arm version of the test we see similar results:

```
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).

A priori, there are many possible explanations for the mismatch between these experimental observations and the SC model:

1. There is an error in the test – i.e., it isn't testing what we think it is. It is quite easy to make errors when hand-writing subtle tests, so this needs care, but if there's a discrepency between the experimental observations and the model, then that's an issue exposed by the test as written irrespective of what we think it's testing.

2. There is an error in the test harness. This is certainly possible, but `litmus7` is reasonably mature, and mismatches can almost always be traced to another cause. One can also gain some confidence in the harness from the fact that in extensive use for many tests it does give the expected results.

3. There is an error in the surrounding OS that is corrupting the test results. This is possible in principle, but we have not seen it in practice. If this were an issue one would expect to probably see it even for straightforward tests, which we do not.

4. There is an error in the hardware processor design. This is rare but certainly happens. Over the years this testing has identified a number of hardware errata, in core designs from several vendors.

5. There is a manufacturing defect in this particular instance of the processor. One can check this by contrasting experimental data from different instances of the same processor.

6. There is an error in our calculation of what the model allows for the test. For subtle tests and models this becomes challenging to do reliably by hand, especially for large numbers of tests, so one would typically use tools (such as `rmem`, `isla-axiomatic`, or `herd`) to do this. Of course, one then has to be concerned with the possibility of bugs in those tools.

7. There is an error in the model. In this case this is the correct explanation:

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

In fact, SC is not a good model for any major multiprocessor architecture, including IBM Power, RISC-V, SPARC, or Itanium, or for major programming languages, including C, C++, and Java. Instead, all these have some form of *relaxed memory model* (or *weak memory model*), allowing some non-SC behaviour. In the remainder of this text we'll explain this, explore what programmers can actually depend on, and establish good models, for x86, Arm-A, IBM Power, and RISC-V.

As we shall see, hardware relaxed-memory behaviour arises from microarchitectural optimisations, including out-of-order and speculative execution, and sophisticated storage hierarchies and cache protocols, and programming-language relaxed-memory behaviour arises from the combination of this and established compiler optimisations. These hardware and software optimisations aim to leave the observable sequential behaviour unchanged from what one would expect in a straightforward in-order design, but some change the observable behaviour of concurrent code.

1.6 Architecture specifications

To understand what it means to be a "good model", and how we can conclude that SC is not a good model for x86 or Arm, we have to understand clearly what processor *architectures* are, how they relate to processor *implementations*, and what properties a good architecture definition should have.

Each specific processor implementation, such as an Intel i7-7700K, an AMD Ryzen 7 1800X, a Qualcomm Snapdragon 865, a Samsung Exynos 990, or an IBM Power 9 Nimbus, includes cores based on a specific *microarchitecture* and detailed design. We cannot use this internal structure of an implementation as a programming model: it is far too complex, it is generally commercially confidential, and it is too specific – most software has to run correctly on a range of similar implementations, past, present, and future, not just on one specific hardware implementation.

Processor implementations are grouped in families, e.g. the many variants of x86 processors developed by Intel, AMD, and VIA, and the many variants of Arm-A processors developed by Arm and its architecture partners (Apple, Qualcomm, Nvidia, Samsung, etc.). An *architecture specification* aims to define an envelope of the *programmer-observable behaviour* of all members of such a family: the set of all behaviour that a programmer might see by executing multithreaded programs on any implementation of that family and examining the results. It thereby defines the hardware/software interface, serving both:

1. as a criterion for correctness of hardware implementations: any observable behaviour of any conforming hardware implementation should be within that envelope; and

2. as a specification of what programmers can depend on: they should be able to assume that any behaviour outside that envelope does not occur.

Architectures have to be *loose* specifications, to allow runtime variations in behaviour for each single implementation (e.g. due to differences in timing), and to accomodate the observable variations arising from different microarchitectural design choices among current, past, and future implementations. As we shall see experimentally, these differences can be quite substantial. For both reasons, they typically constrain only "functional correctness" properties, not timing and performance behaviour, as those can vary widely within and between implementations (though this leads also to the possibility of side-channel attacks).

This looseness makes software development challenging: in principle one might think that programmers should and can "program to the architecture", but normal software development

relies on testing software running on specific processor implementations. The runtime variations may mean that program bugs are not exposed in some particular test runs, and testing may be done on a specific implementation which may not exhibit some architecturally allowed behaviour that other implementations do or will exhibit. This increases the need for models that one can reason about mathematically, to ensure (at least for critical software) that all cases are covered.

The main architecture specifications are defined by industry vendors or groups, and published as pdf documents. Intel and AMD produce x86 processors, as described in their

- Intel 64 and IA-32 Architectures Software Developer's Manual [62] and
- AMD64 Architecture Programmer's Manual [17].

For our purposes these effectively define a single architecture, though the documents differ. VIA also produce x86 processors.

Arm define three main architectures, A-Profile (for applications), M-profile (for microcontrollers), and R-profile (for real-time). We focus on the first, as described in their

- Arm Architecture Reference Manual, for A-profile architecture [14].

Arm design specific cores (e.g. the Cortex-A76), which other vendors licence and integrate into SoCs (e.g. the Samsung Exynos 990). Arm also licence the architecture itself, allowing other architecture-partner vendors to design their own conforming cores (e.g. the Qualcomm Kryo).

The IBM Power architecture specification is now maintained by the OpenPOWER Foundation:

- Power ISA Version 3.1C [73].

This is the architecture used for the IBM POWER10 processor.

The RISC-V architecture is defined by the RISC-V International organisation (formerly the RISC-V Foundation):

- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [113]
- The RISC-V Instruction Set Manual Volume II: Privileged Architecture [112]

with many vendors and academic groups designing RISC-V implementations.

There are, and have been, many other important multiprocessor architectures, of course. Some have had interesting or influential relaxed-memory behaviour or specifications, notably SPARC, Alpha, and Itanium, and we will refer to them along the way, but we focus on the above currently dominant architectures for application and server-class general-purpose processors.

Each architecture definition is factored into several parts. The largest is typically the definition of the encoding and sequential behaviour of individual instructions, the *instruction-set architecture* (ISA), which is typically intricate but semantically relatively straightforward. Then there is the *concurrency architecture*, defining how instructions in different hardware threads interact with each other and with whatever mechanisms are provided for cache management. The latter is our focus here, along with the interaction between these two. Historically, most architectures have defined both parts in prose text, often with informal pseudocode for instruction behaviour.

Architectures also change over time, of course (each of the above exists in a sequence of versions) but this tends to happen more slowly than the development of new processor implementations. It usually aims to ensure backwards compatability, so that code written for one version, that should run correctly on conforming implementations of it, will also run correctly on conforming implementations of a newer version. In some cases one is also concerned with the converse, e.g. where a range of implementation and architecture versions co-exist within the

same software ecosystem. Major changes to the concurrency models are relatively rare; we will discuss a few important cases below.

All this might lead one to think that an architecture specification cannot be "correct" or "incorrect" – that each is what it is, as defined by the organisation in control of it. In practice, however, there is a delicate interplay between these specifications and the de facto standard of the hardware implementations that exist, or that are being designed. There are many properties that a good architecture specification should have, which we now discuss.

Sound with respect to existing hardware An architecture specification should be *sound* with respect to current implementations: it should admit all the behaviour that they can exhibit. Cases where this is found not to hold, for implementation behaviours that are deemed erroneous, are often documented as processor errata, but in some cases the architecture is loosened to accommodate some important (and impractical to change) implementation. Implementations often allow some boot-time configuration that allows specific optimisations to be turned off or simplified, which sometimes allows errata to be worked around.

In principle, soundness with respect to a particular hardware implementation could be established by mechanised mathematical proof of the correctness of that implementation, say down to the RTL level. In practice, that is still infeasible for full-scale architectures and implementations (though see e.g. [76] for recent work in that direction), and one instead gains partial confidence by extensive testing. Test results can conclusively demonstrate unsoundness, however – as we just did for the hypothesised SC model for x86.

Sound with respect to future hardware An architecture should also be loose enough to permit future microarchitectural innovation. That is hard to predict, of course, so this creates an incentive for the architecture to be as loose as possible, which can be in tension with the need for it to be strong enough for software and simple enough to understand.

Opaque with respect to hardware implementation detail In general an architecture should not unnecessarily expose any microarchitectural implementation detail, to keep the programmer's model simple, to avoid over-committing to some specific implementation and thereby excluding future improvements, to avoid skewing the thinking of hardware implementors, and to avoid revealing confidential information.

Complete with respect to hardware Architectures do *not* aim to be complete with respect to hardware, i.e. to capture all the functional properties of programmer-observable behaviour for specific implementations, for exactly the reasons above that mean that they have to be loose specifications.

Strong enough for software In the other direction, an architecture specification should be *strong enough* to support the intended corpus of existing software, and all desired programming idioms. A specification that allowed arbitrary behaviour would be trivially sound but not useful.

Showing rigorously that an architecture is strong enough to be useful is even more challenging than proving that one is sound with respect to a hardware implementation. One would like to argue that it is strong enough to support correctness proofs of all the key software that should run above it, but that is very far from the state of the art – and essentially all software is buggy in any case. In practice this is generally left to informal argument, or even left entirely implicit. However, for the concurrency model aspects of an architecture, what one can do is prove that a particular compilation scheme from a higher-level language (with its own concurrency model, and therefore its own strong-enough concern) is correct, and tensioning the two against each other like this has been very important in the design of both. We return to this later.

In principle one could experimentally assess whether an archicture is strong enough by testing a substantial body of software above an architecturally aggressive emulator, that pseudo-randomly chooses executions that exercise the extremes of what the architecture allows. For example, for x86 one could have an emulator that has very large write buffers, and flushes them roughly synchronised with data races in the code. Our understanding is that this has been done to some extent within several vendors, but we are not aware of such work for more relaxed architectures.

Precise and unambiguous For an architecture to serve its dual purposes, as a correctness criterion for hardware and a programming model for software, it should be *precise*: for any hypothetical programmer-observable behaviour, of any initial machine state, it should unambiguously define whether that behaviour is allowed or forbidden. Of course, in some cases this might depend on a specific architecture version, on some documented implementation-defined choice, or on some aspect of the architected (programmer-visible) machine state.

Ambiguity and imprecision should not be confused with looseness. On the contrary, the fact that architectures have to be loose specifications creates even more need for precision: for a subtle loose specification, it is essential that it precisely defines what is allowed and what is not, rather than leaving that vague.

The prose and pseudocode of most traditional architecture descriptions are generally reasonably precise about sequential behaviour but much less clear when it comes to concurrent behaviour and relaxed-memory phenomena — it is very hard to produce prose that unambiguously and completely captures these subtleties. Instead, we will develop models in *precise mathematics*.

Clear Architecture specifications have to be *clear*. They serve as a principal means of communication between the hardware and software development communities, and as such have to be comprehensible to both parties. This creates an incentive for an architecture specification to be as simple as possible – which is sometimes in tension with the desire to admit particular hardware optimisations.

One might think that the need for model definitions to be mathematically precise is in conflict with the need for them to be comprehensible to a wide audience, but in practice one can reconcile the two, by transcribing the mathematical definitions into prose (manually or automatically), and by providing tools that compute the allowed behaviour of test cases and let users explore them.

Another interesting tension relating to clarity is between models that are as concise as possible, just defining what whole-program behaviour is allowed or not, and models that are *explanatory*, that somehow explain how the allowed behaviour emerges from hardware optimisations. Several quite different styles of model definition are used – operational, axiomatic, and promising – which we return to in Chapter 5 and later. Typically (though there can be exceptions), operational models aim to explain the allowed behaviour in microarchitectural terms, while axiomatic models aim to define the allowed executions more abstractly in terms of the allowed orderings.

Executable as a test oracle Another major shortcoming of prose specifications is that they cannot be used directly for testing hardware or software against. Instead, one has to rely on test suites with manually curated outcomes, and on simulators written based informally on the prose. In practice, major vendors typically have one or more “golden models”: executable simulators, written in-house and sometimes confidential, that they use to test against. However, as we understand it, these have often focussed just on sequential behaviour.

We will instead develop concurrency models that are *executable as test oracles*, i.e., that, given some (small) test and a potential observable result, can compute whether or not that

result is allowed by the model. In some cases they will have the stronger property that they are *exhaustively executable as a test oracle*: given some small test, they can compute the set of all the model-allowed results, as we did by hand for the sequential consistency model and the simple test above.

Given a model that is executable as a test oracle, an experimental setup (such as `litmus7`) for running tests on actual hardware, and a set of tests, one can automatically compare a model with the experimentally observable behaviour exhibited by specific hardware implementations, without having to manually curate the intended outcome for each test.

This is essential to support *experimental semantics* [51, 50]: the development of models informed and validated in part by such comparisons with experimentally observed behaviour, and to support experimental validation of hardware designs against such models. However, it is important to note that good architecture models cannot be developed solely by fitting to some experimental data, as they have to be loose specifications that also satisfy all the other properties in this list.

Having a model that is executable as a test oracle means that one can *generate* tests, either randomly or in some directed way, without having to manually curate the intended outcome of each; we return to this later.

Executable-as-test-oracle models can also be used directly to exhaustively model-check small concurrent algorithms.

Incrementally executable A different way in which an architecture model might be executable is *incrementally*, computing just single possible executions, e.g. by making pseudo-random choices at points where the specification is loose. This makes the model directly usable as an emulator, which may be useful for testing larger bodies of software for which it becomes infeasible to compute the set of all model-allowed outcomes. Note that such an emulator would be *architecturally complete*, in the sense that it can, pseudorandomly, generate any behaviour allowed by the architecture – in contrast to any specific implementation, and to conventional emulators such as QEMU.

Mathematically validated One typically cannot prove, mathematically, that an architecture specification is "correct", because there is no more authoritative statement of what "correct" would mean. However, proving results about a specification can be invaluable, both by establishing specific desired results, and boosting confidence in the detailed definition by exercising it in quite different ways to testing. Many different kinds of result can be useful, including:

- Equivalences between models in different styles, such as operational and axiomatic, for the same architecture.
- Refinements between models, showing that for programs that are in some sense well-behaved, a simpler model accurately captures all the observable behaviour of a more fundamental model.
- Correctness of compilation schemes or compilers, from a high-level language model or semantics down to an architecture-level model.
- Soundness of program logics, model-checking algorithms, and other techniques for reasoning about programs.

Authoritative Ideally, a mainstream architecture specification would also be *authoritative*, defined by the relevant vendor or industry group. Socially, for each of these major architectures, there is an individual or small group that (perhaps following consultation with architecture partners and software users) has the authority to decide whether some observed behaviour is a bug or not – this is the ultimate role of an architect, to decide the architectural intent.

However, in some cases the vendor or other industry group produces prose specifications that do not have the above properties, or simply has not yet considered all the issues, and there one might end up with a model by a third party that aims to capture – and perhaps becomes – the de facto standard.

Accurately capturing the architectural intent Where there is an architect or group with the social authority to decide the architectural intent, one would like to confirm that any formal specification accurately captures that intent, allowing no more and no fewer behaviours than they intend to be allowed.

That can be challenging, especially when moving for the first time from an architecture specified only in prose to a formal model in an initially-unfamiliar form, that may be quite intricate. One can gain significant assurance through walking through the model in detail, and also by checking it gives the intended allowed behaviour for a set of litmus tests that is small enough for this to be tractable but which still covers a wide range of behaviour.

However, the consequences of a formal model may not all be obvious, and some of them may simply never have previously been considered, by the architects or anyone else. Some of the relaxed phenomena that we discuss later were of this kind: they emerged only from the process of developing and evaluating a model, and that informed later decisions about the architectural intent.

In practice, all architecture specifications are subject to change, at least in detail, in the face of implementation and usage reality – nothing is set in stone forever. However, on the positive side, when one has an established model, test suite, and intended results for those tests, any change to the model can be evaluated with respect to all those, which is much more tractable than establishing an initial formal model.

Consistency with the de facto standard One also has to consider consistency with the de facto standard. For any architecture with significant hardware implementations and software usage, the behaviour of those implementations, and the assumptions on hardware implementation behaviour implicit in the corpus of software, can tightly constrain possible architectural choices.

1.7 Programming language compiler effects

At the language level, observable relaxed-memory behaviour arises from the *combination* of any hardware relaxed-memory effects, for the memory accesses in the generated machine code, and a diverse collection of compiler optimisations. Just as for hardware implementations, these have been developed over many decades to optimise performance while preserving sequential behaviour, but they 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, the message passing example below – another very important litmus-test shape – should work as expected. This shape is a simple version of a common idiom for passing data between threads, in which Thread 0 writes some data to `x` (perhaps a large data structure), before writing a flag value to `y` to signal to other threads that the data is ready, and Thread 1 busy-waits reading `y` until it sees the new flag value, and then reads the data from `x`. In SC, if the Thread 1 read of `y` sees the write of `y=1` then the program-order-later Thread 1 read of `x` will see the write of `x=1`; the program will either print nothing or 1.

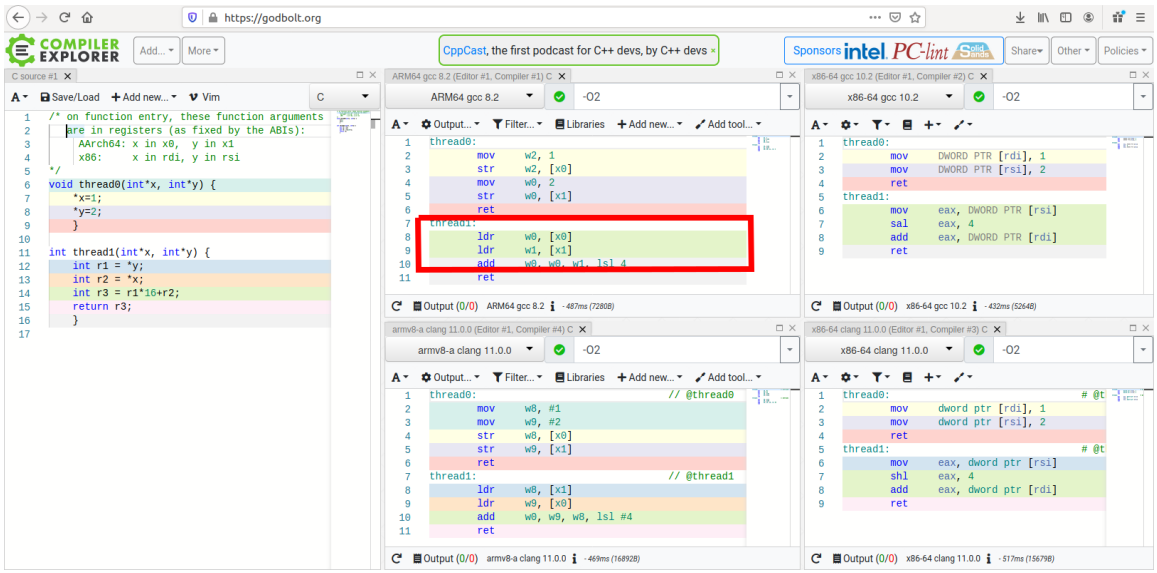


Figure 1.1: Compiler reordering

Thread 0	Thread 1
`x = 1` `y = 1`	`if (y == 1)` `  printf("%d", x)`

However, if there's some other read of `x` in the context, then common subexpression elimination could replace the second read by a reuse of the value from the first (which could easily be kept in a register), effectively reordering the Thread 1 reads of `y` and `x` to (the result of naively compiling):

Thread 0	Thread 1

`x = 1`
`y = 1`
	`int r1 = x`
	`if (y == 1)`
	`  printf("%d", r1)`

Compiler-introduced relaxed memory effects will typically be specific to some compiler version, optimisation level, and the details of the code. For example, in Fig. 1.1 ARM64 gcc 8.2 reorders the `thread1` loads, but the other compilations shown do not. One can see this in Compiler Explorer (short link) (full link).

This makes it harder to experimentally test what high-level language relaxed-memory behaviour is in practice, and, as we shall see, the interplay between hardware and compilation effects makes it harder to define reasonable concurrency models for high-level languages.

A general-purpose high-level language should provide a common abstraction over the common hardware architectures, that is efficiently implementable with respect to both:

- the cost of providing whatever synchronisation the language-level concurrency model mandates above those various hardware concurrency models; and
- the impact of providing the language-level model on existing compiler optimisations.

1.8 Programming language specifications

Programming language specifications play a similar role to architecture specifications, serving as a criterion for correctness of language implementations, and as a specification of what pro-

grammers in the language should be able to depend on.

Historically, also like architecture specifications, they have been expressed just in prose, despite many decades of research on more precise and less ambiguous mathematical semantics of programming languages, and the specification of concurrency behaviour has been particularly lacking. For example, a specification of concurrency was added to the ISO C and C++ language specifications only in 2011, following an effort led by Boehm [55, 43, 48].

They often differ from most architecture specifications in having a looser relationship between the organisation maintaining the specification and those implementing it.

Of particular interest here are the C++ and C standards, maintained by ISO/IEC JTC 1/SC 22/WG 21 and WG 14 respectively, e.g. with recent versions:

- ISO/IEC 14882:2017 Information technology — Programming languages — C++ [82]
- ISO/IEC 9899:2018 Information technology — Programming languages — C [83]

We will also touch briefly on Java, JavaScript, and WebAssembly.

1.9 Status of the models

We discuss the validation of the concurrency models we cover, with varying combinations of experimental testing, discussion with architects, and proof, as we go. To briefly summarise the origin and status of the models:

x86 For x86, the x86-TSO model we describe is widely accepted as the de facto standard. It was developed by Owens, Sarkar, Sewell, Zappa Nardelli, and Myreen, originally published in [101, 120]. This followed our earlier attempts [114] to interpret the then-confusing Intel and AMD documentation (the current vendor documentation [62, 17] is improved but still hard to interpret). It was informed by brief communications with Intel, AMD, and VIA architects. The operational and axiomatic models have been proved equivalent, first by Owens in HOL4 [101], and later, for a more modern presentation of the axiomatic model, by Durbaba in Isabelle [67].

IBM Power The IBM Power model we describe is also a good de facto standard to the best of our knowledge, capturing the architectural intent. It was developed based on detailed discussions with one of the senior IBM designers, Derek Williams, and extensive experimental testing on Power G5, 5, 6, 7, and 8. The vendor documentation [72] has not been updated to reflect the model design, however. The examples, discussion, and operational model that we present are some of the results of an extended line of work, mainly by Alglave, Flur, Maranget, Pulte, Sarkar, Sewell, and others, variously in collaboration or separately, that developed a series of axiomatic and operational models for IBM Power [20, 21, 116, 34, 23, 115, 92, 93, 36, 77, 71]. Much of the discussion text for this, Arm-A, and RISC-V is taken from the forthcoming PhD thesis by Flur.

Arm-A For Arm-A, we present operational and axiomatic models. The axiomatic model [64] was developed principally by Deacon, then at Arm, and was incorporated into the Arm documentation of the time [12]; the operational model was developed principally by Flur, Pulte, Sarkar, and Sewell. Both build on the above-cited work and [70, 71], and on extensive discussion between them and with Grisenthwaite (the Arm Chief architect). They incorporate a substantial simplification to the previous ARMv8-A architectural intent, switching to a "multi-copy atomic" model (as we describe later). The two models were proved equivalent by Pulte [106, 105]. Further academic work by Armstrong et al. integrated the axiomatic model with the full Arm-A instruction semantics [39, 40, 41], and work by Simner et al. developed extensions for some systems aspects: instruction fetch [125], virtual memory [124], and exceptions [123],

again in collaboration with Grisenthwaite. Further work within Arm by Alglave (now at Arm) and colleagues has refined the Arm documentation model in various respects [26].

RISC-V For RISC-V concurrency, the RISC-V Foundation (since reconstituted as RISC-V International) established its RISC-V memory model working group in 2017-03, chaired by Daniel Lustig [13]. Flur, Maranget, Pulte, Sarkar, and Sewell contributed substantially from 2017-07, together with several others, to what became the RISC-V weak memory ordering model (RVWMO), which was ratified in 2018-07. This is very similar (though not identical) to the multi-copy atomic Arm-A model. The RISC-V formal axiomatic and operational models were incorporated into the RISC-V unprivileged specification [74].

C/C++ For C/C++, the early language standards did not attempt to specify concurrent behaviour. There was an extended effort to define a model, from around 2004 onwards, led by Boehm and recorded in the `cpp-threads` mailing list [6] and the ISO C++ concurrency subgroup (which later became the ISO C++ Concurrency Study Group, WG21/SG1) and its working papers [1]. Boehm and Adve outlined the model under construction in 2008 [55]. From 2009 Batty, Owens, Pichon-Pharabod, Sarkar, Sewell, and Weber worked to formalise the design and build tools; this uncovered and led to fixes for a number of flaws in the model [43, 53, 46]. The ISO C++11 standard [48] incorporated text in tight correspondence to the mathematical model of Batty et al., and this was ported to the ISO C11 standard. More recent work by many authors has continued to improve the model, although the "thin-air problem", which we return to later, continues to be a major open problem for this and other high-level language models. A separate line of work by Alglave, Maranget, McKenney, Parri, and Stern defined a model for the concurrency primitives used in the Linux kernel [32, 33] (though this too has related open problems).

1.10 Exercises

Exercise 1.1 *Show that Peterson's algorithm ensures mutual exclusion for two threads, and Dekker's algorithm as well, under SC. For simplicity, consider that the loops execute only once.*

Exercise 1.2 *Write the interleavings of the following program:*
r1=x; y=r1+1 || r2=y; z=r2+1 || r3=z

Exercise 1.3 *Identify another program that exhibits non-SC behaviour on an Arm machine, by running it experimentally.*

Exercise 1.4 *Identify other non-SC behaviour induced by compilers, by finding a source program such that its optimised version exhibits behaviour under SC that the source program does not exhibit*

Exercise 1.5 *???*

Part I

SC, x86, tools, and approach

Chapter 2

x86 basic phenomena

Throughout this part we focus on a simple "user" fragment of x86, including loads, stores, read-modify-write operations, and barriers, but excluding misaligned and mixed-size accesses, string operations, and all "systems" aspects. We consider just coherent write-back memory, with no exceptions, "non-temporal" operations, self-modifying code, page-table changes, or device memory.

To help build a solid intuition, and to introduce some basic concepts and litmus tests that we'll elaborate on for other architectures and for programming languages, we'll approach the x86 behaviour incrementally: we'll discover the underlying model bit-by-bit with a few key examples.

2.1 Litmus tests and candidate executions

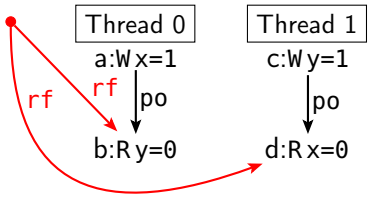
We've already seen one non-SC x86 behaviour, of the SB test on the left below, typeset from the `SB.litmus` file.

SB x86

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

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

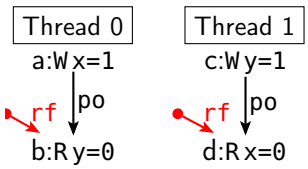
Final: `0:rax=0; 1:rax=0;`



On the right above we draw the *candidate execution* of this that we are interested in. Each candidate execution is a graph with nodes that are memory access events and with edges indicating various relationships between them. Each memory access, e.g. `a:`$\mathtt{t}_0$`:W x=1`, has a unique identifier (`a`, `b`, etc.) and a thread ID ($\mathtt{t}_0$, $\mathtt{t}_1$, etc.), and is either a write `W` or read `R` of a memory location (`x`, `y`, etc.) with a specific value (here `0` or `1`). Usually the thread IDs are clear from the context and we elide them.

The edges are *program order* (`po`, in black) and *reads-from* (`rf`, in red), together with other relations that we introduce later. Program order relates each event to its successors (if any) in some control-flow unfolding of the program. Here, there are no conditional branches, so there is only one possible control-flow unfolding, and program order just follows the program source text. Program order is transitive, but we normally draw only its transitive reduction – just the edges between adjacent events. Note that these graphs are quite different to the SC labelled transition system we saw in Section 1.4: that was a diagram of all possible SC interleaving executions, with nodes representing global states and arrows the transitions between them; while this shows the events of one particular (not necessarily SC) execution, as the nodes, with arrows indicating various relations between those.

The reads-from relation relates each write event to all the read events that read from it in the execution. In this execution, both reads read from the initial state, indicated with arrows from a red dot. To de-clutter the diagrams, we usually draw such a dot for each read from the initial state, as below, but they all represent the single shared initial state.



Each litmus test has a condition on the final state, e.g. the Final: `0:rax=0; 1:rax=0;` condition of this test. The final condition usually uniquely determines a single non-SC candidate execution of interest (under minimal assumptions on the model): for this test, the interesting execution is the only one in which both reads read `0` from the initial state (and the internal semantics of each instruction is respected). This execution is the one shown in the diagram. Note that the final condition only identifies some final state(s) of interest; it doesn't imply whether that state is observed or not experimentally on specific hardware implementations, or whether it is allowed or not in any specific model.

Litmus tests can be interesting for many different reasons: some capture the pattern of some important concurrent programming idiom, some may not arise in normal programming but are useful to explore exactly what is observed or allowed in implementations or models, and some arise from automatically and/or systematically generated families. Litmus tests can also be useful for identifying errors in hardware implementations, though most of those we shall see are not tuned for that (e.g. by stressing particular hardware resources); they are mostly aimed at concisely capturing some semantic choice, of what behaviour is or is not allowed.

2.2 SB: store buffering?

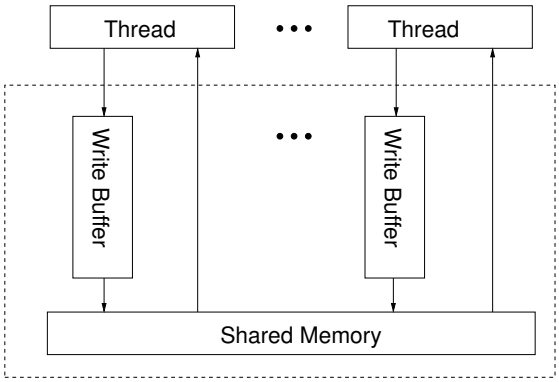
We saw in §1.5 that the non-SC final state of SB, `0:rax=0; 1:rax=0`, is experimentally observable on at least some x86 processor implementations, and in fact it's easily observable on all that we have tested, which include a number from both Intel and AMD.

As we saw in §1.4, this is not an SC-allowed final state. To understand how it might arise, one can consider the microarchitectural hardware-implementation optimisations that would permit it. There are two obvious possibilities:

1. Hardware often *buffers* writes, decoupling the execution of program-order-later instructions from the propagation of the write down to memory. That propagation can take a long time, and it is often not necessary (for the correct execution of a program) for the program-order-later instructions to wait for it, so this can provide a big performance gain.

2. Hardware often executes instructions *out of order*, subject to checks on their dependencies. This allows, for example, later instructions to go ahead while earlier instructions are waiting for read values to come back from memory.

Either could explain the experimental data for this test. For 1, in a processor implementation with write buffers for each hardware thread, as in the cartoon microarchitecture below (which we will later see is not quite right), the writes to `x` and `y` might still be in the Thread 0 and Thread 1 store buffers, respectively, when the reads of `y` and `x` get satisfied from memory.



For 2, because the load instructions are to different locations to the program-order-preceding store instructions, and have no register dependencies on them, they might be executed first.

With what we know so far, we can't tell whether either one of these, or both, or something else, is the true explanation for the experimental observations.

To understand the architectural intent – to know whether the experimental observation is intended by the vendors to be allowed – we can turn to the vendor documentation. In this case that is clear. The Intel SDM [62, Vol.3A, §8.2.2] says

> *Reads may be reordered with older writes to different locations but not with older writes to the same location*

and their Example 8-3 is essentially identical to this litmus test, with the final state given as allowed, while the AMD APM [17, Vol.2, §7.2] says

> *Non-overlapping Loads may pass stores*

again with an example essentially identical to this test.

One should be cautious about the "reordered with" phrasing of the Intel documentation: it suggests that relaxed-memory effects can be explained with a mental model in which instructions are each executed atomically but in a reordered instruction stream. That is sometimes true, but it is not true in general, and it's not the most useful way of viewing the observable x86 behaviour.

To fix terminology: we usually speak of load and store instructions, giving rise to read and write memory events, though the literature and test names are not always consistent with this (otherwise this test would be WB, not SB).

2.3 LB: load request buffering?

We turn now to the dual of SB: the "LB" litmus test below, in which Thread 0 reads x and then writes y while Thread 1 reads y and then writes x. The interesting execution is that in which both reads read from the other thread's write, rather than from the initial state, as shown on the right below.

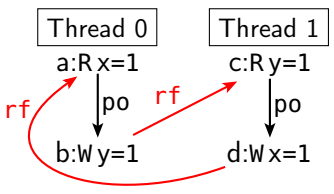
LB x86

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

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

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

`Observation: 0/0`



Experimentally, this is not observed on any x86 processor implementation we have tested.

To see why it is a non-SC execution, one of a and c must do its read first. Suppose without loss of generality that it is a, then as Thread 1 executes in program order (in SC) a is before d, so a must read 0.

Microarchitecturally, the non-SC final state might arise from out-of-order execution, e.g. if the store instructions can be executed out-of-order entirely before the program-order-preceding load instructions, or if the writes can be propagated to memory after the read requests have been created but while they are still buffered (hence the "load buffering" name of the test).

The documented Intel architectural intent for this can be confusing at first sight: the Intel SDM lists a number of "principles" [62, Vol.3A, §8.2.2 *Memory Ordering in P6 and More Recent Processor Families*], including that quoted above, but those do not speak to this case of a read followed by a write to a different address. However, [62, Vol.3A, §8.2.3 *Examples Illustrating the Memory-Ordering Principles*] includes (§8.2.3.3) the statement "The Intel-64 memory-ordering model ensures that a store by a processor may not occur before a previous load by the same processor" and a version of the LB example, unambiguously stating that it is not allowed. The AMD APM states explicitly

> *Stores do not pass loads*

again with an example essentially identical to this test.

We can infer, both from the experimental data and the documented intent, that general out-of-order execution is not observable for x86, and the observed and intended behaviour is so far still consistent with the cartoon microarchitecture in §2.2 above.

2.4 MP: message passing?

Now consider the MP test below. This is a simple form of a very common concurrent programming idiom, in which one thread writes some data (possibly a large data structure) and then writes a flag value indicating that it is ready to be consumed, while another thread reads that flag in a loop until it sees that flag value, before reading the data.

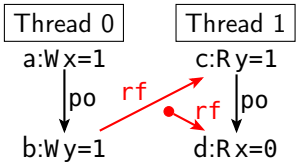
Initial state: x=0; y=0;	
Thread 0	Thread 1
x=1;// *write data* y=1;// *write flag*	**while** (y!=1);// *spin until the flag value is visible* r1=x;// *read data*

To make the litmus-test version as simple as possible, it has just a single read of the flag y, rather than a loop, with the final condition picking out executions in which that read does see the new flag value – as if the loop terminated on its first iteration. The test is also simplified in several other ways compared to what a real software library would have to handle: it doesn't deal with repeated message passing, or with message passing between more than two threads, or with any performance concerns. The interesting execution is that in which the Thread 1 read of y sees the new flag value but its program-order-later read of x sees the "stale" initial-state data rather than that written by Thread 0. For the message-passing software idiom to work reliably, this should be forbidden.

MP x86

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

Observation: 0/100000000



Experimentally, this is again not observed on any x86 processor implementation we have tested – though we'll see later that it is observable on Arm-A and IBM Power processor implementations, so such code needs additional synchronisation on those platforms.

Microarchitecturally, this too could arise from out-of-order execution, either of the two stores on Thread 0 or the two loads on Thread 1, or it could arise even with in-order execution if the two writes can propagate from Thread 0 to memory (or just to Thread 1) out-of-order, e.g. with non-FIFO write buffers, or from some more complex interconnect that has multiple parallel channels.

In the documented vendor architectural intent, it is clearly forbidden. The Intel principles [62, §8.2.2] include:

- *Reads are not reordered with other reads*
- *Writes to memory are not reordered with other writes, with the following exceptions:*
  - *streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and*
  - *string operations (see Section 8.2.4.1).*

and the AMD documentation states

- *Loads do not pass previous loads (loads are not reordered). Stores do not pass previous stores (stores are not reordered)*

which are clear at least about the thread-local behaviour (it's arguably unclear whether this "not reordered with" is supposed to exclude out-of-order write propagation), and [62, Vol.3A, §8.2.3.2 *Neither Loads Nor Stores Are Reordered with Like Operations*] and [17, Vol.2, §7.2] have versions of MP with an unambigous statement that it is forbidden.

x86 processor implementations typically will use out-of-order execution internally, but this experimental data and documented intent again say that that is not observable – hence, whenever such an implementation would result in the forbidden MP final state, it must abort and redo enough of the execution.

The fact that this behaviour is not observed or allowed is what one would expect from a version of the cartoon microarchitecture in §2.2 in which the write buffers are first-in-first-out (FIFO), which previously we left unspecified.

2.5 SB+rfi-pos: write buffers with read-back?

Returning to SB, consider now a variant in which each thread writes to location x or y, reads from that location, and then reads from the other location. The interesting execution is the one in which the first reads see the just-written values while the second reads read from the initial state rather than the value written by the other thread. (We'll explain the name of the test later.)

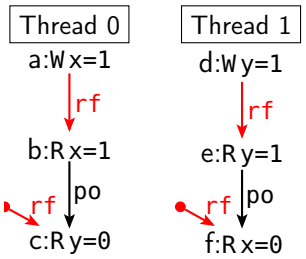
SB+rfi-pos x86

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

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

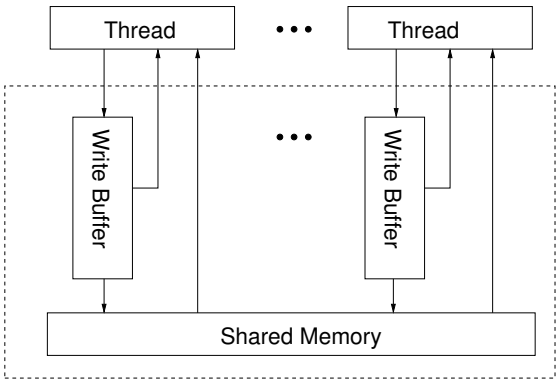
Final: 0:rax=1; 0:rbx=0; 1:rax=1; 1:rbx=0;

Observation: 320/100000000



Experimentally, this is observable, but it is not consistent with the cartoon microarchitecture we have so far, in which writes are buffered and reads read directly from memory. In that, for the first reads to see the just-written values, those writes must have propagated to memory, but then because we believe from the vendor documentation that pairs of reads are not reordered, the Thread 0 reads of `x=1` then `y=0` tell us that the writes `x=1` and `y=1` must reach memory strictly in that order, while the Thread 1 reads of `y=1` then `x=0` tell us that the writes must reach memory in the opposite order. That would be a contradiction, so the second reads couldn't both read from the initial state in the same execution.

This suggests a refined cartoon microarchitecture in which threads can read directly from the write buffer, if it contains a write to the location being read, or directly from memory:



More specifically, each thread should read *the most recent* write in the buffer to the location being read, if there is one, or otherwise read directly from memory, otherwise the behaviour of a single thread in isolation would no longer follow the expected sequential semantics.

In the vendor documentation, the Intel "principles" [62, Vol.3A, §8.2.2] don't speak to this except for a remark that *"The only enhancements in the Pentium 4, Intel Xeon, and P6 family processors are: [...] Store-buffer forwarding, when a read passes a write to the same memory location"*, but [62, Vol.3A, §8.2.3.5 *Intra-Processor Forwarding is Allowed*] says *"The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, each processor may perceive its own store occurring before that of the other."*, while the AMD [17, Vol.2, §7.2] says *"The local visibility (within a processor) for a memory operation may differ from the global visibility (from another processor). Using a data bypass, a local load can read the result of a local store in a store buffer, before the store becomes globally visible. Program order is still maintained when using such bypasses."*; both have a version of this test that is explicitly allowed.

2.6 IRIW: independent reads of independent writes?

We now consider tests with more than two hardware threads, which introduces some interesting questions. First, consider a variant of the above SB+rfi-pos test in which the writes are "pulled out" to separate threads. In this *Independent reads of independent writes* (IRIW) test, Threads 0 and 2 write to `x` and `y` respectively, while Thread 1 reads `x` then `y` and Thread 3 reads `y` then `x`. The interesting execution is that in which the Thread 1 and 3 first reads (`b` and `e`) see the corresponding write, while their second reads (`c` and `f`) do not (they read from the initial state):

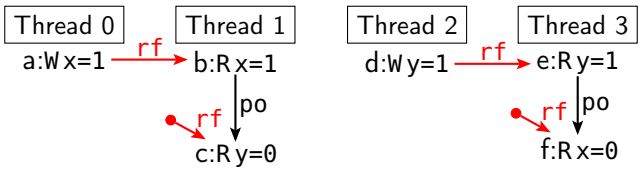
IRIW x86

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

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

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

`Observation: 0/100000000`

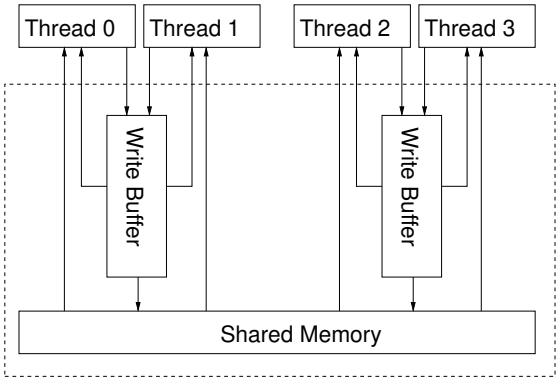


For some time it was debated whether any natural code idioms relied on the absence of IRIW, though it was certainly important when implementing higher-level language models, but in 2020 Osterlund drew attention to an instance in the HotSpot implementation [54].

Experimentally, IRIW is not observed on x86.

Microarchitecturally, would it be allowed in the refined cartoon microarchitecture we just saw in §2.5? No, for essentially the same reason: if the reads are (as far as the programmer can tell) satisfied in order, and (as in the cartoon) they are satisfied from a single memory, then there would be a cycle in the order in which the writes are propagated to that memory, but that is a contradiction.

Is it microarchitecturally plausible? Yes, in various ways. For example, one might have store buffers that are shared between some hardware threads:



In this, Thread 1 could read from Thread 0's write to x, from their shared buffer, before it is propagated to the shared memory that would allow Thread 3 to read it, and symmetrically in the same execution for y. (Of course, one might still have such shared buffers, but tag entries with the hardware thread that created them, and allow reading just from those; then the shared buffers would not be directly observable.) Alternatively, one might have a cache protocol in which invalidates are propagated sufficiently lazily to make this observable even without shared store buffers.

The current x86 vendor documentation unambiguously forbids IRIW, though historically it did not; we'll return to that later. The Intel SDM principles [62, Vol.3A, §8.2.2] include

- *Any two stores are seen in a consistent order by processors other than those performing the stores*

with a version of IRIW [62, Vol.3A, §8.2.3.7 *Stores Are Seen in a Consistent Order by Other Processors*], while the AMD APM [17, Vol.2, §7.2] says:

- *Stores to different locations in memory observed from two (or more) other processors will appear in the same order to all observers.*

also with a version of IRIW, stated to be forbidden.

2.7 WRC: write-to-read causality?

In a similar vein, consider now the following version of MP in which the first write is "pulled out" to another thread. Here Thread 0 writes to x; Thread 1 reads from that write and then

writes y; and Thread 2 reads from that write of y and then reads x. The interesting execution is that in which those first two reads see the writes but the last Thread 2 read of x reads from the initial state, not from the Thread 0 write of x.

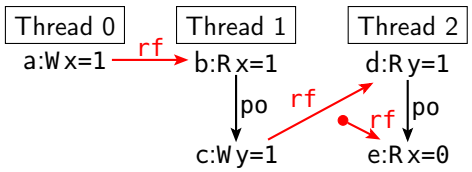
WRC x86

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



This is a simple form of message passing across more than two threads, though with the Thread 0 write of x serving as both the first data and flag writes.

Experimentally, this is not observed on x86.

Microarchitecturally, in the refined cartoon microarchitecture of §2.5 (or in the first one, of §2.2), this would be forbidden: for Thread 1 to read x=1, it must have propagated to memory; the Thread 1 write will be observably in order w.r.t. the Thread 1 read (from the vendor documentation); for Thread 2 to read y=1, that must have propagated to memory; and the Thread 2 reads will be observably in order (from the vendor documentation); so by transitivity (in cartoon microarchitecture execution time) by the time Thread 2 reads x, the Thread 0 write of x=1 must have propagated to memory and so (as Thread 2 can have no writes to x in its buffer) must be read from.

Just like IRIW, however, WRC would be observable in other plausible microarchitectures, including a version of the §2.6 cartoon in which Threads 0 and 1 share a non-FIFO write buffer that is not also shared by Thread 2.

The vendor documentation is clear that this specific test behaviour is forbidden – both Intel and AMD include an essentially identical test – but it less clear what is allowed in general. The Intel principles [62, Vol.3A, §8.2.2] include:

- *Memory ordering obeys causality (memory ordering respects transitive visibility).*

and [62, Vol.3A, §8.2.3.6 *Stores Are Transitively Visible*] says *"The memory-ordering model ensures transitive visibility of stores; stores that are causally related appear to all processors to occur in an order consistent with the causality relation."* while the AMD [17, Vol.2, §7.2] says

- *"Dependent stores between different processors appear to occur in program order".*

However, "transitive visibility", "the causality relation", and "dependent stores" are not precisely defined. The essence of relaxed memory models is that some relations that one might naively think were respected are in fact not, so this is an unfortunate omission.

Architectures that require each write to become visible to other threads (or, in some usages, to all threads) at the same time are known as *multi-copy atomic* (MCA) [57], as discussed in §**??**, while others are *non-multi-copy-atomic* (non-MCA). SC and x86 are MCA, while Arm-A, RISC-V, and IBM Power are non-MCA.

2.8 SB+mfences: restoring order with fences

Any relaxed architecture or programming language needs mechanisms to let the programmer enforce stronger ordering where necessary. The details and names vary widely, but basically there are *fences* or *memory barriers*, which are additional instructions that enforce additional ordering between the instructions before and after them, and variants memory instructions that are *annotated* with various extra strengths. Stronger ordering can also sometimes be determined by page-table attributes or processor modes.

For x86, the simplest is the `mfence` memory barrier instruction. Adding an `mfence` to each thread of SB, between the store and load, gives the SB+mfences test below.

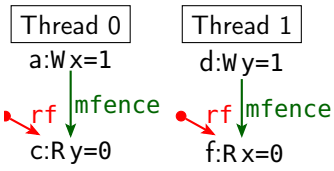
SB+mfences x86

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

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

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

`Observation: 0/100000000`



In the candidate execution, we draw an `mfence` edge between each pair of memory events separated by an `mfence` in program order, and we usually suppress the parallel po edge for clarity.

Experimentally, adding `mfences` makes the non-SC final state of SB unobserved, and the vendor architectural intent is clear that it is forbidden. The Intel principles [62, Vol.3A, §8.2.2] include:

- *Reads cannot pass earlier LFENCE and MFENCE instructions*
- *Writes and executions of CLFLUSH and CLFLUSHOPT cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.*
- *MFENCE instructions cannot pass earlier reads, writes, or executions of CLFLUSH and CLFLUSHOPT.*

and [62, Vol.3A, §8.2.5 *Strengthening or Weakening the Memory-Ordering Model*] adds *"MFENCE – Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream"*. The intent here is reasonably clear, though again the terminology is less so than it might be, with this "cannot pass" in addition to the earlier "are not reordered with", and the microarchitectural "serializes". The AMD documentation states

- *Where sequential consistency is needed (for example in Dekker's algorithm for mutual exclusion), an MFENCE instruction should be used between the store and the subsequent load, or a locked access, such as XCHG, should be used for the store.*

and includes a test essentially identical to SB+fences.

Microarchitecturally, a simple implementation of `mfence` might just pause the execution of program-order-later instructions on its thread until all writes from previous instructions have drained from the thread's write buffer to memory, though one can imagine many more sophisticated implementations.

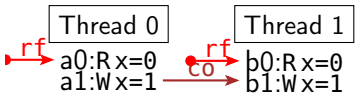
Note that `mfence` does not itself synchronise between threads – the two `mfences` in the test do not synchronise with each other; they just ensure that each threads' write has propagated before the program-order-later read is satisfied.

2.9 Read-modify-write instructions

x86 is not a RISC or load-store architecture, in which the basic instructions are partitioned into instructions that operate on registers and instructions that do a single load or store – in x86 there are many read-modify-write (RMW) instructions that read and write memory, e.g. the increment `INC`. On x86 most of these are by default not *atomic*: they give rise to multiple memory accesses, and other threads can interact with memory between them. For example, the test below just has two parallel increments of `x`:

INC	x86
Initial state: x=0;	
Thread 0	Thread 1
incq (x) //a0,a1	incq (x) //b0,b1
Final: x=1;	

Observation: 1441/1000000



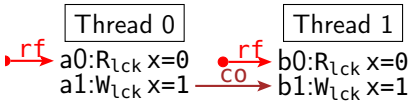
In the candidate execution, we see that each `INC` instruction gives rise to a read and a write event, which we name `a0/a1` and `b0/b1` to indicate that they came from the same instruction. The candidate execution also has a coherence edge `co` between the two writes; we'll return to that later. The interesting final state is that in which both reads read (the same `0` value) from the initial state, then both increment that and write `1`. Note that this is allowed even in an SC semantics, from an interleaving in which the two reads go first, so long as the instruction semantics for `INC` generates distinct read and write events.

Experimentally, this observable on x86, and microarchitecturally, this might happen on almost any implementation.

To force an x86 RMW instruction to be atomic, one can add the `LOCK` prefix (which is literally a one-byte opcode prefix), e.g. as below.

LOCKINC	x86
Initial state: x=0;	
Thread 0	Thread 1
lock incq (x) //a0,a1	lock incq (x) //b0,b1
Final: x=1;	

Observation: 0/1000000



The candidate execution now shows those accesses as LOCK'd.

Experimentally, this makes the specified final state unobserved.

All this is reasonably clear in the vendor documented intent, in [62, Vol.3A, §8.1.1 *Guaranteed Atomic Operations*], [62, Vol.3A, §8.1.2.2 *Software Controlled Bus Locking*], and [17, Vol. 2, §7.3.2 *Access Atomicity*].

The `LOCK` prefix is also supported for various other instructions [62, Vol.3A, §8.1.2.2 *Software Controlled Bus Locking*]: bit test and modify `BTS`, `BTR`, and `BTC`; exchange instructions `XADD`, `CMPXCHG`, and `CMPXCHG8B` (it is implicit for `XCHG`); and arithmetic and logical instructions `INC`, `DEC`, `NOT`, `NEG`, `ADD`, `ADC`, `SUB`, `SBB`, `AND`, `OR`, and `XOR`.

2.10 Synchronising power of locked instructions

These LOCK'd instructions have additional synchronisation force. For brevity, we'll omit the tests and experimental data, and just recall the vendor intent. The Intel principles [62, §8.2.2] include:

- *Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.*
- *In a multiple-processor system, the following ordering principles apply: [...] Locked instructions have a total order.*

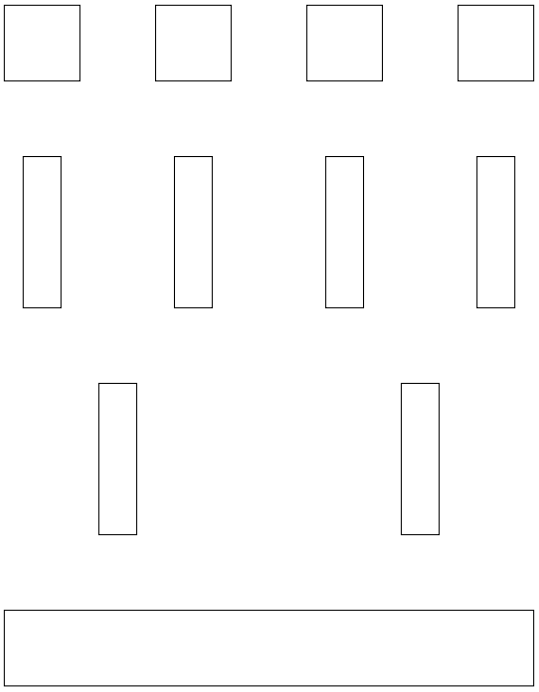
and has examples in [62, Vol.3A, §8.2.3.9 *Loads and Stores Are Not Reordered with Locked Instructions*] and [62, Vol.3A, §8.2.3.8 *Locked instructions have a total order*] while the AMD APM [17, Vol 2., §7.1.3] says:

- *Serializing instructions, I/O instructions, and locked instructions (including the implicitly locked XCHG instruction) can also be used as read/write barriers*

2.11 Exercises

Exercise 2.1 *Show that Peterson's and Dekker's algorithm are not correct under x86-TSO, and show how to fix them using mfence*

Exercise 2.2 *Consider a "hierarchical" version of TSO, in which a thread can be replaced by a TSO machine.*



Which executions forbidden by TSO are allowed by this model?
See Flowing [70].

Exercise 2.3 *Consider a "mixed" version of TSO (generalising the figure in section 2.6), where one thread can share a buffer with another thread for some locations, but share a different buffer with a different thread for some other locations. Which executions forbidden by TSO are allowed by this model? How does this model compare to the one above (use litmus tests)?*

Chapter 3

x86: some vendor documentation history

In the previous chapter we saw some excerpts of the Intel and AMD specifications of their concurrency behaviour, expressed in a combination of informal prose and examples. In some cases this prose was hard to interpret, and this is a common and intrinsic problem: informal prose is a poor medium for loose specification of subtle properties. It's very hard to make prose specifications unambiguous; they are not machine-readable or executable, so one cannot compute the behaviour of test programs that a prose specification allows; and one cannot cannot use them directly as criteria for testing processor implementations.

In the next chapters we show how one can do better, with mathematically precise models for x86 concurrency. To give more context for that, we first recall some of the history of the x86 documented models, as we saw them. The reader who just wants to know what x86 is now can safely skip this.

3.1 pre-IWP (before Aug. 2007)

Early revisions of the Intel SDM (e.g. rev. 22, Nov. 2006) gave an informal-prose model called 'processor ordering', unsupported by any examples. It is hard to see precisely what this prose means, especially without additional knowledge or assumptions about the microarchitecture of particular implementations.

The uncertainty about x86 behaviour that at least some systems programmers had about earlier IA-32 processors can be gauged from an extensive Linux kernel mailing list discussion in 1999 about the correctness of a proposed optimisation to a Linux spinlock implementation [5]. In brief, the `spin_unlock` code initially used a bit-test-and-reset instruction with the LOCK prefix. Manfred Spraul thought this could be improved from around 22 ticks for the "`lock; btrl $0,%0`", to 1 tick for a simple "`movl $0,%0`" instruction, a huge gain. Linus Torvalds asserted that this should not work in general, because of potential CPU reordering of instructions around the `spin_unlock` code. There followed a lengthy discussion of the behaviour of different x86 generations, eventually resolved only by Erich Boleyn, an Architect in an IA32 development group at Intel, who wrote:

> *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.*

> *Since the instructions for the store in the `spin_unlock` have to have been externally observed for `spin_lock` to be aquired (presuming a correctly functioning spinlock, of course), then the earlier instructions to set "b" to the value of "a" have to have completed first. In general, IA32 is Processor Ordered for cacheable accesses. Speculation doesn't affect this. Also, stores are not observed speculatively on other processors.*

It's notable that this is largely in microarchitectural terms, not just in terms of the specified architecture. We return to this optimisation later, where we can explain why it is sound with respect to our models.

3.2 IWP/AMD3.14/x86-CC

In August 2007, an Intel White Paper [9] (IWP) gave a somewhat more precise model, with 8 informal-prose principles P1–P8 supported by 10 litmus-test examples. This was incorporated, essentially unchanged, into later revisions of the Intel SDM (including rev. 26–28), and AMD gave similar, though not identical, prose and tests in rev. 3.14 of their manual [8, Vol. 2,§7.2] (AMD3.14).

These two were a substantial improvement in clarity but still suffered from several issues.

First, AMD3.14 explicitly allowed the §2.6 IRIW example, in which different hardware threads can see writes to independent locations in different orders, while IWP allowed it implicitly, as IRIW is not ruled out by the stated principles[1]. IRIW was not observable in practice in our experimental testing, but a lack of observation is never definitive: testing can never be exhaustive; it covered only some of many x86 processor implementations; and it says nothing about possible planned future x86 implementations. Allowing IRIW gave a rather weak model for programmers. In particular, under reasonable assumptions on the strongest x86 memory barrier, `mfence`, adding `mfences` between pairs of instructions would not suffice to recover sequential consistency, and that is challenging for implementations of high-level languages that aim to provide stronger models. It appeared that some JVM implementations depended on `mfence` sufficing for that, and would not be correct if one assumed only the IWP/AMD3.14 architecture [66]. Instead, one would have to make liberal use of x86 LOCK'd instructions [114, §2.12], with a big performance cost.

Both IWP and AMD3.14 required that causality is respected, in some sense, as in the IWP principle P5 (still in the current documentation [62, Vol.3A, §8.2.2]):

> *P5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility)*

We used these informal specifications as the basis for a formal model, x86-CC [114], for which a key issue was giving a reasonable interpretation to this "causality", which is not defined in IWP or AMD3.14. We interpreted causality as the union of a preserved program order within each thread (capturing e.g. the *"P1 loads are not reordered with other loads"*), a per-location total coherence order over writes (capturing the *"P6 In a multiprocessor system, stores to the same location have a total order"*), a total order over LOCK'd instructions (capturing the *"P7. In a*

[1] This reasoning was later questioned by Intel staff, who wrote [130]:

> *We do not believe that there was any statement in the white paper that allows IRIW or that could be inferred to do so (i.e., there was no statement that the examples given include all limitations on the model). The IRIW example was discussed [...] within Intel before the white paper was published. While there was consensus that it was not allowed by the model, there was no consensus to publish that fact. A decision was made to omit any mention of IRIW in the white paper and to craft the writing so that no inference about IRIW could be drawn.*

However, given such a specification, any reasonable programmer would surely be unable to assume that IRIW cannot occur.

multiprocessor system, locked instructions have a total order"), and the reads-from relation (capturing P5). Roughly speaking, we were interpreting all uses of "order" in the prose principles as referring to the same causality order, which is arguably natural, though naive in hindsight. The model had a linear view order for each hardware thread, over all its events and the memory write events of other threads, and we required that each of those is consistent with the global causality relation. This gave the correct behaviour for the vendor litmus tests, and was consistent with some limited experimental testing for those; it was also provably equivalent to an operational abstract machine model.

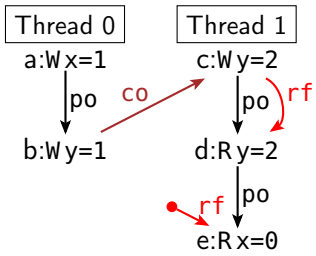
However, as noted by Paul Loewenstein [114, App.A], the model is not sound with respect to observable x86 implementation behaviour, e.g. for the test below:

R+po+rfi-po (n6) x86

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

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

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



This is allowed by implementations with FIFO store buffers, as in the §2.5 cartoon microarchitecture, but not by x86-CC, or by any interpretation we could make of IWP principles P1,2,4 and 6 [103, A.5].

3.3 Intel SDM rev. 29–34 (Nov. 2008–Mar. 2010)

A later substantial change to the Intel memory model specification was in revision 29 of the Intel SDM (revisions 29–34 were essentially identical except for the LFENCE text). This was in a similar informal-prose style to previous versions, but significantly different to IWP. First, IRIW is forbidden [11, Example 8-7, vol. 3A], and the previous coherence condition:

> *P6. In a multiprocessor system, stores to the same location have a total order*

was been replaced by:

> *Any two stores are seen in a consistent order by processors other than those performing the stores*

Second, the memory barrier instructions are now included. It was stated that reads and writes cannot pass `mfence` instructions, together with more refined properties for `sfence` and `lfence`.

Third, same-processor writes were explicitly ordered:

> *P10. Writes by a single processor are observed in the same order by all processors*

(we previously regarded this as implicit in the IWP *"P2. Stores are not reordered with other stores"*).

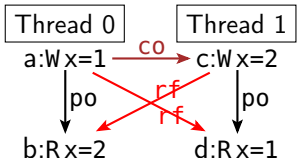
How to interpret "causality" in P5 remained unclear, and the new P9 says nothing about observations of two stores by those two processors themselves (or by one of those processors and one other). The following examples illustrate potentially surprising behaviour that arguably violates coherence. Their final states are not allowed in x86-CC, are not allowed in a pure store-buffer implementation or in x86-TSO, and we have not observed them on actual processors. However, the principles stated in revisions 29–34 of the Intel SDM appear, presumably unintentionally, to allow them. The AMD3.14 Vol. 2, §7.2 text taken alone would allow them, but the implied coherence from elsewhere in the AMD manual would forbid them.

SB+poss (n5) x86

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

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

Final: 0:rax=2; 1:rax=1; x=2;

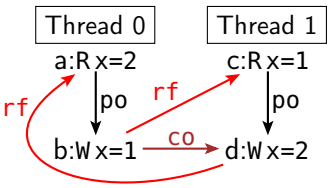


LB+poss (n4b) x86

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

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

Final: 0:rax=2; 1:rax=1; x=2;



All this illustrates once again the difficulty of writing unambiguous and correct loose specifications in informal prose.

3.4 AMD APM version 3.15 (Nov. 2009)

In November 2009, AMD produced a new revision, 3.15, of their manuals. The main difference in the memory model specification was that IRIW was now explicitly forbidden.

3.5 Intel SDM rev.80 (June 2023)

The Intel SDM at the time of writing (revision 80, from June 2023) [62, Vol.3A] describes its memory ordering model as below.

> ***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.*
>
> *In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (Note the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term "processor" refers to a logical processor. For example, a physical processor supporting multiple cores and/or Intel Hyper-Threading Technology is treated as a multi-processor systems.):*
>
> - *Reads are not reordered with other reads.*
> - *Writes are not reordered with older reads.*
> - *Writes to memory are not reordered with other writes, with the following exceptions:*
>   - *streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and*
>   - *string operations (see Section 8.2.4.1).*
> - *No write to memory may be reordered with an execution of the CLFLUSH instruction; a write may be reordered with an execution of the CLFLUSHOPT instruction that flushes a cache line other than the one being written. Executions of the CLFLUSH instruction are not reordered with each other. Executions of*

CLFLUSHOPT that access different cache lines may be reordered with each other. An execution of CLFLUSHOPT may be reordered with an execution of CLFLUSH that accesses a different cache line.

- *Reads may be reordered with older writes to different locations but not with older writes to the same location.*
- *Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.*
- *Reads cannot pass earlier LFENCE and MFENCE instructions.*
- *Writes and executions of CLFLUSH and CLFLUSHOPT cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.*
- *LFENCE instructions cannot pass earlier reads.*
- *SFENCE instructions cannot pass earlier writes or executions of CLFLUSH and CLFLUSHOPT.*
- *MFENCE instructions cannot pass earlier reads, writes, or executions of CLFLUSH and CLFLUSHOPT.*

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

- *Individual processors use the same ordering principles as in a single-processor system.*
- *Writes by a single processor are observed in the same order by all processors.*
- *Writes from an individual processor are NOT ordered with respect to the writes from other processors.*
- *Memory ordering obeys causality (memory ordering respects transitive visibility).*
- *Any two stores are seen in a consistent order by processors other than those performing the stores*
- *Locked instructions have a total order.*

[...]

The processor-ordering model described in this section is virtually identical to that used by the Pentium and Intel486 processors. The only enhancements in the Pentium 4, Intel Xeon, and P6 family processors are:

- *Added support for speculative reads, while still adhering to the ordering principles above.*
- *Store-buffer forwarding, when a read passes a write to the same memory location.*
- *Out of order store from long string store and string move operations (see Section 8.2.4, "Fast-String Operation and Out-of-Order Stores," below).*

together with **8.2.3 Examples Illustrating the Memory-Ordering Principles**.

It still contains much of the previous text and a priori is not obvious how to interpret as a precise model, especially for the mixed "not reordered with", "cannot pass", "not ordered with respect to", "obeys causality", and "have a total order" terminology.

The introductory *that can be further defined as "write ordered with store-buffer forwarding"* does add welcome clarification, however.

3.6 AMD APM 4.07 (April 2020)

The AMD APM at the time of writing (version 4.07 from April 2020) describes their memory ordering model as below (abbreviating the examples that we have already seen) [17, Vol.2].

> ***§7.2 Multiprocessor Memory Access Ordering***
>
> *To improve performance of applications, AMD64 processors can speculatively execute instructions out of program order and temporarily hold out-of-order results. However, certain rules are followed with regard to normal cacheable accesses on naturally aligned boundaries to WB memory. In the examples below, all memory values are initialized to zero.*
>
> *From the point of view of a program, in ascending order of priority:*
>
> - *All loads, stores and I/O operations from a single processor appear to occur in program order to the code running on that processor and all instructions appear to execute in program order.*
> - *Successive stores from a single processor are committed to system memory and visible to other processors in program order. A store by a processor cannot be committed to memory before a read appearing earlier in the program has captured its targeted data from memory. In other words, stores from a processor cannot be reordered to occur prior to a load preceding it in program order. In this context:*
>   - *Loads do not pass previous loads (loads are not reordered). Stores do not pass previous stores (stores are not reordered) [MP]*
>   - *Stores do not pass loads [LB]*
> - *Stores from a processor appear to be committed to the memory system in program order; however, stores can be delayed arbitrarily by store buffering while the processor continues operation. Therefore, stores from a processor may not appear to be sequentially consistent.*
> - *Non-overlapping Loads may pass stores. [SB] Where sequential consistency is needed (for example in Dekker's algorithm for mutual exclusion), an MFENCE instruction should be used between the store and the subsequent load, or a locked access, such as XCHG, should be used for the store. [SB+mfences]*
> - *Loads that partially overlap prior stores may return the modified part of the load operand from the store buffer, combining globally visible bytes with bytes that are only locally visible. To ensure that such loads return only a globally visible value, an MFENCE or locked access must be used between the store and the dependent load, or the store or load must be performed with a locked operation such as XCHG.*
> - *Stores to different locations in memory observed from two (or more) other processors will appear in the same order to all observers. [IRIW]*
> - *Dependent stores between different processors appear to occur in program order, as shown in the code example below. [WRC]*
> - *The local visibility (within a processor) for a memory operation may differ from the global visibility (from another processor). Using a data bypass, a local load can read the result of a local store in a store buffer, before the store becomes globally visible. Program order is still maintained when using such bypasses. [SB+rfi-pos]*
>
>   *There are no constraints on the relative order of when the Store A of processor 0 is visible to processor 1 relative to when the Store B of processor 1 is visible to processor 0.*

If a very strong memory ordering model is required that does not allow local store-load bypasses, an MFENCE instruction or a synchronizing instruction such as XCHG or a locked Read-modify- write should be used between the store and the subsequent load. This enforces a memory ordering stronger than total store ordering. [SB+rfi-mfence-po]

Chapter 4

x86-TSO: creating a good de facto standard model

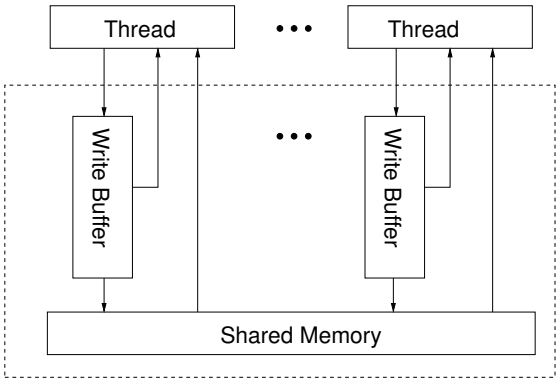
Given the lack of clarity in the vendor prose specifications of 2007–2009, as described in the previous two chapters, we concluded that one could not produce a useful rigorous model, to clarify the allowed behaviour for programmers and to support verification, solely by formalising the "principles" from the prose specifications (as we attempted with x86-CC [114]). Instead, we had to build a reasonable model that was consistent with the given litmus tests, with observed processor behaviour, and with what we knew of the needs of programmers, the vendors' intentions, and the folklore in the area.

To summarise some key facts from the previous two chapters:

- Store buffering (with forwarding) is observable, and explicitly allowed by the prose specifications
- 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, and other pairwise reorderings are explicitly forbidden by the prose
- IRIW and WRC not observable, and (since the Intel SDM rev.29 in Nov. 2008 and AMD APM 3.15 in Nov. 2009) are forbidden by the prose
- `mfence` appears to wait for the local store buffer to drain
- LOCK'd instructions also appear to wait for the local store buffer to drain, before they execute

These strongly suggested that, apart from store buffering, all processors share the same view of memory. Moreover, different processors or hardware threads do not observably share store buffers. To the best of our knowledge, for the usual write-back memory (and excluding non-temporal and string instructions), no other aspects of the microarchitecture (the out-of-order execution, cache hierarchies and protocols, interconnect topology, and so on) are observable to the programmer, except in so far as they affect performance. The obvious conclusion was that x86 is, in practice, much like the SPARC Total Store Ordering (TSO) memory model [126, 2]: the observable effects of store buffers are the only observable relaxed-memory behaviour.

We therefore designed a TSO-like model for x86, x86-TSO by Owens, Sarkar, Sewell, Myreen, and Zappa Nardelli [101, 103, 120]. This is extensionally very similar to the SPARC TSO model, essentially codifying the cartoon microarchitecture of §2.5:



except that it covers the x86 LOCK'd instructions.

Chapter 5

Operational and axiomatic concurrency model definitions

To define a relaxed memory model precisely, there are two main styles of definition one can use, *operational* and *axiomatic*:

- An operational model is expressed as an abstract machine, with a set of possible state, transitions between them, and an initial state.
- An axiomatic model is expressed as a predicate (the "axioms" of the model) on some notion of candidate complete execution graph, defining the set of candidate executions that are allowed.

There are many pros and cons of both styles, which have been much discussed in the literature (which we return to in Part **??**).

Operational models are often, though not necessarily, designed to capture some microarchitectural intuition of how real hardware implementations work, though still abstracting from as many of their details as they can while still defining the intended envelope of behaviour; we call these *abstract microarchitectural* operational models. This can be useful to *explain* relaxed behaviour, if one can arrange for behaviours to arise in the model for essentially the same reasons that they do in hardware implementations, and it can guide model design and testing. The fact that one has to define the model behaviour in any model state helps identify previously unconsidered subtleties. Of course, an actual hardware implementation that extensionally conforms to such a model could be less aggressive than the model, or it could be more aggressive internally, as many are, so long as it does not extensionally allow more programmer-visible behaviours.

Axiomatic models are typically, though again not necessarily, designed to define the allowed complete executions as concisely as possible. Many do so in terms of a notion of candidate execution that has just a single event for each memory read and write, with a single reads-from relation, and other relations, over them – exactly as we have seen in the example execution graphs. Some axiomatic models are more intricate and more microarchitectural, e.g. with events recording when each write propagates to each other thread, and/or orders for each hardware thread, but we do not discuss any models of that kind here. Axiomatic models in this relaxed-concurrency sense should not be confused with the (now largely historical) "axiomatic semantics" approach to programming language definition, originating with Floyd [68] and Hoare [80], in which one defines the meaning of language constructs by giving program-logic proof rules for them.

The main advantage of axiomatic models is usually their brevity: the key part can often be expressed as a page or so of definitions, whereas operational models tend to have a relatively complex state, and execution of a single instruction may involve many transitions. Axiomatic models are also typically mathematically simpler, expressed just in terms of various relation def-

initions (albeit sometimes inductive), but at the cost of the clearer operational and microarchitectural intuition of operational models. That said, one potential concern with abstract microarchitectural models is that they could be mistakenly taken by hardware designers as prescribing the intensional internal structure of implementations, rather than just the intended extensional envelope of programmer-visible behaviour, and some operational model designs involve choices that do not affect the set of programmer-visible behaviour. Moreover, the architectural intent does not necessarily have a natural operational characterisation – e.g., hypothetically, if it is intended to cover two quite different microarchitectural implementation techniques for which there is not a good operational characterisation of the union of their behaviours.

Operational models support incremental construction of allowed executions. That means one can build tools for interactive or single-random-trace exploration of the allowed behaviour, and to compute all allowed behaviours one can compute an exhaustive exploration of the transition system. That has proved feasible for litmus tests, with some optimisations, but exhaustive exploration quickly becomes prohibitive for larger examples. Axiomatic models do not directly support incremental construction of partial executions; they only characterise the allowed complete executions. For small litmus-test examples, one can calculate the set of all allowed outcomes, or whether some final state is allowed, using constraint-solving approaches (which historically have been faster than the operational tools, though scaling is still a concern), but interactive or single-random-trace exploration is harder.

The fact that operational models construct candidate executions incrementally means that they can execute the intra-instruction semantics of each instruction instance on concrete values, without requiring symbolic execution of that semantics (sometimes modulo calculation of some register and memory footprint information), which axiomatic tools often do.

Each style has advantages for certain kinds of proof: operational models supports induction on model-execution traces, while axiomatic models directly give one explicit properties of complete executions.

The ideal is to have models expressed in both styles, with a proof of equivalence between them, or, failing that, with experimental testing of their equivalence on a substantial body of litmus tests. In the following chapters, we'll describe such pairs of models, with equivalence proofs, for SC, x86-TSO, and a user fragment of Arm-A; in other cases, we'll just give an operational model (for larger fragments of Arm-A and for IBM Power), or a pair of models with testing rather than proof of their equivalence (for RISC-V).

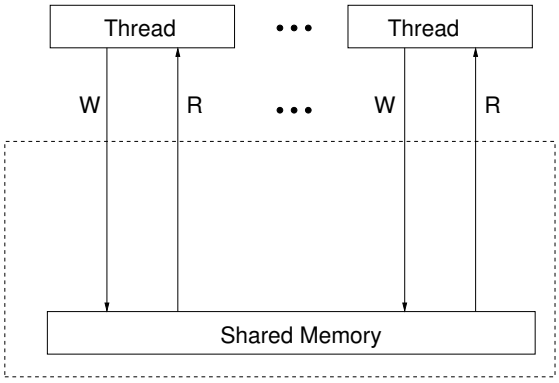
Chapter 6

SC, operationally

To express relaxed memory models precisely, we use standard simple discrete mathematics. Fig. 6.1 recalls the usual basic terminology and notation, for sets, tuples, records, lists, functions, relations, and formulas, and we'll introduce more as needed. Previous familiarity with this shouldn't be necessary to read the text.

Before describing x86-TSO, we first give a precise operational model for SC, to introduce the notation we'll be using.

For now, this will model just the memory of an SC machine – the dashed box of the cartoon microarchitecture below – leaving the hardware thread behaviour informal. We'll return later to how the thread behaviour can also be made precise. For SC and x86-TSO, we will later codify that each hardware thread executes in order, generating writes and reads in the obvious way (and `mfences` and RMWs for x86), but here we are only defining the memory behaviour in isolation.



6.1 An operational SC model

Interface The *interface* between the hardware threads and the memory is codified with write and read *events*:

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

These are defined with a grammar: the above just says that event e is either a write event, of the form $a{:}t{:}\mathrm{W}\ x{=}v$, or a read event, of the form $a{:}t{:}\mathrm{R}\ x{=}v$. We write *events* for the set of all these. Here:

- a is a unique event ID, from a set *eid*
- t is a hardware thread ID, from a set *tid*

Sets are unordered collections of distinct *elements*. $\{a_1, \ldots a_n\}$ is the finite set with elements $a_1, \ldots, a_n$, and $\{\}$ is the empty set, with no elements. $A \cup B$ is the union of A and B, containing all the elements of either. $A \cap B$ is the intersection of A and B, containing the elements that are in both. $A \times B$ is the product of A and B, containing all pairs (a, b) where a is an element of A and b is an element of B. $A \to B$ is the set of all (total) mathematical functions from A to B. For a variable x and formula P, $\{x \mid P\}$ is the set of all a for which P with x instantiated to a is true.

Tuples are fixed-length ordered sequences of elements. $(a_1 \ldots a_n)$ is an n-tuple of $a_1, \ldots, a_n$ (where $n \geq 2$), and (a_1, a_2) is the pair of a_1 and a_2. Parentheses $(\cdot)$ are also used for grouping expressions and formulas, as usual.

Records are like tuples except with named fields. $\langle \mathit{FieldName}_1 := a_1; \ldots; \mathit{FieldName}_n := a_n \rangle$ is the record with those fields and values. $r.\mathit{FieldName}$ is the value of the *FieldName* field of record r. $r \oplus \langle \mathit{FieldName} := a \rangle$ is the record obtained by updating the *FieldName* field of r to a.

Lists are arbitrary-length ordered sequences of elements. $[a_1, \ldots, a_n]$ is the list of elements $a_1, \ldots, a_n$ (where $n \geq 0$), so [] is the empty list and $[a]$ is the singleton list with a single element a. $l_1 \mathbin{@} l_2$ is the concatentation of lists l_1 and l_2.

Functions f from a set A to a set B map each element of A to some element $f\,a$ of B. These are pure total mathematical functions; they have no side-effects. $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ is the function that maps each of those a_i (which must be distinct, and be all the elements of A) to the corresponding b_i (which need not be distinct or include all the elements of B). $f\,a$ is the result of applying function f to argument a. $f \oplus (a \mapsto b)$ is the mathematical function that is like f except that it maps a to b

A *binary relation* $\to$ over a set A is a set of pairs (a, a') of elements of A, or in other words a subset $\to \subseteq A \times A$.

Formulas (or *Predicates*) P express properties that might be true or false.

$a \in A$ is true	iff a is an element of A
$A \subseteq B$ is true	iff A is a subset of B, i.e., if all elements of A are elements of B
$\neg P$ is true	iff P is false
$P \wedge P'$ is true	iff P and P' are both true
$P \vee P'$ is true	iff P is true or P' is true
$P \implies P'$ is true	iff P implies P', i.e., if P is true then P' is true (if P is false, then $P \implies P'$ is vacuously true irrespective of P')
$\forall x \in A.\ P$ is true	iff for all elements $a \in A$, P with x instantiated to a is true (if A is empty, this is vacuously true)
$\exists x \in A.\ P$ is true	iff there exists some $a \in A$, such that P with x instantiated to a is true

Normally one would drop all of those "is true"s, of course, but they may be helpful on a first reading. If the ranges of the quantifiers are clear from the context, they can be omitted, writing just $\forall x.\ P$ and $\exists x.\ P$. One often uses infix notation for binary relations, e.g. for a relation $\to$, writing $a \to a'$ for $(a, a') \in \to$.

Figure 6.1: Discrete maths notation and terminology: sets, tuples, records, lists, functions, relations, and formulas

- x is a memory address, from a set *addr*, and
- v is a memory value, from a set *value*

These are exactly like the events we've seen in candidate execution diagrams except that their thread ID is explicit, rather than implicit in the diagrams. Note that these are atomic: read events include both the address and the result value of the read; they are not split into separate read-request and read-result events (later we will sometimes use finer-grain events, but for SC and x86-TSO it is not necessary). Exactly what the sets of memory addresses and values are is not important now, as we are deferring mixed-size accesses and virtual memory to later. One could imagine, for example, a set of byte addresses and byte values, or a set of 8-byte (64-bit) aligned addresses and 8-byte values.

We use the following auxiliary functions to deconstruct events:

- $\mathrm{id}(e)$, $\mathrm{thread}(e)$, $\mathrm{addr}(e)$, $\mathrm{value}(e)$ extract the respective components of event e, and
- $\mathrm{isread}(e)$, $\mathrm{iswrite}(e)$, $\mathrm{isdequeue}(e)$, $\mathrm{ismfence}(e)$ identify the corresponding kinds of event.

States The *states* m of the operational SC model are complete memory states, giving a value for each address.

$$m : \mathit{addr} \to \mathit{value}$$

This just states that m has the type $\mathit{addr} \to \mathit{value}$, of all (mathematical) functions from *addr* to *value*.

Behaviour The *behaviour* of the model is expressed as a transition system $m \xrightarrow{e} m'$, meaning that state m can participate in event e and transition to state m'; here e is the *label* of the transition. Formally, $_ \xrightarrow{} _$ is a ternary relation, a subset of the set of triples (m, e, m') of a memory, label, and memory $(\mathit{addr} \to \mathit{value}) \times \mathit{events} \times (\mathit{addr} \to \mathit{value})$. It's defined by two *rules*:

WM: Write to memory

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

RM: Read from memory

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

The WM write rule says that any memory state m can participate in a write transition with label $a{:}t{:}\mathsf{W}\,x{=}v$, with the new state $m \oplus (x \mapsto v)$ being m updated with address x holding value v. In other words, any thread t can always write any value v to memory at any address x.

The RM read rule says that for any memory state m, if m contains v at address x (the precondition above the line), can participate in a read transition with label $a{:}t{:}\mathsf{R}\,x{=}v$, with the new state being m unchanged. In other words, any thread t can read v from memory at address x if the memory does contain v at x.

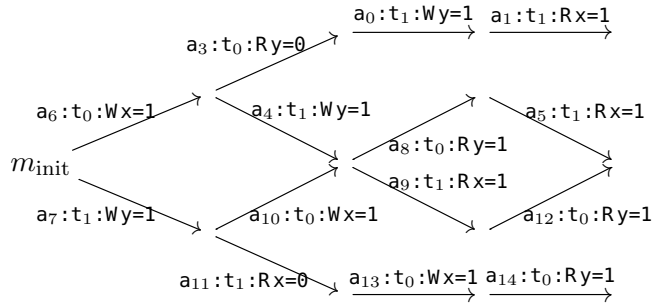
Note that neither rule constrains the event ID or, more interestingly, the thread ID: the SC memory doesn't care which thread is doing a write or read.

Initial State We take the *initial state* of the model to be m_{init}, the memory m in which every address is mapped to value 0.

Traces are just sequences of events $e_1, \ldots, e_n$ with distinct event IDs, and the *traces of the model* are traces for which there are states $m_1, \ldots, m_n$ and transitions starting with the initial state:

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

This includes the traces through the graph we saw in Chapter 1:



though because this is just the semantics of memory, independent of the threads, it also allows many others, e.g.

$$m_{\text{init}} \xrightarrow{a_1:t_0:Wx=42} \xrightarrow{a_2:t_0:Rx=42}$$

in which a thread writes 42 to x then reads it back. The model does not allow traces like:

$$m_{\text{init}} \xrightarrow{a_1:t_0:Wx=42} \xrightarrow{a_2:t_0:Rx=43}$$

in which a thread writes 42 to x then (without any other intervening write) reads a distinct value back.

Note how the model captures the essence of SC: reads read from the most recent write to the same address, in some interleaving of whatever the threads do.

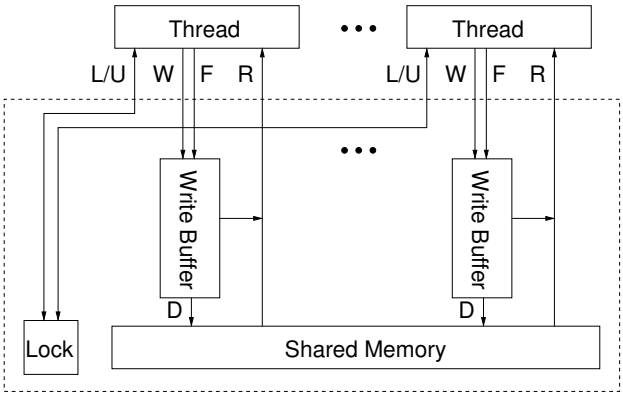
6.2 Exercises

Exercise 6.1 *Write the SC transition system for the litmus tests of Chapter 2.*

Chapter 7

x86-TSO, operationally

The x86-TSO operational model essentially formalises the cartoon microarchitecture of §2.5, with per-thread FIFO write buffers and forwarding paths, and the addition of a lock to the abstract machine state to capture the atomicity of x86 LOCK'd RMW instructions:



As for SC, for now we will formalise just the x86-TSO memory behaviour – the dashed box of the cartoon microarchitecture above – leaving the hardware thread behaviour informal. The hardware threads each execute in order, generating writes, reads, `mfences`, and RMWs in the obvous way.

7.1 An operational x86-TSO model

Interface The interface between the hardware threads and the memory includes write and read events as in SC, together with new events $a{:}t{:}\mathsf{F}$ for a `mfence` memory barrier, $a{:}t{:}\mathsf{L}$ for the start of a LOCK'd instruction, and $a{:}t{:}\mathsf{U}$ for the end of a LOCK'd instruction. The latter two let us define the instruction semantics of LOCK'd instructions in terms of the semantics of their un-LOCK'd counterparts, by wrapping them in a $a{:}t{:}\mathsf{L}/a{:}t{:}\mathsf{U}$ pair. The events also include *dequeue* events $a{:}t{:}\mathsf{D}_w\ x{=}v$, where w is some write event $a'{:}t{:}\mathsf{W}\ x{=}v$. These are not part of the thread/memory interface, but they are useful in reasoning; they instrument the point at which the abstract machine dequeues a write from its write buffer to the shared memory.

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

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

$$m : \langle M : \mathit{addr} \to \mathit{value};\\ B : \mathit{tid} \to \mathit{write_event\ list};\\ L : \mathit{tid\ option}\ \rangle$$

Here:

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

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

Behaviour To define the behaviour, two auxiliary functions are useful. 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 *blocked* in machine state m if some other thread holds the lock ($m.L = \text{SOME}\, t'$ for some $t' \neq t$) and *not blocked* otherwise, i.e., if either it holds the lock ($m.L = \text{SOME}\, t$) or the lock is not held ($m.L = \text{NONE}$).

The behaviour of the model is again expressed as a transition system $m \xrightarrow{e} m'$, defined by the following seven rules. For each, we give a prose transcription of the rule underneath.

RM: Read from memory

$$\frac{\text{not_blocked}(m, t)\quad m.M(x) = v\quad \text{no_pending}(m.B(t), x)}{m \xrightarrow{a{:}t{:}\mathsf{R}\,x=v} m}$$

Thread t can read v from memory at address x if t is not blocked, the memory does contain v at x, and there are no writes to x in t's store buffer. (Note that the event ID a is left unconstrained by the rule.)

RB: Read from write buffer

$$\frac{\text{not_blocked}(m, t)\quad \exists a'\ b_1\ b_2.\ m.B(t) = b_1 \mathbin{@} [a'{:}t{:}\mathsf{W}\,x{=}v] \mathbin{@} b_2\quad \text{no_pending}(b_1, x)}{m \xrightarrow{a{:}t{:}\mathsf{R}\,x=v} m}$$

Thread t can read v from its store buffer for address x if t is not blocked and has v as the value of the most recent write to x in its buffer. We express the latter mathematically by saying that there exists some decomposition of the buffer $m.B(t)$ into the concatentation of three lists: b_1, a singleton list $[a'{:}t{:}\mathsf{W}\,x{=}v]$, and b_2, where b_1 has no writes to x.

WB: Write to write buffer

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

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

DM: Dequeue write from write buffer to memory

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

If Thread t is not blocked, it can silently dequeue the oldest write from its store buffer and update memory at that address with the new value, without coordinating with any hardware thread. (we record the write in the dequeue event just to simplify proofs.)

M: MFENCE

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

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

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

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

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

L: Lock

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

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

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

U: Unlock

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

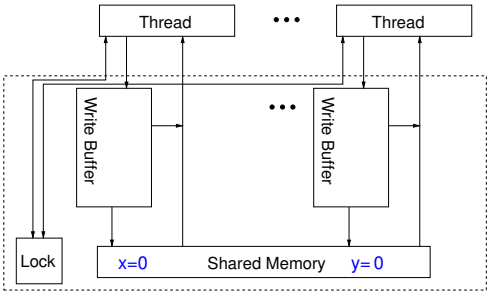
If t holds the lock, and its store buffer is empty, it can end a LOCK'd instruction, resetting the lock.

7.2 x86-TSO operational example: SB

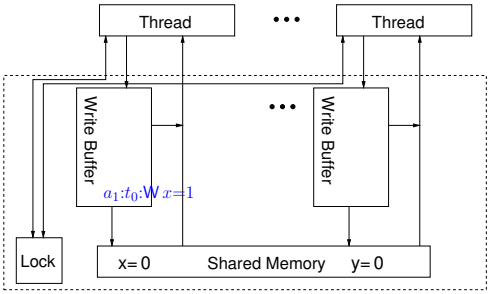
Now that we finally have a mathematically rigorous model, we can go back to the first Chapter 1 SB example and see precisely why the model allows it. We continue to gloss over the instruction semantics – we just assume that store and load instructions generate single write and read events and have the obvious effects on registers, which is fine for simple tests like this one.

As in Chapter 6, a final state is allowed in the model if there is some trace of transitions from the initial state. We give such a trace in detail below, illustrating the different memory model states it goes through. For this example, we assume $addr = \{x, y\}$ and $tid = \{t_0, t_1\}$.

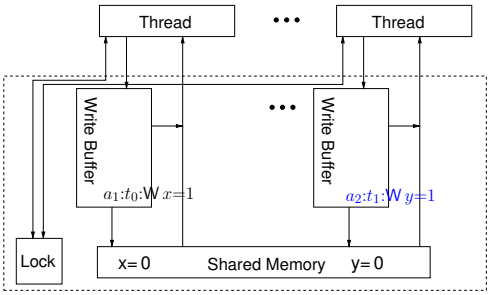
$$m_{\text{init}} = \begin{array}{l}\langle M := \{x \mapsto 0,\ y \mapsto 0\};\\ \ B := \{t_0 \mapsto [],\ t_1 \mapsto []\};\\ \ L := \text{NONE}\ \rangle\end{array}$$



$$\xrightarrow{a_1{:}t_0{:}\text{W}\,x{=}1} \begin{array}{l}\langle M := \{x \mapsto 0,\ y \mapsto 0\};\\ \ B := \{t_0 \mapsto [a_1{:}t_0{:}\text{W}\,x{=}1],\ t_1 \mapsto []\};\\ \ L := \text{NONE}\ \rangle\end{array}$$



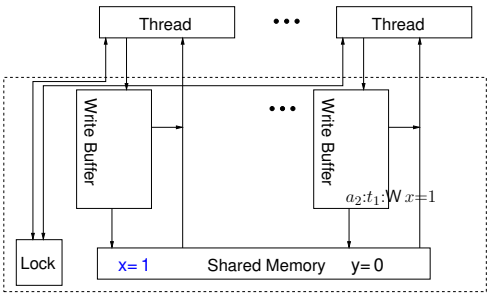
$$\xrightarrow{a_2{:}t_1{:}\text{W}\,y{=}1} \begin{array}{l}\langle M := \{x \mapsto 0,\ y \mapsto 0\};\\ \ B := \{t_0 \mapsto [a_1{:}t_0{:}\text{W}\,x{=}1],\ t_1 \mapsto [a_2{:}t_1{:}\text{W}\,y{=}1]\};\\ \ L := \text{NONE}\ \rangle\end{array}$$



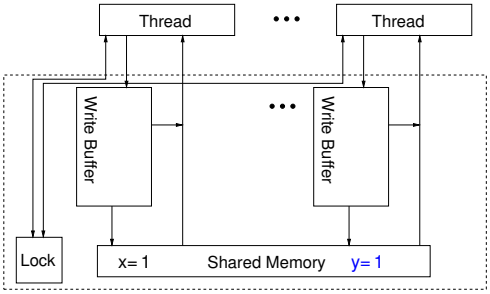
$\xrightarrow{a_3{:}t_0{:}\text{R}\,y{=}0}$ ...the same x86-TSO memory model state

$\xrightarrow{a_4{:}t_1{:}\text{R}\,x{=}0}$ ...the same x86-TSO memory model state

$$\xrightarrow{a_5{:}t_0{:}\mathsf{D}_{a_1:t_0:\mathsf{W}\,x=1}\;x=1} \quad \begin{array}{l}\langle M := \{x \mapsto 1,\ y \mapsto 0\};\\ B := \{t_0 \mapsto [],\ t_1 \mapsto [a_2{:}t_1{:}\mathsf{W}\,y{=}1]\};\\ L := \textsc{None}\ \rangle\end{array}$$



$$\xrightarrow{a_6{:}t_1{:}\mathsf{D}_{a_2:t_1:\mathsf{W}\,y=1}\;y=1} \quad \begin{array}{l}\langle M := \{x \mapsto 1,\ y \mapsto 1\};\\ B := \{t_0 \mapsto [],\ t_1 \mapsto []\};\\ L := \textsc{None}\ \rangle\end{array}$$



Putting those together, we have a complete trace:

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

in which both threads read 0, as required.

This example, and the x86-TSO model, are small and simple enough that this is viable, but it quickly becomes error-prone and tedious. Manual exploration of these models usually isn't viable: one needs tools to reliably compute the allowed transitions of the model. An even more detailed view would explain why each premise of each of the operational rules holds, at each step – showing the derivation of each transition.

Reasoning directly why the model does *not* allow some behaviour for a litmus test requires one to enumerate *all* of the executions that the memory model allows and that are consistent with the instruction semantics of the test, not just exhibit the single execution of interest. The combinatorial explosion of the number of different interleavings makes that much worse, as we already saw for SC in §1.4; it needs tool support or more sophisticated reasoning.

7.3 x86-TSO operational example: spinlocks

For a simple real-world example, we look at the implementation of spinlocks from an old version of Linux.

If one has multiple threads concurrently operating on the same data, the simplest way to avoid confusion is to enforce *mutual exclusion* between the operations using some kind of locking. For example if one has one thread trying to add `1` to `x` in parallel with another trying to add `7` to `x`:

```
int x=0;
```

```
r0 = x;          |  r1 = x;
x  = r0 + 1;     |  x  = r1 + 7;
```

then in an SC language, or a naive compilation of C to an SC or x86-TSO machine, the two might concurrently read the initial value of `x` and then write `1` and `7` (in one or the other order), with an unintended `1` or `7` final result instead of the intended `8`. In ISO C, this program is illegal as-is: it has undefined behaviour because of the potential for concurrent accesses to the "non-atomic" location `x`, as we'll explain in Part **??**.

One can prevent these problems by protecting the potentially concurrent accesses to `x` with a lock, something like this:

```
int x = 0;
lock_t l = LOCK_UNLOCKED;

lock(&l);          |  lock(&l);
r0 = x;            |  r1 = x;
x  = r0 + 1;       |  x  = r1 + 7;
unlock(&l);        |  unlock(&l);
```

In an SC semantics the `lock()`s and `unlock()`s would eliminate most of the otherwise-possible interleavings of the two threads: there would be just two possible executions, in which one or other thread successfully takes the lock first, and then the other thread would be blocked in its `lock()` call until the first executes its `unlock()`.

To implement those `lock()` and `unlock()` functions, there are many possible locking algorithms, with different properties (performance under different workloads, fairness, interaction with an ambient software-thread scheduler, etc.). The simplest is a *spinlock*, in which `lock()` repeatedly checks the lock value and, if it is the `LOCK_UNLOCKED` value, sets it to some `LOCK_LOCKED` value indicating that the lock is taken, while the `unlock()` resets the lock value to `LOCK_UNLOCKED`.

Even in SC, the `lock()` clearly has to check and set the value *atomically*, otherwise multiple threads could all read an `LOCK_UNLOCKED`, write `LOCK_LOCKED`, and continue into their critical section of code that should have been protected.

A real-world spinlock implementation is more complicated. We'll look at one from the Linux kernel, version 2.6.24[1]. This is now an old version, released in 2008, but it's instructive and fairly simple. One might now usually use different lock algorithms, or phrase the implementation differently.

The representation First there is the representation: these spinlocks use a `raw_spinlock_t` struct with an `unsigned int` member `slock`, though the code only actually uses the low-order byte of that, considered as a signed one-byte value. The lock is deemed to be free if that holds `1` (`__RAW_SPIN_UNLOCKED`), and taken if it holds a zero or negative value; it's an implicit invariant that it never holds a positive value more than `1`.

```
typedef struct {
  unsigned int slock;
} raw_spinlock_t;

#define __RAW_SPIN_LOCK_UNLOCKED        { 1 }
```

[1]The original source is in https://elixir.bootlin.com/linux/v2.6.24.7/source/include/asm-x86/spinlock_types.h and https://elixir.bootlin.com/linux/v2.6.24.7/source/include/asm-x86/spinlock_32.h.

The lock implementation The `lock()` implementation is `__raw_spin_lock()` below. This is a C function taking a pointer `lock` to a `raw_spinlock_t`, but the body of the function is embedded x86 assembly, with our comments added. In all this, one has to take care not to confuse the x86 LOCK'd instructions and this implementation of language-level spinlocks.

Line 4 does a LOCK'd decrement of (one byte of) the function argument `lock->slock` (the `%0`): an atomic read of the lock value and write of that minus one. If the result of the decrement is non-negative, then the lock must have held `1`, which means this thread has now successfully acquired the lock and can proceed into its critical section; Line 5 jumps to label `3` and `__raw_spin_lock()` returns. Otherwise, the lock must have held `0` or less, which means some other thread holds the lock and this one has to spin. It first runs an inner loop, reading the lock on line 8 and jumping back to label `2` if it is still `0` or negative. Otherwise, it jumps back to label `1`, the start of the outer loop, to retry the LOCK'd decrement.

```
1  static inline void __raw_spin_lock(raw_spinlock_t *lock)
2  {
3    asm volatile("\n1:\t"                    // label 1 - outer spin
4                 LOCK_PREFIX " ; decb %0\n\t" //   LOCK'd decrement of lock->slock
5                 "jns 3f\n"                  //   if non-negative, jump to 3
6                 "2:\t"                      // label 2 - inner spin
7                 "rep;nop\n\t"               //   performance optimisation
8                 "cmpb $0,%0\n\t"            //   read lock->slock and compare with 0
9                 "jle 2b\n\t"                //   if less than or equal 0, jump to 2
10                "jmp 1b\n"                  //   jump to 1
11                "3:\n\t"                    // label 3 - enter critical section
12                : "+m" (lock->slock) : : "memory");
13 }
```

This nested loop structure is a performance optimisation: microarchitecturally, the LOCK'd decrement needs to get exclusive read/write access to the cache line holding the lock, so if multiple threads are contending for the lock, that will be continually passed between them. The inner loop, on the other hand, just needs read access, and cache protocols typically allow multiple hardware threads to simultaneously have that.

The Line 7 `rep;nop` is another performance optimisation: a `PAUSE` instruction which is a hint to the hardware that this is a spin-wait loop[2].

The x86-TSO model only covers the architected functional behaviour, so one cannot see or reason about these performance effects in terms of the model. As we noted in §1.6, architectures typically have to be loose specifications, to accommodate differences in performance from one hardware implementation to another, and to avoid the complexity that a performance model for even a single implementation would involve, so this is essentially forced. Whether one can build intermediate-level models, that allow some generic reasoning about performance effects, is an especially interesting question given the current importance of side-channel attacks, but we do not consider it here.

[2]The Intel documenentation [62, Vol.2B 4-235 *Pause—Spin loop Hint*] says:

> *Improves the performance of spin-wait loops. When executing a "spin-wait loop," processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops. An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor's power consumption.*

The unlock implementation The unlock implementation has two versions: the first just writes `1` to (the low order byte of) `lock->slock`, while the second uses an `xchgb` to do the same (recall that `xchg` is implicitly LOCK'd).

```
/*
 * __raw_spin_unlock based on writing $1 to the low byte.
 * This method works. Despite all the confusion.
 * (except on PPro SMP or if we are using OOSTORE, so we use xchgb there)
 * (PPro errata 66, 92)
 */

#if !defined(CONFIG_X86_OOSTORE) && !defined(CONFIG_X86_PPRO_FENCE)

static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
  asm volatile("movb $1,%0" : "+m" (lock->slock) :: "memory");
  // write (byte) lock->slock = 1
}

#else

static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
  char oldval = 1;

  asm volatile("xchgb %b0, %1"
               : "=q" (oldval), "+m" (lock->slock)
               : "0" (oldval) : "memory");
  // XCHG (byte) lock->slock = 1
}

#endif
```

This difference is essentially the same as that debated in the Linux kernel mailing list discussion we mentioned in §3.1 – but now we can try to understand whether (and why) each is correct with respect to the x86-TSO operational model. We'll focus on an example, a version of our above example with the first `__raw_spin_unlock()` implementation:

```
int x=0;
raw_spinlock_t l;
l.slock = __RAW_SPIN_LOCK_UNLOCKED;
```

`__raw_spin_lock(&l);`	`__raw_spin_lock(&l);`
`r0 = x;`	`r1 = x;`
`x  = r0 + 1;`	`x  = r1 + 7;`
`__raw_spin_unlock(&l);`	`__raw_spin_unlock(&l);`

Fig. 7.1 sketches one execution of this with the x86-TSO operational model, showing the memory model state and events for the spinlock (writing l for the address of `l.slock`). The accesses to `x` are indicated schematically, while all other accesses, e.g. for the function calls, are omitted altogether (they should all be to thread-local locations, so are not very interesting). The comments indicate what's going on in the code of the two threads.

	x86-TSO memory model state				Thread transitions	
	Memory	Write buffers		Lock		
	M	$B\,t_0$	$B\,t_1$	L	Thread 0	Thread 1
1	$\{l \mapsto 1\}$	[]	[]	NONE		
2					`__raw_spin_lock`	`__raw_spin_lock`
3					`lock decb` R	
4	$\{l \mapsto 1\}$	[]	[]	SOME t_0		t_1 blocked
5					R l=1; W l=0	
6	$\{l \mapsto 1\}$	[W l=0]	[]	SOME t_0		
7					D l=0	
8	$\{l \mapsto 0\}$	[]	[]	SOME t_0		
9					U	
10	$\{l \mapsto 0\}$	[]	[]	NONE	return	t_1 not blocked
11					critical section	
12					...read and write `x`	`lock decb` R
13	$\{l \mapsto 0\}$	[]	[]	SOME t_1	t_0 blocked	
14						R l=0; W l=−1
15	$\{l \mapsto 0\}$	[]	[W l=−1]	SOME t_1		
16						D l=−1
17	$\{l \mapsto -1\}$	[]	[]	SOME t_1		
18						U
19	$\{l \mapsto -1\}$	[]	[]	NONE	t_0 not blocked	spin inner loop
20					...read and write `x`	
21					...read and write `x`	`cmpb` R l=−1
22					...read and write `x`	`cmpb` R l=−1
23					...read and write `x`	`cmpb` R l=−1
24					end of critical section	
25					`__raw_spin_unlock`	
26					`movb` W l=1	
27	$\{l \mapsto -1\}$	[W l=1]	[]	NONE		`cmpb` R l=−1
28					return	
29	$\{l \mapsto -1\}$	[W l=1]	[]	NONE	...subsequent code	`cmpb` R l=−1
30					D l=1	
31	$\{l \mapsto 1\}$	[]	[]	NONE	...subsequent code	
32					...subsequent code	`cmpb` R l=1
33					...subsequent code	spin outer loop
34					...subsequent code	`lock decb` R
35	$\{l \mapsto 1\}$	[]	[]	SOME t_1	t_0 blocked	
36						R l=1; W l=0
37	$\{l \mapsto 1\}$	[]	[W l=0]	SOME t_1		
38						D l=0
39	$\{l \mapsto 0\}$	[]	[]	SOME t_1		
40						U
41	$\{l \mapsto 0\}$	[]	[]	NONE	t_0 not blocked	return
42	$\{l \mapsto 0\}$	[]	[]	NONE	...subsequent code	critical section
43					...subsequent code	...read and write `x`

Figure 7.1: Example spinlock execution

In this execution, at a high level, Thread 0 tries and succeeds to take the spinlock `l` first, and enters its critical section to read and write `x`, then Thread 1 tries to take the spinlock but fails, and hence spins, re-reading `l` repeatedly until Thread 0 releases the spinlock. Thread 0 then continues with whatever the subsequent code is, while Thread 1 enters its critical section to read and write `x`. Looking more closely, there are several interesting things to note, about this algorithm, the x86-TSO model, and operational modelling in general.

At the start, both threads enter the `__raw_spin_lock` function and are ready to begin their `lock decb` instructions. In this execution, Thread 0 does the initial L transition first, so Thread 1 cannot begin its `lock decb` until Thread 1 gets to its U transition (by the premise $m.L = \text{NONE}$ of the **L: Lock** rule); it is blocked (in the sense defined as part of the model).

As part of its `lock decb`, Thread reads the initial spinlock value, at Line 5. In principle any read might be either from memory or the local write buffer. In the initial state of this example, the write buffers are all empty and there is no preceding code, so this has to be the $l = 1$ from the initial-state memory – but even if there was some write to l previously in the local buffer, the **L: Lock** rule requires the local buffer to have drained to memory before the rest of the LOCK'd instruction can execute, in its $m.B(t) = [\,]$ precondition.

At Line 5 Thread 0 writes $l = 0$ to indicate that the spinlock is now taken; that goes into its write buffer. The **U: Unlock** rule requires that any thread t's write buffer has drained before it can do a U transition ($m.B(t) = [\,]$), so this cannot happen until the D l=0 transition of the **DM: Dequeue write from write buffer to memory** rule – so memory is guaranteed to contain the 0 value before Thread 1 is unblocked.

Thread 0 then returns from `__raw_spin_lock` (in an optimised compilation, that might well have been inlined, but that affects only thread-local accesses; it doesn't affect anything we're discussing here), and enters its critical section, ready to read and write `x`.

Meanwhile, at Line 12 Thread 1 begins its `lock decb` (it can do this because the x86-TSO memory model machine lock is not held, even though at a higher level of abstraction the spinlock is). It reads the lock value $l = 0$, decrements it, and writes $l = -1$; again that has to be written to memory (with the Line 16 dequeue event D l=-1) before the U at the end of this instruction. Because the result of the decrement was negative, Thread 1 enters its inner spin loop (the instruction semantics for the `jns 3f` on Line 5 of `__raw_spin_lock` will just fall through to the next instruction).

Between the Thread 1 L and U, Thread 0 was blocked, so it couldn't read from memory (**RM**) or its write buffer (**RB**), dequeue writes from its write buffer (**DM**), or start another LOCK'd instruction (**L**). It could do any register-only instructions, write to its write buffer (**WB**), or (if its write buffer is empty) `mfences` (**M**), as those rules don't have a $\text{not_blocked}(m, t)$ or $m.L = \text{NONE}$ precondition, but none of those would be observable to other threads. This illustrates an interesting point about the design of operational models: there are often some minor choices between variants which don't affect the set of extensional observable behaviour that the model allows, but might be more or less intuitive or convenient to work with. Here, we could have required that all those rules have a not_blocked precondition. That would have slightly simplified reasoning about examples like this one, as one would know that nothing else could happen during a LOCK's instruction, but it would have been slightly further from microarchitectural intuition and the vendor documentation. It would change the set of allowed traces of events, but we conjecture that it would not change the set of allowed final states. In some cases, one might want to define multiple variants of a model and prove such equivalence results.

Thread 1 might then repeatedly read the lock value, with its `cmpb` instruction, but for the moment each time it will read the -1. The `cmpb` will thus set flags which will make the subsequent `jle 2b` jump back to the start of the inner spin loop, to read the lock value again. The `pause` instruction (the `rep;nop`) has no effect in the x86-TSO machine.

Eventually Thread 0 reaches the end of its critical section (Line 24) and calls

`__raw_spin_unlock`. This just does a write of $l = 1$, which of course initially just goes into the Thread 0 write buffer. Thread 1 thus might continue to spin, reading $l = -1$ from memory (Lines 27 and 29) while Thread 0 continues with its subsequent code. The effect of this buffering of the unlock write is thus effectively just to extend the time that Thread 0 holds the lock, as far as Thread 1 is concerned.

At this point the earlier Thread 0 write to `x` (not shown) might have been dequeued to memory or might still be in Thread 0's buffer, below the write of l – recall that these buffers are FIFO. The Thread 0 subsequent code might involve writes to other locations, which again will initially just go into its buffer, but above the write of l.

At Line 30 the unlock write of $l = 1$ is finally dequeued. Nothing in the x86-TSO definition requires transitions to be taken as soon as they are enabled – indeed, one couldn't require that, as there are often multiple enabled transitions. For non-terminating executions there are questions of *fairness*, i.e., whether an enabled transition must eventually be taken, but we will not consider those here.

For that Line 30 dequeue to happen, Thread 0's buffer must first have drained to memory, so the Thread 0 write to `x` must now be in memory.

Note that at Line 27 the Thread 0 write of l and Thread 1 read of l are both enabled (and at 30 the Thread 1 dequeue of that write and a Thread 1 read of l are): the implementation of the spinlock, like that of other synchronisation primitives, necessarily involves some kind of *data race*, even though correct code using them might not. We'll return to data races in detail in Part **??**.

The dequeue of $l = 1$ means that the next Thread 1 inner-loop `cmpb` read of l will see $l = 1$, and will thus fall through its `jle` and `jmp 1b` back to its outer loop, to attempt again to take the lock with its `lock decb`. This time that succeeds (temporarily blocking t_0 between the L and U), whereupon Thread 1 returns from `__raw_spin_lock` and enters its critical section, to read and write `x`. One might wonder why the algorithm, following a successful read of a positive value from l in the inner spin loop, has to re-read l in the outer-loop `lock decb`. That's because nothing prevents another thread taking the lock between that successful read. The time window for that is small, but it could (and eventually would) happen – and one obviously needs such algorithms to be correct in general, not just most of the time, as otherwise there will be unpredictable (and hard to debug) errors in all the higher-level software using them.

As a result of all this, at least in this execution, Thread 1 will correctly read the result of Thread 0's write to `x`.

Underflow One should ask whether the decrements of the `lock decb` can underflow, below -128. With only two threads, it cannot, basically as each thread can only decrement l once, before spinning. If, while the lock is taken and Thread 1 is in its inner spin loop, additional threads were to try to take the lock, each would decrement l again on their first outer-loop `lock decb`, so if there are more than 128 hardware threads, this algorithm is wrong. When the code was first written, that might legitimately have seemed far in the future, but at the time of writing of this text (2023), x86 processors with 256 hardware threads are available. It would be quite unlikely to run such an old version of Linux on modern hardware, but in general it is hard to predict how long software will continue to be used, and how future hardware might evolve.

Proof of correctness This example shows some interesting cases, but it is far from a proof, or even a statement, that this spinlock implementation works in general above x86-TSO. It is just a single allowed trace of a single example usage of the spinlock implementation, and moreover just one-shot (without repeated use of the lock by each thread) and just on two threads. However, one can state and prove correctness above x86-TSO, as done by Owens [102]. There are some interesting subtleties involved that we won't go into here, but at a high level, his [102, Theorem 2] says that, if an x86 program uses spinlocks correctly with respect to an SC semantics

– i.e., if in every such execution, every pair of competing non-spinlock events are separated by the release and acquire of a spinlock – then any x86-TSO execution of the program has an SC execution with the same subsequence of writes, and with reads reading from the corresponding write. The proof uses a generally useful notion of *triangular race freedom* [102, §2]. This is our first example of proving the soundness of a higher-level (and in this case simpler) memory model above an architectural memory model.

Returning to the Linux mailing list discussion We can now return to the Linux kernel mailing list discussion mentioned in §3.1, which was about a similar (though not identical) spinlock implementation. The initial concern was about potential CPU reordering of instructions around the `spin_unlock` code. With respect to the x86-TSO operational model (and thus also with respect to any hardware implementations for which it is a sound model), one can see informally from the example that this is not a problem. The only observable reordering in the model arises from store buffering (the instructions of each thread execute in-order), and the store buffers are FIFO. The unlock write might be buffered, but that FIFO buffering means that Thread 0's writes to `x` within its critical section will be visible to Thread 1 before the unlock write is, even though the buffering means that the unlock might be delayed (in abstract-machine execution time) w.r.t. when one might think it occurs. To make a fully rigorous argument, one also needs to consider various other cases – which is what the proof by Owens effectively does.

The variant implementation of `__raw_spin_unlock` uses an implicitly LOCK'd `xchgb`. Thread 1 would therefore be blocked between the L and U of that, but that would just delay some of the Thread 1 inner-loop reads (e.g. Lines 27, 29). More interestingly, it would require the Thread 0 buffer to have drained before the L and before the U. That would mean the spinlock is guaranteed to be available to Thread 1 before the Thread 0 subsequent code executes, but that doesn't really affect anything.

However, the original comment for the variant implementation says that this was necessary on some particular (Pentium Pro) x86 implementation, due to *processor errata* 66 and 92. Processor vendors typically document errata that arise in specific hardware implementations, sometimes with workarounds, and it is not very uncommon (though unfortunate) for there to be some that involve their relaxed memory model behaviour. The Pentium Pro is now only of historical interest, but one can see in the 1999 *Pentium Pro Processor Specification Update* [81] that these errata are *Delayed line invalidation issue during 2-way MP data ownership transfer* and *Potential loss of data coherency during MP data ownership transfer*, with a documented workaround for the latter: *"Deterministic barriers beyond which program variables will not be modified can be achieved via the usage of locked semaphore operations. These should effectively prevent the occurrence of this erratum"*.

Another question is whether there might be compiler reordering of accesses around the `__raw_spin_lock` and `__raw_spin_unlock` implementations. The `volatile` and the `memory` clobber of the GCC inline assembly prevent this [75].

7.4 Discussion

Restoring SC In §3.2 we noted in connection with the IRIW test that some JVM implementations historically relied on `mfence` recovering SC, for their implementation of the intended higher-level memory model to be correct. For x86-TSO, adding an `mfence` between every pair of memory accesses does restore SC. More precisely, if the program executed by the thread semantics has an `mfence` between every pair of memory accesses, then any execution in x86-TSO will have essentially identical behaviour to the same program with `nop`s in place of `mfence`s in SC. Here "essentially identical" means that they have the same set of interface traces except with the F and $\mathsf{D}_w\, x{=}v$ events erased. This would not normally be a sensible implementation, of course, as it would perform very badly, but it is an important first property of the model.

Hardware and software threads The threads of the x86-TSO model are hardware threads, not cores or software threads. Many processor implementations have some form of *simultaneous multithreading* (SMT) or *hyperthreading*, in which multiple hardware threads execute on a single core, sharing some hardware resources. As far as the functional behaviour of user code is concerned, these behave as independent hardware threads, each with their own x86-TSO write buffer, though they may well have different performance (and side-channel) properties.

Most software is written above some software thread abstraction, with an operating system or hypervisor multiplexing many software threads onto fewer hardware threads, and context switching between them as required (only the implementation of critical parts of an operating system or hypervisor will typically be "bare metal" for x86 processors). For this to be correct, the context switching code has to ensure that the hardware-thread write buffer that has been used by one software thread is drained before switching that hardware thread to run another software thread. Given that, x86-TSO is also a sound model for software threads: the model the OS and hardware together provide to the application binary code.

x86-TSO is an *abstract* machine We reiterate that this x86-TSO operational model is an *abstract* machine: it is a conceptual tool to specify the envelope of architecturally intended *programmer visible behaviour* of any x86 processor. It is based on some hardware intuition, but it is not a description of real hardware implementation internals. Those are now exceedingly complex, with many sophisticated internal optimisations that include out-of-order execution. The force of the model is really that, of those, only per-thread FIFO write buffers are (ignoring timing and performance effects) visible to programmers.

Note also that the model is intentionally a rather loose specification, in several ways:

- The write buffers of the model are unbounded, while specific hardware implementations will have particular sizes. Those might or might not be documented, and other microarchitectural optimisations might have also the observable effect of write buffering, and software should be written to be correct for any size. The model allows the behaviour of implementations with any specific size, and so any software that is correct above the model is correct above any such implementation that conforms to it.

- The model allows an arbitrary interleaving of any enabled transitions; it doesn't give any progress or fairness property. Again, this allows the wide variation that might occur in implementations, and software should be written to be correct irrespective of potential delays of specific hardware threads. In particular, the dequeue transitions of the model can be taken – or not – whenever they are enabled. In some cases one might need to know whether write-buffer entries are guaranteed to eventually drain. The original definition of x86-TSO [101, 103] did include such a condition, but we won't cover it here; we'll only consider finite executions.

x86-TSO and SPARC TSO x86-TSO is based on, and very similar to, the SPARC Total Store Order model from the early 1990s [3, App.K], [4, Ch.8 and App.D], [127]. SPARC defined three models:

- TSO (Total Store Order)

- PSO (Partial Store Order)

- RMO (Relaxed Memory Order)

though to the best of our knowledge, only TSO was used in practice (implementations were either not as weak as PSO or RMO, or configuration registers were used by software to disable those weaknesses).

Those definitions were in an axiomatic style, but SPARC TSO is essentially an axiomatic characterisation of the behaviour arising from per-thread FIFO write buffers. x86-TSO is extensionally very similar except that it covers the x86 CISC instructions with multiple memory accesses, not the SPARC RISC instructions, and covers the x86 `mfence` barrier and LOCK'd instructions rather than the specific RMW operations of SPARC.

7.5 Exercises

Exercise 7.1 *Write the TSO transition system for the litmus tests of Chapter 2, with a state diagram for each node (including the contents of buffers).*

Exercise 7.2 *Design an operational model for Partial Store Ordering (PSO), in which writes to different locations are not ordered with each other.*

Exercise 7.3 *Identify an execution of a program that is allowed by PSO but not by TSO.*

Exercise 7.4 *Design an operational model for a variant of PSO in which addresses equal modulo 2^n share a buffer.*

Exercise 7.5 *Design an equivalent formulation of TSO based on FIFO* load *buffers. Hint: when a thread performs a write, always add it to the thread's load buffer, remembering that it was performed by that thread (see [15]).*

Exercise 7.6 *Design an operational model in which visibility of a write implies visibility of previous writes by that thread, and writes seen by the time of those writes (transitively). See release/acquire, and specifically [85].*

Chapter 8

Making the operational models executable as a test oracle: the RMEM tool

As we have seen, computing the allowed executions of a concurrency model, even for small examples, quickly becomes challenging and error prone. The RMEM tool [117] (https://github.com/rems-project/rmem) lets one interactively or exhaustively explore the behaviour of various operational models for x86, Arm-A, IBM Power, and RISC-V. RMEM (originally 'ppcmem') has been developed from 2010 onwards, for operational models for IBM Power [116], Power load-reserve/store-conditional synchronisation [115], integration with medium-scale instruction semantics using the first version of Sail [77], the "Flowing" and "POP" models for non-multicopy-atomic Arm-A, including integration with medium-scale instruction semantics [70], mixed-size accesses for Power and Arm-A [71], the "Flat" model for multicopy-atomic Arm-A [106], integration with full-scale Sail instruction semantics for Arm-A and RISC-V [39], the Promising-Arm model [107], and Arm-A instruction-fetch and instruction/data cache maintenance [125]. RMEM has been developed mainly by Susmit Sarkar, Peter Sewell, Luc Maranget, Shaked Flur, Christopher Pulte, Jon French, and Ben Simner, with contributions from Scott Owens, Pankaj Pawan, Francesco Zappa Nardelli, Sela Mador-Haim, Dominic Mulligan, Ohad Kammar, Jean Pichon-Pharabod, Gabriel Kerneis, Alasdair Armstrong, Thomas Bauereiss, and Jeehoon Kang (all in roughly chronological order).

RMEM has two user interfaces: it can either be run in-browser, with the web interface, or one can install it locally and use the command-line interface (CLI). The two have essentially the same functionality; the web interface is easier to get started with and more convenient, but the command-line version has much better performance, which is essential for larger tests. There is a rich collection of features – we return later to more of those, and to its internals, but for now just introduce some basic usage.

Installing RMEM locally To install RMEM locally, to use the command-line interface:

1. follow the instructions at https://github.com/rems-project/rmem

Documentation is at https://github.com/rems-project/rmem. One can then explore a test like SB in the x86-TSO model by:

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

In this default interactive model, it prints the current model state and its possible transitions at each step, as shown in the Fig. 8.1 screenshot (this is the same data as the web interface console pane). There are many useful commands, including:

```
help                        list commands
```

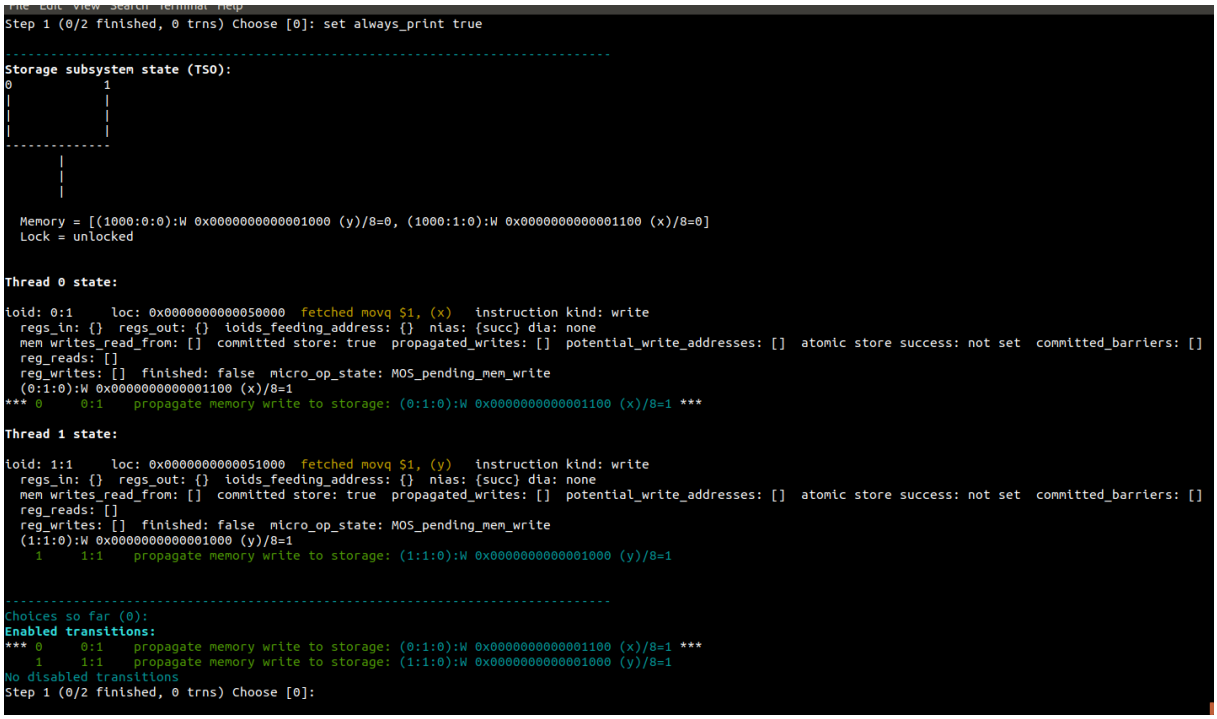


Figure 8.1: RMEM command-line interface

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

For non-interactive exhaustive search:

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

which shows the allowed final states and paths to them:

```
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"`.

The candidate execution diagrams in this text are mostly produced automatically by RMEM and Flur's `litmus-latex` package [69], with commands such as:

```
rmem -interactive false -eager true -hash_prune true -pp_hex false -dot_final_ok true \
```

```
-graph_backend tikz -model relaxed -dot_dir litmus-tikz-figures/   MP.litmus
```

that produces `MP.tikz` for the execution diagram and `MP.states`; litmus tests are typeset with the `\litmusassem` command.

The RMEM web interface To use the web interface, go to http://www.cl.cam.ac.uk/users/pes20/rmem, load an x86 litmus test such as SB with the "Load litmus" menu, clicking "Load from library", "x86", and "SB", then set "All eager" in the Execution menu. There are several possible pane contents: State shows the current model state; Graph shows the candidate execution built up so far; Sources shows the litmus source of the test; Console shows messages and lets one type commands; Trace show the trace of transitions so far; and Help shows the documentation. One can split panes horizontally or vertically, remove them, resize them, or change their contents.

One can then explore the allowed x86-TSO transitions interactively. At each step, the enabled transitions are listed at the end of the State pane and highlighted in the Graph, and one can choose them either by clicking on those or typing their index number from the list into the Console (RMEM doesn't use exactly the same rule names or event names as those we have seen, but the correspondence should be clear). Buttons at the top let one go back, next, or restart. The RMEM version of the model includes semantics for a fragment of the x86 instruction set, which adds some complexity to the display and options – for example, without "All eager", one can step through the individual steps within each instruction. The "Link to this state" menu at the top gives one a link to share. Figures 8.2–8.5 show screenshots from an interactive execution of SB. Often, one wants to interactively search for an execution which satisfies the final state constraint from the test, to see why it is allowed. Interactive exploration is not so good for understanding why some final state is not allowed, both because that involves quantifying over all possible executions, and because the interface only shows which transitions are allowed in the current state; it doesn't show which transitions are not allowed, or the reasons why not, as that would be overwhelming.

One can also automatically explore what the model allows – normally from the initial state for a test. The "Search" menu at the top lets one explore all the allowed behaviours, either just a single random trace or exhaustively (though for larger examples this sometimes hits a resource limit or takes too long in the web interface, whereupon one should use the command-line interface instead). The latter does an exhaustive enumeration of the entire transition system for a test, with the default "hash prune" optimisation to prevent duplicate exploration of the same state. This is essential for reasonable performance, as there are many uninteresting interleavings, e.g. where two independent events could happen in either order. For SB the web-interface exhaustive exploration works, printing

```
States 4
2     *>0:RCX=0x0; 1:RCX=0x0;  via "0;0;1;0;2;1"
2     :>0:RCX=0x0; 1:RCX=0x1;  via "0;0;1;2;0;1"
2     :>0:RCX=0x1; 1:RCX=0x0;  via "0;1;1;2;3;0"
2     :>0:RCX=0x1; 1:RCX=0x1;  via "0;1;2;1;3;0"
```

to the console. This shows the four reachable final states (considering just the values of the registers used in the test's final condition), with a $*$ for the state satisfying the final condition. For each it shows a "follow list": a sequence of transition indices that reaches such a state (this is just one of potentially many such sequences). One can click on those to set a sequence of default transitions and then step through those just by pressing `return`.

8.1 Exercises

Exercise 8.1 *exhibit all the four major behaviours of MP, and of SB*

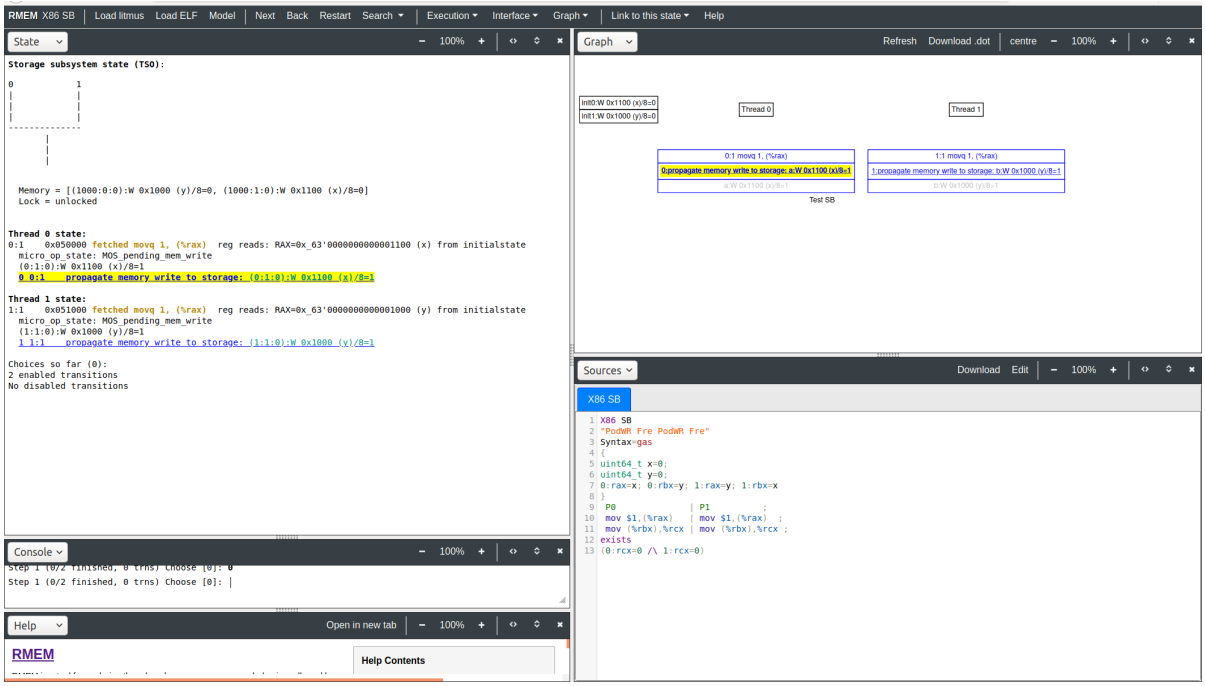
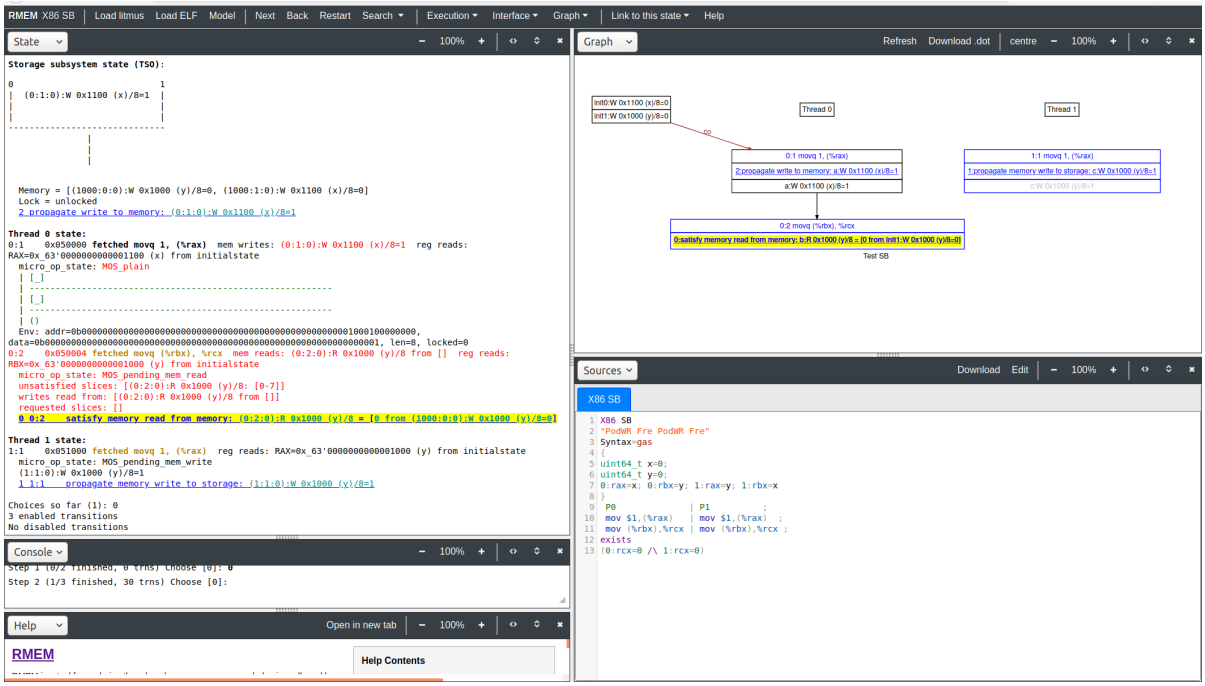


Figure 8.2: RMEM web interface: SB (1 of 4)



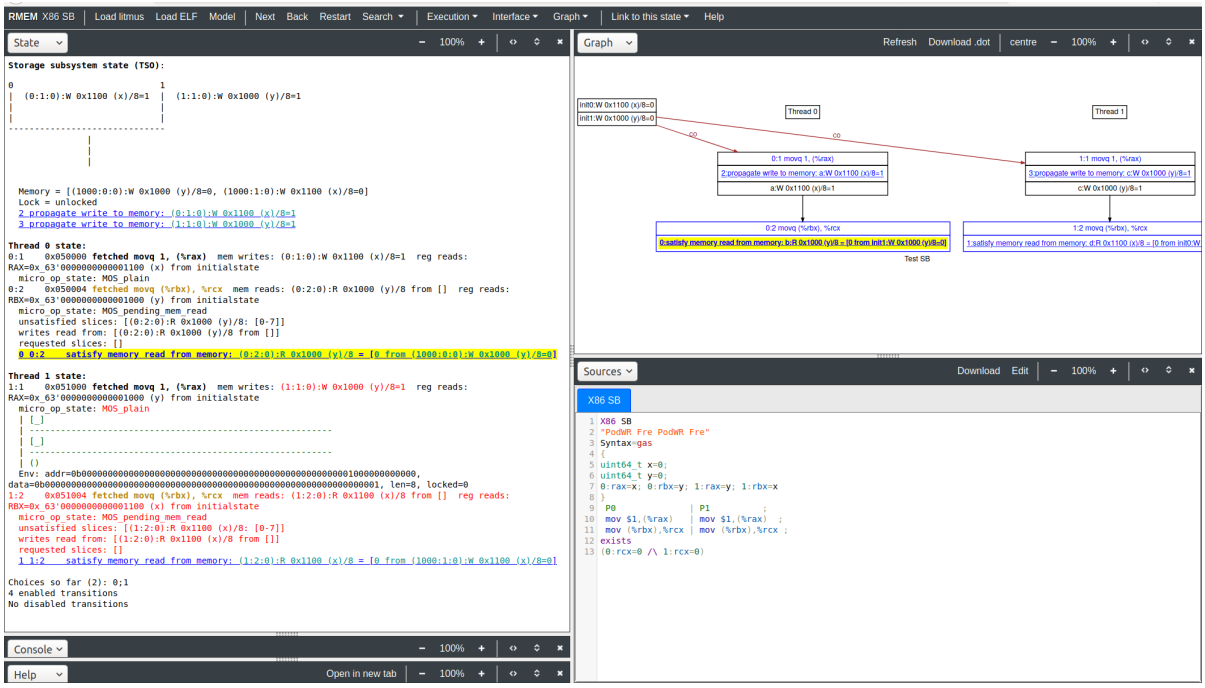
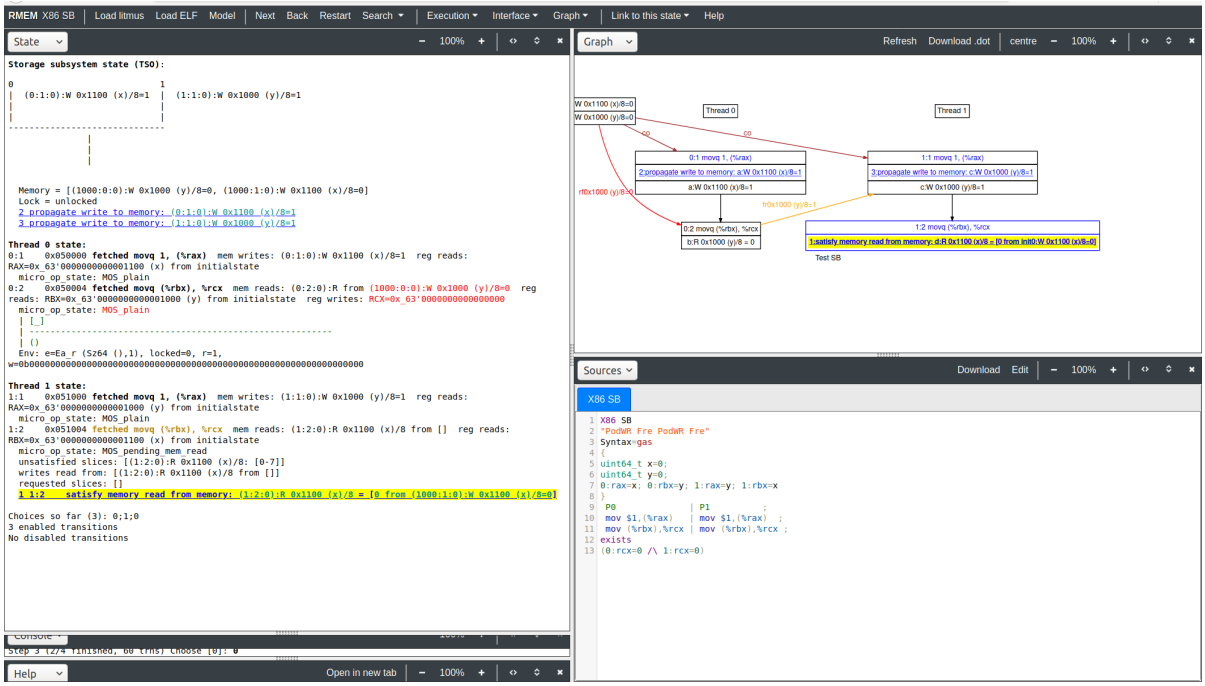


Figure 8.3: RMEM web interface: SB (2 of 4)



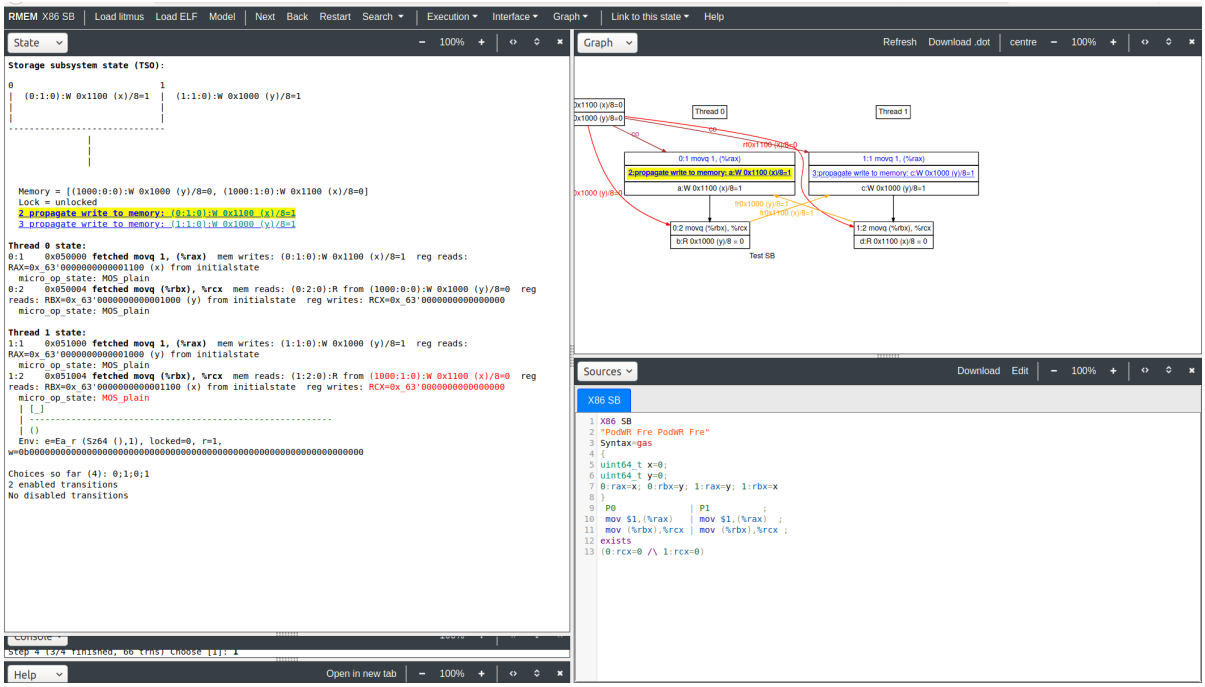


Figure 8.4: RMEM web interface: SB (3 of 4)

Figure 8.5: RMEM web interface: SB (4 of 4)

Chapter 9

SC, axiomatically

The SC and x86-TSO operational models define the allowed behaviour as the observable behaviour of abstract machines: one can construct allowed executions incrementally, by picking from among the enabled transitions at each state, and a final state is allowed iff there is some execution of the machine leading to it. In contrast, axiomatic models typically aim to characterise the allowed behaviour more explicitly, with a predicate that defines whether an arbitrary complete candidate execution graph is allowed or not.

In this chapter we'll describe an axiomatic model for SC. We'll first discuss coherence, which semantically is essentially per-location sequential consistency, then give an axiomatic model for SC and prove it equivalent to the operational SC model of Chapter 6. In the next chapter we'll give an axiomatic model for x86-TSO. This has been proved equivalent to the operational model, in a mechanised proof in Isabelle [99] by Paul Durbaba [67]; we'll sketch that proof.

Many styles of axiomatic model have been used over the years. The first axiomatic model for x86-TSO, by Owens, Sarkar, and Sewell [101, 103, 120], was based on the SPARCv8 TSO specification [3]; it was formalised in HOL4 [**?**] and proved equivalent to the operational model, with a partly mechanised and partly hand proof [101, 103]. The axiomatic models we describe here are in the "herd" style of Alglave and Maranget [36], including their TSO model [31], to set the scene for the later axiomatic models of Arm-A and RISC-V in that style. We'll return to the differences in styles in the Part **??** related work.

9.1 Execution graphs, informally

The "axiomatic" approach defines the semantics of a program as a set of execution graphs, each such graph representing a potential behaviour of the program. Sometimes, each execution graph also contains a justification for why this behaviour is possible, in which case there might be multiple justifications for a behaviour, thereby exposing some internal details in the semantics. For example, under SC, the semantics of SB is given by the following four execution graphs:

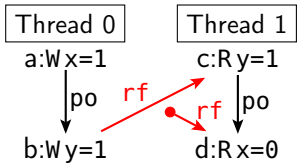
This set of execution graphs contains exactly those execution graphs that are both consistent with the program, and consistent with the axiomatic memory concurrency model proper. In a sense, it is generated by a comprehension:

$$[\![P]\!] = \{X \mid \text{program-consistent}(P, X) \wedge \text{memory-model-consistent}(X)\}$$

The former ensures that the execution is an execution of the program, and the latter ensures that the execution is allowed by the machine. Loosely, the former corresponds to the instruction semantics, and the latter corresponds to the ?? (pipeline, speculation, ...) and and the storage subsystem. (although in pratice, it is sometimes convenient to move some of the responsibility around).

9.2 Execution graphs, formally

An axiomatic concurrency model defines, for any candidate execution graph, whether that is allowed or not. Up to now, we've drawn execution graphs informally, e.g. this candidate execution for MP:



To define axiomatic models, we first have to make precise what an execution graph is, as a mathematical object. It's convenient to split this into two pieces. A candidate execution graph comprises:

- a *candidate pre-execution* X, which is a pair $(E, \xrightarrow{\text{po}})$ of a set E of events and a program-order relation $\xrightarrow{\text{po}}$, and
- a *candidate execution witness* W, which is a pair $(\xrightarrow{\text{rf}}, \xrightarrow{\text{co}})$ of a reads-from relation $\xrightarrow{\text{rf}}$ and a coherence relation $\xrightarrow{\text{co}}$ (introduced in the next section),

that satisfy the well-formedness properties detailed below. These rule out nonsense graphs, prior to any specific memory model requirements. For example, the reads-from relation must be from writes to reads; graphs in which there is a reads-from $\xrightarrow{\text{rf}}$ edge from a write to a write are not well-formed.

Intuitively, the pre-execution arises from some control-flow unfolding of the program for each hardware thread, while the execution witness gives the interactions between threads.

The events E are some subset of the memory write, read, and barrier events defined by the grammers of Chapters 6 and 7. For x86-TSO, the internal dequeue events are not used in the x86-TSO axiomatic model, which abstracts from that operational mechanism. For the moment we will ignore x86 LOCK'd instructions; later the information from the LOCK/UNLOCK events will be represented as a relation rather than with events.

The relations are all binary relations over E, i.e. sets of pairs (e, e'), where $e \in E$ and $e' \in E$ are elements of E (said another way, they are subsets of the set of all such pairs: $\xrightarrow{\text{po}} \subseteq E \times E$). We use some standard properties and operations on binary relations, recalled in Figs. **??** and **??** on pages **??** and **??**.

The well-formedness properties are:

- E is a finite set of events, with unique IDs:
  $\forall e, e'.\ e \neq e' \implies \mathrm{id}(e) \neq \mathrm{id}(e')$
  We restrict E to a finite set for simplicity, to avoid technical issues that arise for infinite executions. Where E is clear from context, we let e range over all events of E, and r and w range over the read and write events of E.
- *program order* $\xrightarrow{\text{po}}$ is an irreflexive transitive relation over E, that only relates pairs of events from the same thread and that is total among those:
  1. $\xrightarrow{\text{po}}$ is irreflexive, i.e., no event is related to itself:
     $\forall e.\ \neg(e \xrightarrow{\text{po}} e)$
  2. $\xrightarrow{\text{po}}$ is transitive, i.e., if $e \xrightarrow{\text{po}} e'$ and $e' \xrightarrow{\text{po}} e''$ then $e \xrightarrow{\text{po}} e''$:
     $\forall e, e', e''.\ (e \xrightarrow{\text{po}} e' \wedge e' \xrightarrow{\text{po}} e'') \implies e \xrightarrow{\text{po}} e''$
  3. $\xrightarrow{\text{po}}$ only relates events from the same thread:
     $\forall e, e'.\ e \xrightarrow{\text{po}} e' \implies \mathrm{thread}(e) = \mathrm{thread}(e')$

4. $\xrightarrow{\text{po}}$ relates all pairs of distinct events from the same thread one way or the other:
   $\forall e, e'.\ (\text{thread}(e) = \text{thread}(e') \wedge e \neq e') \implies e \xrightarrow{\text{po}} e' \vee e' \xrightarrow{\text{po}} e$
   (when one deals with instructions that make multiple memory accesses, this last does not always hold, but we assume it for now)

- *reads-from* $\xrightarrow{\text{rf}}$ is 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 with other reads reading the values of the initial state:

  1. for any reads-from edge, it must be from a write to a read, and they must have the same address and value:
     $\forall e, e'.\ e \xrightarrow{\text{rf}} e' \implies \text{iswrite}(e) \wedge \text{isread}(e') \wedge \text{addr}(e) = \text{addr}(e') \wedge \text{value}(e) = \text{value}(e')$
  2. for any e'', there is at most one source of a reads-from edge to it:
     $\forall e, e', e''.\ (e \xrightarrow{\text{rf}} e'' \wedge e' \xrightarrow{\text{rf}} e'') \implies e = e'$
  3. for any read, if there is no reads-from edge to it, its value must be that of the initial state:
     $\forall e.\ (\text{isread}(e) \wedge \neg \exists e'. e' \xrightarrow{\text{rf}} e) \implies \text{value}(e) = m_{\text{init}}(\text{addr}(e))$

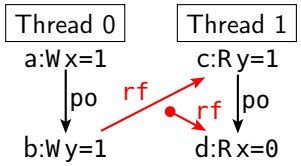
  As a minor technical choice, reads from the initial state are represented by the absence of any $\xrightarrow{\text{rf}}$ edge to the read event, and the first write in the coherence order for each location is implicitly coherence after the initial state, without an explicit edge. One could instead model the initial state as normal writes subject to additional conditions; there are pros and cons of each approach.

- We introduce coherence in the next section, but give its well-formedness conditions here for ease of reference later; one might want to skip over this on a first reading. A coherence relation $\xrightarrow{\text{co}}$ is 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:

  1. $\xrightarrow{\text{co}}$ is irreflexive, i.e., no write is related to itself:
     $\forall e.\ \neg(e \xrightarrow{\text{co}} e)$
  2. $\xrightarrow{\text{co}}$ is transitive, i.e., if $e \xrightarrow{\text{co}} e'$ and $e' \xrightarrow{\text{co}} e''$ then $e \xrightarrow{\text{co}} e''$:
     $\forall e, e', e''.\ (e \xrightarrow{\text{co}} e' \wedge e' \xrightarrow{\text{co}} e'') \implies e \xrightarrow{\text{co}} e''$
  3. $\xrightarrow{\text{co}}$ only relates writes, and only writes to the same address:
     $\forall e, e'.\ e \xrightarrow{\text{co}} e' \implies \text{iswrite}(e) \wedge \text{iswrite}(e') \wedge \text{addr}(e) = \text{addr}(e')$
  4. $\xrightarrow{\text{co}}$ relates all pairs of distinct writes to the same address one way or the other:
     $\forall e, e'.\ (e \neq e' \wedge \text{iswrite}(e) \wedge \text{iswrite}(e') \wedge \text{addr}(e) = \text{addr}(e')) \implies e \xrightarrow{\text{co}} e' \vee e' \xrightarrow{\text{co}} e$

  There are further minor technical choices here that make things more convenient: (a) coherence is in principle a family of disjoint relations, comprising one order per location, but it's more convenient to work with their union as a single relation; and (b) by defining coherence to be an irreflexive total order per location, rather than a reflexive total order per location, we have more convenient definitions of the coherence predecessor and successor of each write.

For example, the above candidate execution graph:



comprises the candidate pre-execution $(E, \xrightarrow{\mathsf{po}})$ and candidate execution witness $(\xrightarrow{\mathsf{rf}}, \xrightarrow{\mathsf{co}})$ where:

$$\begin{array}{lcl} E & = & \{\mathtt{a}{:}t_0\mathtt{:Wx=1},\ \mathtt{b}{:}t_0\mathtt{:Wy=1},\ \mathtt{c}{:}t_1\mathtt{:Ry=1},\ \mathtt{d}{:}t_1\mathtt{:Rx=0}\} \\ \xrightarrow{\mathsf{po}} & = & \{(\mathtt{a}{:}t_0\mathtt{:Wx=1},\ \mathtt{b}{:}t_0\mathtt{:Wy=1}),\ (\mathtt{c}{:}t_1\mathtt{:Ry=1},\ \mathtt{d}{:}t_1\mathtt{:Rx=0})\} \\ \xrightarrow{\mathsf{rf}} & = & \{(\mathtt{b}{:}t_0\mathtt{:Wy=1},\ \mathtt{c}{:}t_1\mathtt{:Ry=1})\} \\ \xrightarrow{\mathsf{co}} & = & \{\} \end{array}$$

and these do satisfy the above well-formedness conditions.

Spelling this execution out in detail (which one wouldn't normally do on paper) highlights a final minor technical choice: one could take the relations to be over events, as we do here, or over event IDs. One often conflates events, metavariables over events, and event IDs in non-mechanised maths.

9.3 Coherence

As we noted in Chapter 6, in SC, viewed operationally, each read reads from the most recent write to memory to the same location, in some interleaving of what the threads do (each executing in program order). In other words, for any SC execution there is a single total order – the order of some SC operational-model trace (including events by all threads and to all locations), with each read reading from the most recent write to the same location in that order.

In more relaxed models, including x86-TSO, that is no longer the case, but architectures typically do guarantee a similar property for normal memory accesses if we consider each location in isolation. This *coherence* property can be stated in several equivalent ways, which we first give in prose and then make precise.

1. in any execution, for each location, there is a total order over all the writes and reads to that location, which is consistent with each thread's program order, and in which each read reads from the most recent write; or,

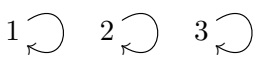
2. in any execution, for each location, the execution restricted to just the writes and reads to that location is SC; or,

3. in any execution, for each location, there is a total order over all the writes to that location, the *coherence order*, and for each thread, the order is consistent with the thread's program-order for its writes and reads to that location, and with reads reading from the most recent write.

Microarchitecturally, processor implementations use elaborate hierarchies of caches, to provide fast access to frequently used memory, while still letting code refer to that with addresses from the common large address space of the underlying shared memory. They use elaborate cache protocols to propagate information between parts of these cache hierarchies. The internal details of these are largely not our concern here, as we are trying to characterise the architecturally allowed behaviour, not the microarchitectural implementations (see computer architecture texts such as Hennessy and Patterson [79] and Handy [78], and high-level descriptions of the storage hierarchy in successive generations of IBM Power processors in [100, 132, 128, 89, 84, 131]. However, it is useful to know that cache hierarchies typically work with *cache line* sized and aligned chunks of memory, perhaps 64 or 128 bytes (these sizes may or may not be architecturally defined or architecturally exposed, and they might not be common across all caches in a system). In a simple cache hierarchy and cache protocol, say with just one cache per hardware thread, for any cache line, at most one hardware thread should have exclusive write access to it at any one time, otherwise writes might be lost. For example,

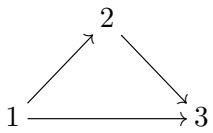
Recall that a binary relation $\rightarrow$ over a set A is a set of pairs (a, a') of elements of A, or in other words a subset $\rightarrow \subseteq A \times A$. One often works with binary relations that have some combination of standard properties.

$\rightarrow$ is *reflexive* if, for all elements a of A, a is related to itself: $\forall a.\ a \rightarrow a$. Here we assume a, a', a'' etc. range over the elements of A, otherwise we would have to write explicit bounds on the quantifier, e.g. $\forall a \in A.\ a \rightarrow a$. We will usually be using relations over the events E of some execution graph, and leave the bounds implicit. For example, fixing $A = \{1, 2, 3\}$, then $\rightarrow = \{(1,1), (2,2), (3,3)\}$ is the smallest reflexive relation over A:

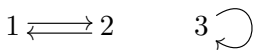


$\rightarrow$ is *irreflexive* if it is not reflexive, i.e. if no element of e is related to itself $\neg\exists a.\ a \rightarrow a$.

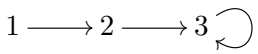
$\rightarrow$ is *transitive* if, whenever $a \rightarrow a'$ and $a' \rightarrow a''$ then $a \rightarrow a''$: $\forall a, a', a''.\ (a \rightarrow a' \wedge a' \rightarrow a'') \implies a \rightarrow a''$. For example, $\rightarrow = \{(1,2), (2,3), (1,3)\}$ is transitive, but without the $1 \rightarrow 3$ edge it wouldn't be. When one draws relations which are known to be transitive, one typically omits all the edges which are implied by transitivity.



$\rightarrow$ is *symmetric* if, whenever $a \rightarrow a'$, then $a' \rightarrow a$: $\forall a, a'.\ a \rightarrow a' \implies a' \rightarrow a$. For example, $\rightarrow = \{(1,2), (2,1), (3,3)\}$ is symmetric, but without the $1 \rightarrow 2$ edge it wouldn't be.



$\rightarrow$ is *antisymmetric* if there are no distinct elements related both ways, or, equivalently, if $a \rightarrow a'$ and $a' \rightarrow a$ then a and a' must be the same element: $\forall a, a'.\ a \rightarrow a' \implies a' \rightarrow a$. Note that this not just the negation of "symmetric". For example, the above is not antisymmetric, but $\rightarrow = \{(1,2), (2,3), (3,3)\}$ is antisymmetric.



$\rightarrow$ is *acyclic* if there are no cycles: $\neg\exists n \geq 1,\ a_1, \ldots a_n.\ a_1 \rightarrow \ldots \rightarrow a_n \rightarrow a_1$. Any transitive relation is acyclic iff it is irreflexive.

A transitive relation $\rightarrow$ is *total* if, for any two distinct elements, they are related one way or the other: $\forall a, a'.\ (a \neq a') \implies (a \rightarrow a' \vee a' \rightarrow a)$.

It's common to use relations that are *preorders*, i.e. reflexive and transitive, or *partial orders*, i.e. reflexive, transitive, and antisymmetric (preorders can contain cycles, while partial orders have no loops except the reflexive cycles for each element). For axiomatic models it's often more convenient to work with irreflexive relations.

One often uses relation symbols such as $\leq$ or $\sqsubseteq$ for binary relations that are orders of one kind or another. In axiomatic relaxed memory models we'll use several different relations over events, including the $\xrightarrow{\mathsf{po}}$ and $\xrightarrow{\mathsf{rf}}$ that we've already seen. Some are transitive while others are not, but we use arrows $\Rightarrow$ for all of them, orienting the arrows to be "forwards" (in whatever sense is appropriate) to ease intuition. These relations shouldn't be confused with the transition relations of the operational models, also written with arrows.

Figure 9.1: Discrete maths notation and terminology: properties of binary relations

All the standard set-theoretic operations, union $\cup$, intersection $\cap$, etc., can be used for relations, viewing them as sets of pairs.

The *identity relation* $\xrightarrow{\text{id}}$ on a set A, also written $[A]$ in the axiomatic memory model literature, is the set of all pairs $a \xrightarrow{\text{id}} a$ for the elements $a \in A$.

Given two binary relations $\xrightarrow{\text{r}}$ and $\xrightarrow{\text{s}}$ over A, their *composition* $\xrightarrow{\text{r}}\xrightarrow{\text{s}}$ has $a \xrightarrow{\text{r}}\xrightarrow{\text{s}} a''$ iff there exists some a' such that $a \xrightarrow{\text{r}} a'$ and $a' \xrightarrow{\text{s}} a''$.

Given two binary relations $\xrightarrow{\text{r}}$ and $\xrightarrow{\text{s}}$ over A, the relation $\xrightarrow{\text{r}} \setminus \xrightarrow{\text{s}}$ has $a \,(\xrightarrow{\text{r}} \setminus \xrightarrow{\text{s}})\, a'$ iff $a \xrightarrow{\text{r}} a'$ and not $a \xrightarrow{\text{s}} a'$.

The *transitive closure* $\rightarrow^+$ of a relation $\rightarrow$ is the union of all the self-compositions $\rightarrow$, $\rightarrow\rightarrow$, $\rightarrow\rightarrow\rightarrow$, etc.

The *reflexive transitive closure* $\rightarrow^*$ of a relation $\rightarrow$ is the union of the identity and all those self-compositions.

The *inverse* $\rightarrow^{-1}$ of a relation $\rightarrow$ has $a \rightarrow^{-1} a'$ iff $a' \rightarrow a$.

Two relations $\xrightarrow{\text{r}}$ and $\xrightarrow{\text{s}}$ are *consistent* if there are no a, a' such that $a \xrightarrow{\text{r}} a'$ and $a' \xrightarrow{\text{s}} a$, or equivalently if $\xrightarrow{\text{r}} \cap \xrightarrow{\text{s}}{}^{-1} = \{\}$.

Figure 9.2: Discrete maths notation and terminology: constructing binary relations

if two threads act on disjoint locations within a cache line, they might write to those simultaneously, then either one or other copy of the cache line would get written back last to memory, overwriting the other.

In a simple hardware implementation, the coherence order over writes to a location is thus just the order in which the hardware threads gain write access to the cache line containing it. For example, one can imagine the cache line ownership changes giving rise to the execution below.

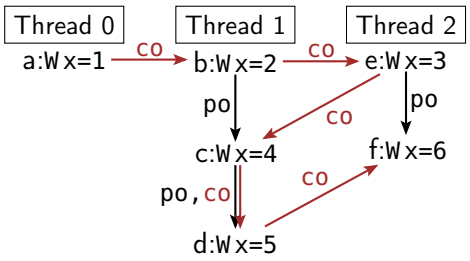
nW x86

Initial state: x=0;

Thread 0	Thread 1	Thread 2

movq $1, (x) //a	movq $2, (x) //b
	movq $4, (x) //c
	movq $5, (x) //d
		movq $3, (x) //e
		movq $6, (x) //f

Final: x=6;



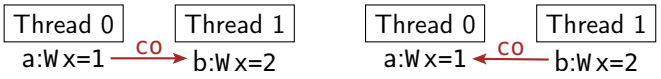
In a more sophisticated implementation, the cache protocol may be more complex, and the choice of coherence order between two writes by different threads might sometimes be resolved at different points in the microarchitecture, e.g. according to which reaches some shared store buffer first. Moreover, in speculative and out-of-order designs, some computation may be done which is later found (by hardware hazard checking) to violate coherence, requiring roll-back and restart: coherence must be enforced at several places in the overall hardware design, not just in the cache protocol.

One way or another, for architectures that guarantee coherence, all hardware implementations have to provide the illusion of (1)–(3). Without this architectural guarantee, one wouldn't even have correct sequential behaviour for independent threads: for memory that is not coherent, software would have to use extra synchronisation for every access to potentially shared cache lines. (That said, mainstream architectures often do include special memory types or special kinds of access that are not coherent without such extra synchronisation, e.g. the x86 "non-temporal" instructions.)

The coherence order is part of the data of an execution graph. For example, a litmus test with just two writes `a` and `b` to the same location, potentially has two execution graphs, one with `a` $\xrightarrow{\text{co}}$ `b` and one with `b` $\xrightarrow{\text{co}}$ `a` (graphs with no coherence edge, or with an edge both ways, are ruled out by the well-formedness conditions):

1+1W x86

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



When writing litmus tests, it's usually best to have a final condition that identifies a unique execution of interest. This test has a final condition `x=2`, so only the left execution is consistent with that (as coherence is from `a` to `b`, the final memory state must have the `Wx=2` overriding the `Wx=1`). Usually one wants the values of writes to each location distinct from each other and from the initial-state value; for ease of reading, we normally choose them in increasing order along the intended coherence order. For tests with at most two writes to each location, with values distinct from each other and from the initial state, the coherence order is determined by the final memory state. Otherwise one might have to add "observer" threads to the test, that read each location multiple times, with final assertions about the values they read, to make the final condition identify a unique execution.

The coherence relation, formally Formally, a coherence relation $\xrightarrow{\text{co}}$ for a candidate pre-execution $(E, \xrightarrow{\text{po}})$ is 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 – as spelt out in previous section.

The from-reads relation We've defined the coherence order as a relation just over writes, not writes and reads, as in (3) above:

3. in any execution, for each location, there is a total order over all the writes to that location, the *coherence order*, and for each thread, the order is consistent with the thread's program-order for its writes and reads to that location, and with reads reading from the most recent write

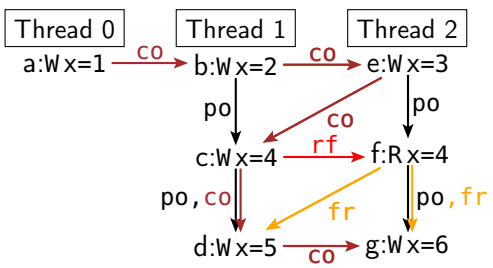
How do we express that it is consistent with program order, and that reads read from the most recent write? It's useful to first define an auxiliary relation. In a coherent memory model, given the reads-from and coherence relations of an execution graph, any read is in some sense "before" all the coherence successors of the write that it reads from. For example, consider a version of the above `nW` with a read `f` of `x` added to Thread 2. A priori, in an execution with the coherence order shown above, that might read from `e`, `c`, or `d`. If it actually reads from `c`, as shown below, we say there is a *from-reads* edge $\xrightarrow{\text{fr}}$ from the read `f` to each of `d` and `g` – all the coherence successors of `c`.

nWR x86

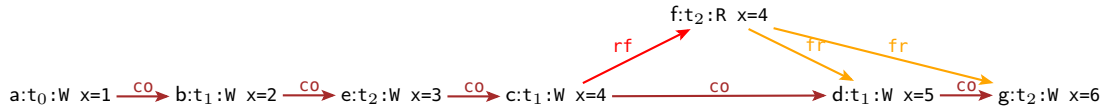
Initial state: `x=0;`		
Thread 0	Thread 1	Thread 2

`movq $1, (x) //a`	`movq $2, (x) //b`
	`movq $4, (x) //c`
	`movq $5, (x) //d`
		`movq $3, (x) //e`
		`movq (x), %rax //f`
		`movq $6, (x) //g`

Final: `2:rax=4; x=6;`		

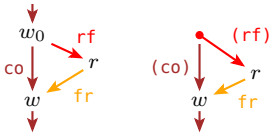


Redrawing the coherence, reads-from, and from-reads edges of that candidate execution to emphasise how the read fits into the coherence order:



This $\xrightarrow{\mathsf{fr}}$ is a derived relation, computed from the other relations of an execution graph rather than part of the data thereof. The idea dates back at least to Ahamad et al. [19]: given the $\xrightarrow{\mathsf{rf}}$ and $\xrightarrow{\mathsf{co}}$ relations of an execution graph, one defines the from-reads relation $\xrightarrow{\mathsf{fr}}$ to relate each read to all $\xrightarrow{\mathsf{co}}$-successors of the write it reads from (or to all writes to its address if it reads from the initial state).

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



or in other words

$$\xrightarrow{\mathsf{fr}} \ = \ (\xrightarrow{\mathsf{rf}}{}^{-1} \xrightarrow{\mathsf{co}}) \cup \{(r, w) \mid \mathrm{iswrite}(w) \wedge \mathrm{addr}(w) = \mathrm{addr}(r) \wedge \neg\exists w_0.\ w_0 \xrightarrow{\mathsf{rf}} r\}$$

This is useful to lift $\xrightarrow{\mathsf{co}}$, which defines an order just over writes, to properties of read/write and read/read pairs. First, it defines a sense in which any read is either after or before any write to the same address: the read either reads from the write or some proper coherence successor of it, or it is from-reads before the write.

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

Proof. The proof is a simple case analysis using the well-formedness conditions and the definition of $\xrightarrow{\mathsf{fr}}$.

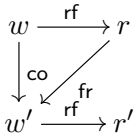
1. Suppose there exists some w' such that $w' \xrightarrow{\mathsf{rf}} r$. As the well-formedness conditions require that $\xrightarrow{\mathsf{co}}$ is total over same-address writes, one of the following holds
   (a) Case $w' = w$. Then $w \xrightarrow{\mathsf{id}} w' \xrightarrow{\mathsf{rf}} r$.
   (b) Case $w \xrightarrow{\mathsf{co}} w'$. Then $w \xrightarrow{\mathsf{co}} w' \xrightarrow{\mathsf{rf}} r$.
   (c) Case $w' \xrightarrow{\mathsf{co}} w$. Then $r \xrightarrow{\mathsf{fr}} w$, by the first clause of the definition of $\xrightarrow{\mathsf{fr}}$.
2. Otherwise, there is no such w'. Then $r \xrightarrow{\mathsf{fr}} w$, by the second clause of the definition of $\xrightarrow{\mathsf{fr}}$.

□

Second, it lifts the coherence relation over writes to a useful relation over pairs of reads that read from them, for reads that do not read from the same source.

Lemma 2 *For any two same-address reads r and r', either they read from the same write (or both from the initial state), or $r \left(\xrightarrow{\mathsf{fr}}\xrightarrow{\mathsf{rf}}\right) r'$, or $r' \left(\xrightarrow{\mathsf{fr}}\xrightarrow{\mathsf{rf}}\right) r$.*

Proof. Suppose they read from two distinct writes w and w' respectively. Then by the well-formedness conditions, either $w \xrightarrow{\mathsf{co}} w'$ or $w' \xrightarrow{\mathsf{co}} w$, respectively, and in each case the conclusion follows from the definition of $\xrightarrow{\mathsf{fr}}$. One also has to consider the case in which one reads from the initial state; then again the conclusion follows from the definition of $\xrightarrow{\mathsf{fr}}$.

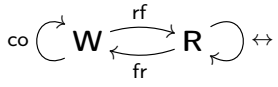


□

Combining these two lemmas, and writing $r \xrightarrow{\leftrightarrow} r'$ iff r and r' are two distinct reads with the same address that either both read from the same write or both from the initial state (an irreflexive, symmetric, and transitive relation), we have:

Lemma 3 *For any two distinct same-address events (reads or writes) e and e', then they are related one way or the other by at least one of $\xrightarrow{\mathsf{rf}}$, $\xrightarrow{\mathsf{co}}$, $\xrightarrow{\mathsf{fr}}$, $(\xrightarrow{\mathsf{co}}\xrightarrow{\mathsf{rf}})$, $(\xrightarrow{\mathsf{fr}}\xrightarrow{\mathsf{rf}})$, and $\xrightarrow{\leftrightarrow}$.*

It's useful when reading this to keep the "types" of the relations $\xrightarrow{\mathsf{rf}}$, $\xrightarrow{\mathsf{co}}$, $\xrightarrow{\mathsf{fr}}$, and $\xrightarrow{\leftrightarrow}$ in mind, as they respectively only relate write/read, write/write, read/write, and read/read pairs:

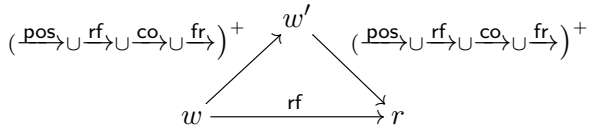


The coherence property We can now express the coherence property axiomatically, as a predicate on candidate executions. So far, there is nothing constraining the reads-from, coherence, and from-reads relations with respect to program order. The coherence order of a candidate execution specifies a total order over writes, and the reads-from and from-reads relations situate reads within that, but nothing yet prevents (for example) a read reading from a program-order-future write on the same thread. The coherence property (3) can be expressed as a property of candidate execution graphs by forbidding certain cycles, requiring:

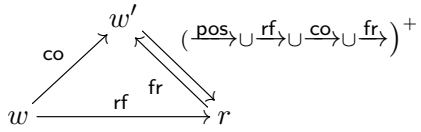
$$\mathrm{acyclic}\big(\xrightarrow{\mathsf{pos}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}}\big)$$

where $\xrightarrow{\mathsf{pos}}$ ("po-same") is the subset of the program-order relation $\xrightarrow{\mathsf{po}}$ that only relates pairs of events which are to the same location. Each of $\xrightarrow{\mathsf{pos}}$, $\xrightarrow{\mathsf{rf}}$, $\xrightarrow{\mathsf{co}}$, and $\xrightarrow{\mathsf{fr}}$ only relate pairs of events which are to the same location, so this is equivalent to a check for each location, that the union of those relations, each restricted to events of that location, is acyclic. (Recall that a relation $\rightarrow$ is acyclic if there is no path $e_1 \rightarrow \ldots \rightarrow e_n \rightarrow e_1$ (for $n \geq 1$), or, equivalently, if $\rightarrow^+$ is irreflexive.)

Our prose above said "each read reads from the most recent write", but it was vague about the order with respect to which that "most recent" was supposed to be. The acyclicity condition checks that each read reads from the most recent write with respect to the transitive closure of that union of relations, $(\xrightarrow{\mathsf{pos}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})^+$. To see this, first note that the $\xrightarrow{\mathsf{rf}}$ relation is included in the union, so no reads-from edge can go from a later write to an earlier read. On the other side, suppose (for a contradiction) that some read reads from an earlier write w that is not the most recent, i.e. there is some distinct intervening w':



The writes w and w' must be to the same address (because all these relations only relate same-address pairs), and so must be related by $\xrightarrow{\mathsf{co}}$ one way or the other (by the well-formedness requirements on $\xrightarrow{\mathsf{co}}$), but then because that union is acyclic, we must have $w \xrightarrow{\mathsf{co}} w'$. The read r thus reads from a coherence predecessor of w', so (by the definition of $\xrightarrow{\mathsf{fr}}$) we have $r \xrightarrow{\mathsf{fr}} w'$.

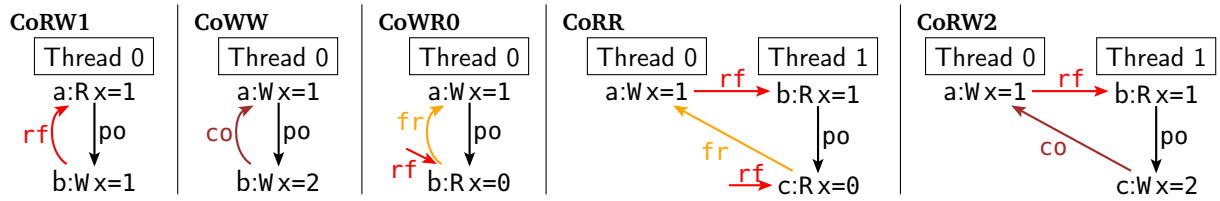


But that contradicts the acyclicity of that union.

Note also that, because the well-formedness conditions require that $\xrightarrow{\text{co}}$ is total, the most recent write for any read is unique (or it reads from the initial state): for any read r there cannot be two distinct writes w, w' that both precede r without any intervening writes and which are unrelated to each other (all with respect to the transitive closure of the union of those relations). This is why we can speak of *the* most recent write.

For any finite acyclic relation, there exists some total order that is consistent with that relation (any topological sort of the relation), and any total order is acyclic, so this acyclicity condition is equivalent to requiring there to be a per-location total order over all reads and writes that is consistent with all those relations, which formalises coherence property (1).

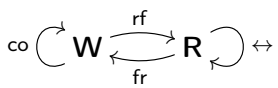
The small-shape characterisation of coherence The coherence property forbids cycles in $(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$. Such cycles might a priori be of any length, and span any number of threads, but in fact the property is equivalent to ruling out the following five specific one- and two-thread coherence shapes, as noted by Alglave [36, §4.2],[23, A.3, p.18].



CoRW1 prevents reading from a program-order-later write. CoWW prevents a write being coherence-after a program-order-later write. CoWR0 prevents a read reading from a coherence-predecessor of a program-order-earlier write (which should hide that coherence-predecessor as far as this read is concerned). CoRR prevents two program-order-related reads reading from distinct writes in the opposite order to their coherence order. CoRW2 prevents a write being coherence-before the write that a program-order predecessor read from.

Theorem 1 *For any candidate execution,* $\text{acyclic}(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$ *iff each of* $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{co}}$, $\xrightarrow{\text{fr}}$, $\xrightarrow{\text{co}}\xrightarrow{\text{rf}}$, *and* $\xrightarrow{\text{fr}}\xrightarrow{\text{rf}}$ *are consistent with* $\xrightarrow{\text{pos}}$.

Proof. The left-to-right implication is immediate from the fact that, for any binary relation R, $R \subseteq R'$ and $\text{acyclic}(R')$ implies $\text{acyclic}(R)$. For the right-to-left implication, suppose on the contrary that there is a cycle in $(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$. By Lemma 3, any edge $e \xrightarrow{\text{pos}} e'$ in that cycle must be in one of $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{co}}$, $\xrightarrow{\text{fr}}$, $(\xrightarrow{\text{co}}\xrightarrow{\text{rf}})$, $(\xrightarrow{\text{fr}}\xrightarrow{\text{rf}})$, and $\xleftrightarrow{}$ or their inverses. In the first five cases, the former (those $\xrightarrow{\text{pos}}$-forwards paths) are included in $(\xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})^+$, and the latter (those $\xrightarrow{\text{pos}}$-backwards paths) are ruled out by the theorem premise, which forbids exactly those shapes. Hence, there must be a cycle in $((\xleftrightarrow{} \cap \xrightarrow{\text{pos}}) \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$. This cannot be a cycle in $(\xleftrightarrow{} \cap \xrightarrow{\text{pos}})$ alone, as $\xrightarrow{\text{pos}}$ is acyclic. Hence by the relation types

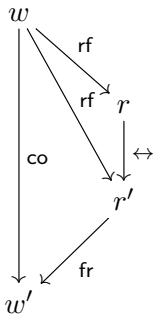


there must be some write event in the cycle, and the cycle from that event must be in

$$\left(\xrightarrow{\text{co}} \cup \left(\xrightarrow{\text{rf}} \left(\xleftrightarrow{}^*\right) \xrightarrow{\text{fr}}\right)^+\right)^+$$

But $(\xrightarrow{\text{rf}} (\xleftrightarrow{}^*) \xrightarrow{\text{fr}}) \subseteq \xrightarrow{\text{co}}$: the $\xleftrightarrow{}$ relation is transitive, and if $w \xrightarrow{\text{rf}} r \xleftrightarrow{} r' \xrightarrow{\text{co}} w'$ then by the definition of $\xleftrightarrow{}$, read r' also reads from w, and by the definition of $\xrightarrow{\text{fr}}$, write w is coherence-

before w'.



Then, because $\xrightarrow{\text{co}}$ is transitive, there must be some cycle in $\xrightarrow{\text{co}}$. But that contradicts the well-formedness conditions.

□

The diagrammatic version is at first sight slightly weaker than this theorem statement, as its CoRR and CoRW2 have their $\xrightarrow{\text{fr}}\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{co}}\xrightarrow{\text{rf}}$ paths via a distinct thread, but if either of those paths go against $\xrightarrow{\text{pos}}$ within a single thread, then at least one of $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{co}}$, and $\xrightarrow{\text{fr}}$ individually must go against $\xrightarrow{\text{pos}}$, and those are all ruled out in either case.

Same-thread (internal) and other-thread (external) relations It's often useful to split the reads-from, coherence, and from-reads relations into their same-thread and other-thread parts (recall that in the x86-TSO operational model, reads from same-thread writes and reads from other-thread writes have quite different ordering properties). Following Alglave et al. [21, 36], these are the *internal* and *external* parts, respectively. We define derived relations:

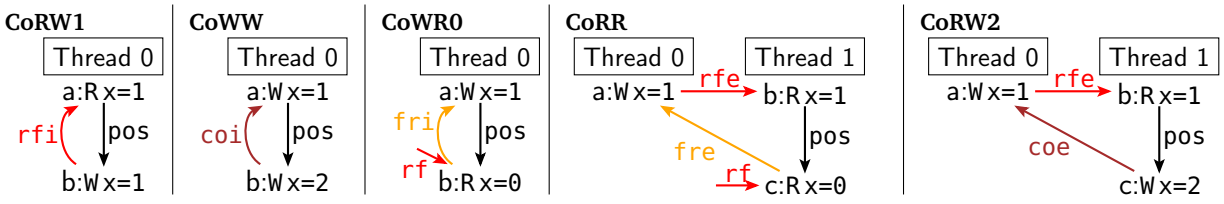
$$e \xrightarrow{\text{rfi}} e' \quad \textit{iff} \quad e \xrightarrow{\text{rf}} e' \wedge \text{thread}(e) = \text{thread}(e')$$
$$e \xrightarrow{\text{rfe}} e' \quad \textit{iff} \quad e \xrightarrow{\text{rf}} e' \wedge \text{thread}(e) \neq \text{thread}(e')$$

and similarly for $\xrightarrow{\text{coi}}$, $\xrightarrow{\text{coe}}$, $\xrightarrow{\text{fri}}$, and $\xrightarrow{\text{fre}}$. The small-shape characterisation of coherence can then be expressed as the combination of checking that each of the three intra-thread (internal) relations are consistent with same-address program order:

- $\text{acyclic}\left(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{rfi}}\right)$
- $\text{acyclic}\left(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{coi}}\right)$
- $\text{acyclic}\left(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{fri}}\right)$

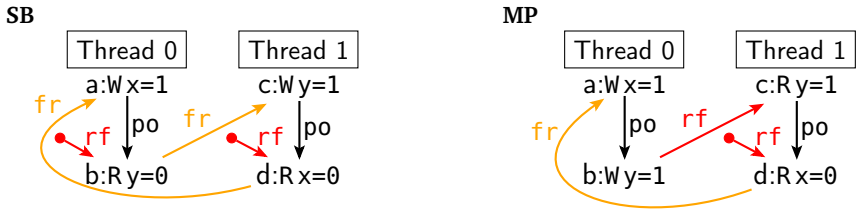
and checking that the two inter-thread (external) relation compositions are consistent with same-address program order:

- $\text{acyclic}\left(\xrightarrow{\text{pos}} \cup \left(\xrightarrow{\text{coe}}\xrightarrow{\text{rfe}}\right)\right)$
- $\text{acyclic}\left(\xrightarrow{\text{pos}} \cup \left(\xrightarrow{\text{fre}}\xrightarrow{\text{rfe}}\right)\right)$



9.4 An axiomatic SC model

Coherence alone is a rather weak property. For example, the candidate executions of SB and MP that we have seen before:



are forbidden in SC, and that MP execution is also forbidden in x86-TSO, but neither violate coherence. Drawing the derived $\xrightarrow{\text{fr}}$ edges in the diagrams, each diagram does has a cycle in the union of $\xrightarrow{\text{po}}$, $\xrightarrow{\text{rf}}$, and $\xrightarrow{\text{fr}}$, but those $\xrightarrow{\text{po}}$ edges are between events to different locations, so these candidates do satisfy the coherence property $\text{acyclic}\big(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}}\big)$.

However, with the infrastructure of candidate executions and the coherence and from-reads relations in place, we can characterise SC (and later other models) in the same style. Define the executions of the SC axiomatic model to be 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 well-formedness properties, that also satisfy

$$\text{acyclic}(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$$

This is like the coherence property except that it includes all of program order $\xrightarrow{\text{po}}$, not just the same-address part of program order $\xrightarrow{\text{pos}}$. The above example candidate executions are therefore forbidden by the SC axiomatic model, as desired.

This acyclicity check for SC can be restated as a combination of the single-thread coherence checks on the internal relations and an inter-thread check that is purely on the external relations:

Theorem 2 *For any candidate execution,* $\text{acyclic}(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$ *iff*

- $\text{acyclic}\big(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{rfi}}\big)$,
- $\text{acyclic}\big(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{coi}}\big)$,
- $\text{acyclic}\big(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{fri}}\big)$, *and*
- $\text{acyclic}(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{coe}} \cup \xrightarrow{\text{fre}})$.

Proof. The left-to-right direction is trivial. For the right-to-left direction, assume the right-hand-side and (for a contradiction) that there is a cycle in $(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$. Any $\xrightarrow{\text{rfi}}$, $\xrightarrow{\text{coi}}$, and $\xrightarrow{\text{fri}}$ edges in that cycle are included in $\xrightarrow{\text{pos}}$ (they each have to be same-address relations, by the well-formedness properties; they have to be same-thread relations, by definition of the internal relations; they have to be either forwards or backwards in $\xrightarrow{\text{po}}$, by the well-formedness properties; and backwards would contradict the right-hand-side). So there is a cycle in $(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{coe}} \cup \xrightarrow{\text{fre}})$. □

This gives a tighter characterisation of the way in which SC is (much) stronger than coherence, by including all of $\xrightarrow{\text{po}}$ in that inter-thread check

$$\text{acyclic}(\xrightarrow{\text{po}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{coe}} \cup \xrightarrow{\text{fre}})$$

rather than the same-address $\xrightarrow{\mathsf{pos}}$ of the inter-thread clause of the analogous characterisation for coherence

$$\mathrm{acyclic}(\xrightarrow{\mathsf{pos}} \cup \xrightarrow{\mathsf{rfe}} \cup \xrightarrow{\mathsf{coe}} \cup \xrightarrow{\mathsf{fre}})$$

(which can be shown by similar reasoning to the above), or the CoRR and CoRW2 of the small-shape characterisation of coherence

$$\mathrm{acyclic}\big(\xrightarrow{\mathsf{pos}} \cup (\xrightarrow{\mathsf{coe}}\xrightarrow{\mathsf{rfe}})\big) \wedge \mathrm{acyclic}\big(\xrightarrow{\mathsf{pos}} \cup (\xrightarrow{\mathsf{fre}}\xrightarrow{\mathsf{rfe}})\big)$$

9.5 Equivalence of the operational and axiomatic SC models

Sections 6.1 and 9.4 define operational and axiomatic models for SC, but do they allow the same behaviour?[1] They define different shapes of behaviour, so one first has to relate those: the operational model defines a set of allowed traces, while the axiomatic model defines a set of allowed candidate executions. Both models are factored out from the thread-local instruction semantics, so we want to state that the two models allow the same behaviour whatever that is. For a candidate execution, this is just about the pre-execution (the events and program order relation), not about the execution witness (the reads-from and coherence relations).

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

- they have the same events

  $$\{e_1, \ldots, e_n\} = E$$

- the order among same-thread events within the trace and the program-order relation of the pre-execution coincide, i.e.

  $$\{(e_i, e_j) \mid i < j \wedge \mathrm{thread}(e_i) = \mathrm{thread}(e_j) \wedge i, j \in 1, \ldots, n\} = \xrightarrow{\mathsf{po}}$$

Then the statement that the two models have the same extensional behaviour is:

Theorem 3 *For any trace T and candidate pre-execution $\langle E, \xrightarrow{\mathsf{po}}\rangle$ that have the same thread-local behaviour, the following are equivalent:*

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

   $$\mathrm{acyclic}(\xrightarrow{\mathsf{po}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})$$

Proof. For the left-to-right direction, given a trace of the operational model, we construct the execution-witness relations of a candidate execution from it, and check the well-formedness conditions and the acyclicity condition of the axiomatic model. For the right-to-left direction, given a candidate execution satisfying the acyclicity condition, we construct a candidate trace as an arbitrary linearisation of the union of those relations, construct a candidate sequence of operational-model states along that trace, and check that each candidate transition is an instance of one of the **WM: Write to memory** and **RM: Read from memory** rules of the operational model.

[1] One can think of relaxed memory models *extensionally*, as just the set of behaviours allowed, or *intensionally*, as their detailed mathematical definitions (taking care not to confuse the *intension* of a model – its internal structure – with the *intention* of its designers – what they intend). The operational and axiomatic SC models of §6.1, 9.4 are, intensionally, two different models, but they should define the same extensional model.

Define the trace order $e < e'$ iff e is before e' in the trace, i.e.

$$e < e' \quad \textit{iff} \quad \exists i, j.\ e = e_i \wedge e' = e_j \wedge i < j$$

The same-thread part of that, $< \cap \{(e, e') \mid \mathrm{thread}(e) = \mathrm{thread}(e')\}$, satisfies the program-order well-formedness properties 1–4 by construction, and it is equal to the $\xrightarrow{\mathsf{po}}$ relation of the candidate pre-execution (which also satisfies those properties) by the same-thread-local-behaviour assumption.

For the left-to-right direction, given a trace, we construct an execution witness by taking the reads-from and coherence orders from the trace order, with each read reading from the most recent write to the same address in the trace order.

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

We first have to check that these satisfy the remaining well-formedness properties of a candidate execution. For $\xrightarrow{\mathsf{rf}}$:

1. for any reads-from edge, it must be from a write to a read, and they must have the same address and value:
   $\forall e, e'.\ e \xrightarrow{\mathsf{rf}} e' \implies \mathrm{iswrite}(e) \wedge \mathrm{isread}(e') \wedge \mathrm{addr}(e) = \mathrm{addr}(e') \wedge \mathrm{value}(e) = \mathrm{value}(e')$

   The first three conjuncts are immediate from the construction of $\xrightarrow{\mathsf{rf}}$. To show that e and e' have the same value, by the definition of SC operational model traces, there are SC model states and transitions

   $$\ldots \xrightarrow{e} m_0 \xrightarrow{e_1} m_1 \ldots \xrightarrow{e_n} m_n \xrightarrow{e'} \ldots$$

   By the **WM: Write to memory** rule of the SC model, the state m_0 after the write e must have the value of that write at its address, $m_0(\mathrm{addr}(e)) = \mathrm{value}(v)$. Then we need to know that in the SC model transitions intervening between e and e', because there are no intervening writes to the same address, the SC memory at $\mathrm{addr}(e)$ remains constant.

   > **Lemma 4** *If $m_0 \xrightarrow{e_1} m_1 \ldots \xrightarrow{e_n} m_n$ then for all addresses a distinct from those of the writes in $e_1, \ldots, e_n$, $m_n(a) = m_0(a)$.*
   >
   > **Proof.** By induction on n, at each step reasoning from the fact that each transition must be an instance of one of the SC operational rules **WM: Write to memory** and **RM: Read from memory**. □

   Finally, by the **RM: Read from memory** rule, $\mathrm{value}(e')$ is that same value.

2. for any e'', there is at most one source of a reads-from edge to it:
   $\forall e, e', e''.\ (e \xrightarrow{\mathsf{rf}} e'' \wedge e' \xrightarrow{\mathsf{rf}} e'') \implies e = e'$

   Suppose $e \xrightarrow{\mathsf{rf}} e''$ and $e' \xrightarrow{\mathsf{rf}} e''$ for distinct e and e'. By the construction of $\xrightarrow{\mathsf{rf}}$, they all have the same address, e and e' are writes, and $e < e''$ and $e' < e''$ in the trace. All distinct events of the trace are related one way or the other by $<$, either $e < e'$ or $e' < e$. The former contradicts the no-intervening-writes clause of the definition of $\xrightarrow{\mathsf{rf}}$ for $e \xrightarrow{\mathsf{rf}} e''$, and the latter contradicts that for $e' \xrightarrow{\mathsf{rf}} e''$.

3. for any read, if there is no reads-from edge to it, its value must be that of the initial state:
   $\forall e.\ (\mathrm{isread}(e) \wedge \neg\exists e'.e' \xrightarrow{\mathsf{rf}} e) \implies \mathrm{value}(e) = m_{\mathrm{init}}(\mathrm{addr}(e))$

   By the construction of $\xrightarrow{\mathsf{rf}}$, if there is is no $\xrightarrow{\mathsf{rf}}$ edge to e then there is no write to the address of e preceding e in the trace (if there were such a write, there would be a $<$-maximal one

without any intervening writes). By Lemma 4, the value in the SC memory for this address is unchanged from the initial state. Finally, by the **RM: Read from memory** rule, $\mathrm{value}(e')$ is that same value.

For $\xrightarrow{\mathsf{co}}$:

1. $\xrightarrow{\mathsf{co}}$ is irreflexive, i.e., no write is related to itself:
   $\forall e.\ \neg(e \xrightarrow{\mathsf{co}} e)$

   By the definition of traces, the events of a trace have distinct IDs, so there is cannot be distinct indices i and j for which $e = e_i$ and $e = e_j$, and $<$ on indices is irreflexive.

2. $\xrightarrow{\mathsf{co}}$ is transitive, i.e., if $e \xrightarrow{\mathsf{co}} e'$ and $e' \xrightarrow{\mathsf{co}} e''$ then $e \xrightarrow{\mathsf{co}} e''$:
   $\forall e, e', e''.\ (e \xrightarrow{\mathsf{co}} e' \wedge e' \xrightarrow{\mathsf{co}} e'') \implies e \xrightarrow{\mathsf{co}} e''$

   If $e \xrightarrow{\mathsf{co}} e'$ and $e' \xrightarrow{\mathsf{co}} e''$ then by the construction of $\xrightarrow{\mathsf{co}}$ they are all writes and have the same address, and by the transitivity of $<$ on indices $e < e''$.

3. $\xrightarrow{\mathsf{co}}$ only relates writes, and only writes to the same address:
   $\forall e, e'.\ e \xrightarrow{\mathsf{co}} e' \implies \mathrm{iswrite}(e) \wedge \mathrm{iswrite}(e') \wedge \mathrm{addr}(e) = \mathrm{addr}(e')$

   By the construction of $\xrightarrow{\mathsf{co}}$.

4. $\xrightarrow{\mathsf{co}}$ relates all pairs of distinct writes to the same address one way or the other:
   $\forall e, e'.\ (e \neq e' \wedge \mathrm{iswrite}(e) \wedge \mathrm{iswrite}(e') \wedge \mathrm{addr}(e) = \mathrm{addr}(e')) \implies e \xrightarrow{\mathsf{co}} e' \vee e' \xrightarrow{\mathsf{co}} e$

   If $e \neq e'$ then either $e < e'$ or $e' < e$ (by the definitions of traces and $<$ again), and in either case the construction of $\xrightarrow{\mathsf{co}}$ gives a $\xrightarrow{\mathsf{co}}$ edge as required.

Now we check that each of $\xrightarrow{\mathsf{po}}$, $\xrightarrow{\mathsf{rf}}$, $\xrightarrow{\mathsf{co}}$, and $\xrightarrow{\mathsf{fr}}$ go forwards in the trace, i.e. that they are each subsets of $<$. This is just about the construction; it doesn't involve the SC operational model. For $\xrightarrow{\mathsf{po}}$, $\xrightarrow{\mathsf{rf}}$, and $\xrightarrow{\mathsf{co}}$ this is immediate from the construction. For $\xrightarrow{\mathsf{fr}}$, suppose $r \xrightarrow{\mathsf{fr}} w$. In case 1 of the definition of $\xrightarrow{\mathsf{fr}}$, for some w_0, $w_0 \xrightarrow{\mathsf{co}} w$ and $w_0 \xrightarrow{\mathsf{rf}} r$. If $r < w$ we are done, so suppose for a contradiction that $w < r$. By the constructions of $\xrightarrow{\mathsf{co}}$ and $\xrightarrow{\mathsf{rf}}$, w_0 is a write, w_0 and w and r have the same address, $w_0 < w$, and $w_0 < r$. But then $w_0 < w < r$, contradicting the no-intervening-write clause of the construction of $\xrightarrow{\mathsf{rf}}$.

In case 2 of the definition of $\xrightarrow{\mathsf{fr}}$, $\mathrm{iswrite}(w)$ and $\mathrm{addr}(w) = \mathrm{addr}(r)$ and $\neg\exists w_0.\ w_0 \xrightarrow{\mathsf{rf}} r$. Suppose for a contradiction that $w < r$. Then there is at least one write (namely w) with the same address as r before it in $<$. Take the last such write, w', then by the construction of $\xrightarrow{\mathsf{rf}}$, we have $w' \xrightarrow{\mathsf{rf}} r$.

Finally, as $\xrightarrow{\mathsf{po}}$, $\xrightarrow{\mathsf{rf}}$, $\xrightarrow{\mathsf{co}}$, and $\xrightarrow{\mathsf{fr}}$ are all contained in $<$, which by construction is acyclic, their union must be acyclic.

For the right-to-left direction of the theorem, given an execution witness $E = \langle \xrightarrow{\mathsf{rf}}, \xrightarrow{\mathsf{co}} \rangle$ such that $\mathrm{acyclic}(\xrightarrow{\mathsf{ob}})$, where $\xrightarrow{\mathsf{ob}} = (\xrightarrow{\mathsf{po}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})$, we construct a candidate trace $[e_1, \ldots, e_n]$ (just a list of events) as an arbitrary linearisation of $\xrightarrow{\mathsf{ob}}$. The events in the candidate trace have unique IDs by the well-formedness properties of the execution witness.

By $\mathrm{acyclic}(\xrightarrow{\mathsf{ob}})$, we know if $e_i \xrightarrow{\mathsf{ob}} e_j$ then $i < j$ (but not the converse).

Construct memory states $m_1, \ldots, m_n$ along that candidate trace by leaving the memory unchanged for each read event and mutating the memory appropriately for each write event, starting with $m_0 = m_{\mathrm{init}}$, and defining

$$m_{i+1} \quad = \quad \begin{cases} m_i & \text{if } \mathrm{isread}(e_i) \\ m_i \oplus (\mathrm{addr}(e_i) \mapsto \mathrm{value}(e_i)) & \text{if } \mathrm{iswrite}(e_i) \end{cases}$$

Now we check that the candidate trace actually is a trace of the SC operational model, i.e that there is a sequence of transitions

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

For the write events, each transition is an instance of the **WM: Write to memory** rule by construction. For the read events, say r_j at index j, for that transition to be an instance of the **RM: Read from memory** rule, we need $m_{j-1}(\mathrm{addr}(r_j)) = \mathrm{value}(r_j)$.

By the construction of the m_i,

$$m_{j-1}(\mathrm{addr}(r_j)) \quad = \quad \begin{cases} \mathrm{value}(e_i) & \text{where } i \text{ is the largest } i < j \text{ such that } \mathrm{iswrite}(e_i) \\ & \quad \text{and } \mathrm{addr}(e_i) = \mathrm{addr}(r_j), \text{ if there is one, or} \\ m_0(\mathrm{addr}(r_j)) & \text{otherwise} \end{cases}$$

In the first case, write w_i for e_i. We know by the well-formedness of the candidate execution and Lemma 1 that either $w_i \xrightarrow{\mathsf{co}} w_k \xrightarrow{\mathsf{rf}} r_j$ for some k, $w_i \xrightarrow{\mathsf{rf}} r_j$, or $r_j \xrightarrow{\mathsf{fr}} w_i$. If $w_i \xrightarrow{\mathsf{co}} w_k \xrightarrow{\mathsf{rf}} r_j$, then $i < k < j$, which would contradict the "largest". If $r_j \xrightarrow{\mathsf{fr}} w_i$, because $\xrightarrow{\mathsf{fr}}$ is included in the acyclic $\xrightarrow{\mathsf{ob}}$ relation, we have j<i, contradicting i<j. Hence $w_i \xrightarrow{\mathsf{rf}} r_j$, so by well-formedness they have the same value.

In the second case, there is no $i < j$ such that $\mathrm{iswrite}(e_i)$ and $\mathrm{addr}(e_i) = \mathrm{addr}(r_j)$, so because the trace is a linearisation of $\xrightarrow{\mathsf{ob}}$, there is no $w \xrightarrow{\mathsf{ob}} r_j$ such that $\mathrm{addr}(w) = \mathrm{addr}(r_j)$, so there is no $w \xrightarrow{\mathsf{rf}} r_j$, so by the candidate-execution $\xrightarrow{\mathsf{rf}}$ well-formedness, $\mathrm{value}(r_j) = m_0(\mathrm{addr}(r_j))$. □

9.6 Exercises

Exercise 9.1 *Exercise: show that acyclic po co rf fr is equivalent to tot compatible with po and rf*

Exercise 9.2 *Explain why this execution is allowed/forbidden a:Wx42-po->b:Wy1-po-rf->c:Ry1-po->d:Rx0 (MP is interesting because it has a non-trivial fr)*
IRIW
more!!!

Exercise 9.3 *Define a memory model that executes programs in a round-robin way (you may assume that there are only two threads, and that all threads have the same number of instructions)*

Exercise 9.4 *Find a program that has an execution that is allowed in SC, but not in round-robin*

Exercise 9.5 *Define a "sequentialising" memory model that executes programs one thread after another (but with no specific order on which thread). Find a program that has an execution in SC but not in this sequentialising memory model. How does it compare to the round-robin model?*

Chapter 10

x86-TSO, axiomatically

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

- each write has two events, a $w = (\mathsf{W}\, x{=}v)$ when it is enqueued into the local write buffer, and a $\mathsf{D}_w\, x{=}v$ when it is dequeued to memory
- each read has one event, but it can arise in two ways, either reading from the local write buffer or, if there is no write to the same address in that, reading from memory

These distinctions are 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. keep one-event-per-access candidate executions, expressing the conditions that define allowed behaviour just on those.

Perhaps surprisingly, (2) turns out to be possible

As previously mentioned, the first axiomatic model for x86-TSO, by Owens, Sarkar, and Sewell [101, 103, 120], was based on the SPARCv8 TSO specification [3]. The axiomatic model we describe here is the TSO model [31] in the "herd" style of Alglave and Maranget [36]. Both use just a single event per access. We initially ignore LOCK'd instructions and MFENCEs, returning to them later in the chapter. As before, we start with coherence.

10.1 Coherence in x86-TSO

The ultimate coherence order over writes to the same address in an x86-TSO operational model execution is the order in which they reach memory: the trace order of their $\mathsf{D}_w\, x{=}v$ dequeue events. Note that this might not match the enqueue order: two threads could enqueue writes to the same address in one order, and then they could be dequeued from their write buffers in the opposite order.

Reads that read from memory are in the right place in the operational-model trace w.r.t. those dequeue events, after the dequeue of their $\xrightarrow{\mathsf{rf}}$-predecessor and before the dequeues of their $\xrightarrow{\mathsf{fr}}$-successors. For any such read, one knows there is no write to the same address in the local buffer (which would become an $\xrightarrow{\mathsf{fr}}$-successor), by the buffer-empty condition in the x86-TSO **RM** rule. Without that condition, such a write would end up coherence-after all writes that have already reached memory, including the one the read reads from – a coherence violation.

Read events that read from buffers will be *before* the corresponding dequeue event (of the write that they read from) in the trace, but, for each thread, the write enqueue events and reads from buffers are done in program order, and each read from a buffer will be after the $\mathsf{W}\, x{=}v$ enqueue event they read from, and before any $\xrightarrow{\mathsf{po}}$-later enqueue event. The ordering among

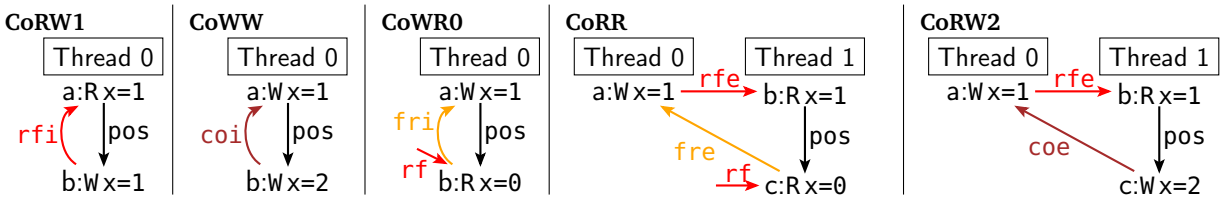
same-thread write enqueues ends up included in the coherence order by the FIFO nature of the buffer: two $\xrightarrow{\text{po}}$-related writes are dequeued in the same order.

As far as the coherence check

$$\text{acyclic}(\xrightarrow{\text{pos}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{co}} \cup \xrightarrow{\text{fr}})$$

goes, therefore, one can think of the reads that read from each buffered write as located in coherence order immediately after the dequeue event of that write, rather than at their positions in the operational trace.

It's useful to see how the x86-TSO operational model forbids each of the small coherence shapes of Theorem 1.



CoRW1: In the operational model, a read can only see a same-thread write that is $\xrightarrow{\text{pos}}$-before it, either in the thread's buffer or in memory.

CoWW: The buffers are FIFO, so two $\xrightarrow{\text{pos}}$ writes are dequeued in $\xrightarrow{\text{pos}}$-order.

CoWR0: The b reads from a coherence-predecessor ::W x=0 (which could be on any thread) of a.

- Case c is on the same thread as b. c must be $\xrightarrow{\text{po}}$-before a, as writes are enqueued in $\xrightarrow{\text{po}}$ 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 $\text{no_pending}(m.B(\mathtt{t}_0), \mathtt{x})$ condition of **RM**.
  - Case b reads from buffer, by **RB**. Then a must still precede c in the buffer. This violates the $\text{no_pending}(b_1, \mathtt{x})$ 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 $\text{no_pending}(m.B(\mathtt{t}_0), \mathtt{x})$ condition of **RM**.

CoRR: The dequeue of a must be before b reads, and b reads before c does. The c reads from a coherence-predecessor ::W x=0 (which could be on any thread) of a, so d must be dequeued before a. But then c would have read from a.

CoRW2: The dequeue of a must be before b reads, and b reads before c is enqueued, which is before c is dequeued. Then c is coherence-before a, so c must be dequeued before a is. But this would be a cycle in machine execution time.

10.2 Local ordering and the external relations

Recall that the axiomatic model for SC could be factored into intra-thread coherence conditions and an inter-thread condition that only involved the external relations (though among accesses to potentially different locations).

x86-TSO has an analogous characterisation. First, say a machine trace T is *complete* if it has no non-dequeued write, and for any write enqueue event w in such, write $D(w)$ for the unique corresponding dequeue event.

For same-thread events in a complete operational machine trace, with trace order $<$:

- If $w \xrightarrow{\mathsf{po}} w'$ then w is dequeued before w' (so $D(w) < D(w')$).
- If $r \xrightarrow{\mathsf{po}} r'$ then r reads before r' reads (so $r < r'$).
- If $r \xrightarrow{\mathsf{po}} w$ then r reads before w is enqueued, and hence before w is dequeued (so $r < D(w)$).
- If $w \xrightarrow{\mathsf{po}} r$, 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 $\xrightarrow{\mathsf{rfe}}$, $\xrightarrow{\mathsf{coe}}$, and $\xrightarrow{\mathsf{fre}}$), all of program order $\xrightarrow{\mathsf{po}}$ except the write-to-read edges is preserved. We can express this by requiring

$$\mathrm{acyclic}\left(\left(\xrightarrow{\mathsf{po}} \setminus (\mathsf{W} \times \mathsf{R})\right) \cup \xrightarrow{\mathsf{rfe}} \cup \xrightarrow{\mathsf{coe}} \cup \xrightarrow{\mathsf{fre}}\right)$$

where W and R are the sets of all write and read events respectively, or equivalently

$$\mathrm{acyclic}\left(\left(\xrightarrow{\mathsf{po}} \setminus \left([\mathsf{W}](\xrightarrow{\mathsf{po}})[\mathsf{R}]\right)\right) \cup \xrightarrow{\mathsf{rfe}} \cup \xrightarrow{\mathsf{coe}} \cup \xrightarrow{\mathsf{fre}}\right)$$

(recalling that $[A]$ is another notation for the identity relation on a set A).

10.3 An x86-TSO axiomatic model, without `MFENCE` and LOCK'd instructions

Define the executions of the x86-TSO axiomatic model (still ignoring read-modify-write instructions and `MFENCE`s) to be all candidate executions over the write and read events of the x86-TSO machine grammar (not including its dequeue events), i.e. all pairs of

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

that satisfy the well-formedness properties, that also satisfy the coherence check:

$$\mathrm{acyclic}(\xrightarrow{\mathsf{pos}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})$$

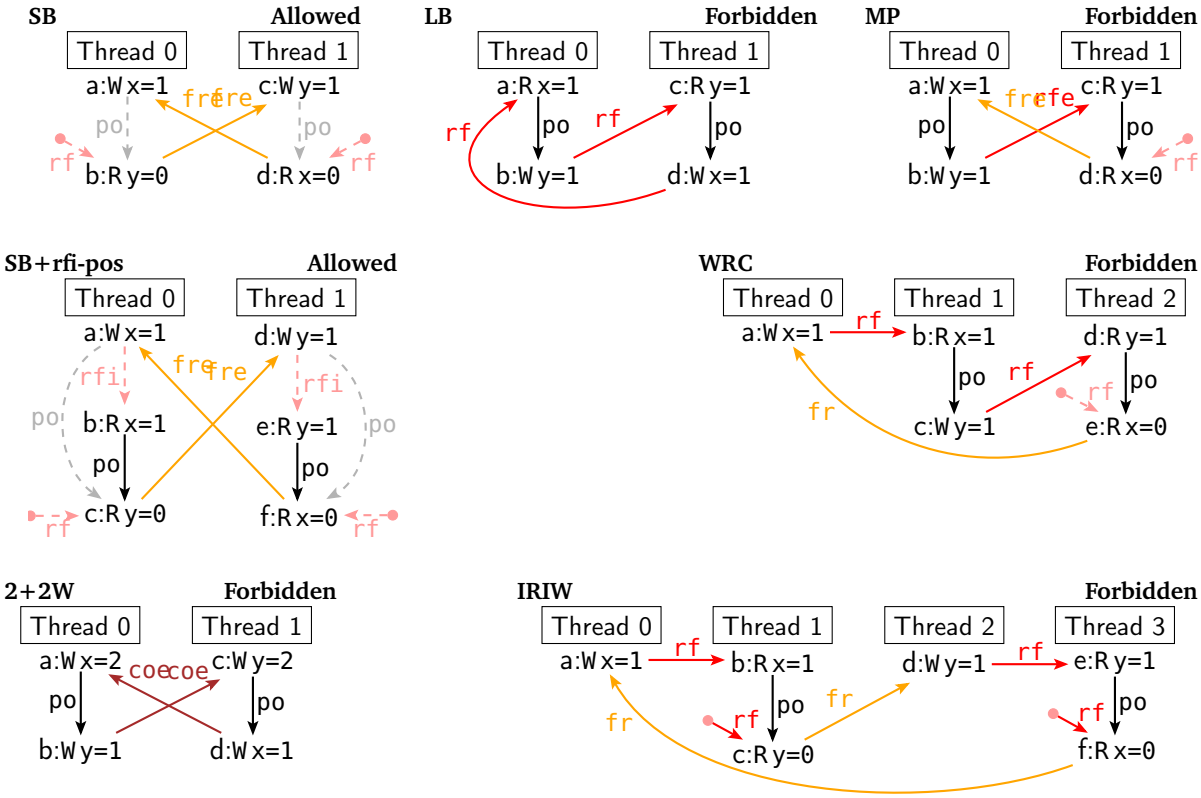
and the above external check:

$$\mathrm{acyclic}\left(\left(\xrightarrow{\mathsf{po}} \setminus \left([\mathsf{W}](\xrightarrow{\mathsf{po}})[\mathsf{R}]\right)\right) \cup \xrightarrow{\mathsf{rfe}} \cup \xrightarrow{\mathsf{coe}} \cup \xrightarrow{\mathsf{fre}}\right)$$

10.4 x86-TSO axiomatic examples

The small coherence shapes, single-address examples, are all forbidden by the coherence check, exactly as before.

For multiple-address examples (leaving aside those involving fences and LOCK'd instructions for the moment), we can draw the candidate executions and highlight the edges that are included in the above external check, showing the others dashed and lightened, e.g. for some of the key examples from previous chapters:



For each of these, there is a cycle in those edges iff it is forbidden in the x86-TSO operational model.

10.5 Equivalence of the operational and axiomatic x86-TSO models, without `MFENCE` and LOCK'd instructions

The statement of the equivalence of the x86-TSO operational and axiomatic models is similar to that for SC, except that in relating the two notions of execution – operational traces and axiomatic candidate pre-executions – we have to account for the fact that the former has dequeue events and the latter does not.

Say an x86-TSO trace $T = [e_1, \ldots, e_n]$ (with trace order $<$) and an x86-TSO candidate pre-execution $\langle E, \xrightarrow{\text{po}} \rangle$ have the same thread-local behaviour if

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

Then:

Theorem 4 *For any candidate pre-execution $\langle E, \xrightarrow{\mathsf{po}}\rangle$, 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 $X = \langle \xrightarrow{\mathsf{rf}}, \xrightarrow{\mathsf{co}}\rangle$ for $\langle E, \xrightarrow{\mathsf{po}}\rangle$ such that* $\mathrm{acyclic}(\xrightarrow{\mathsf{pos}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}})$ *and* $\mathrm{acyclic}\left(\left(\xrightarrow{\mathsf{po}} \setminus \left([\mathsf{W}](\xrightarrow{\mathsf{po}})[\mathsf{R}]\right)\right) \cup \xrightarrow{\mathsf{rfe}} \cup \xrightarrow{\mathsf{coe}} \cup \xrightarrow{\mathsf{fre}}\right)$.

This has been proved by Durbaba in the Isabelle mechanised theorem prover [67]. We won't give a full proof here, but the basic idea is:

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 $\xrightarrow{\mathsf{ob}}$, mapping write events onto dequeue transitions and adding write enqueue transitions as early as possible, then check the operational machine admits it.

10.6 Relational algebra Cat notation for axiomatic model definitions

In general, axiomatic models could be defined using arbitrary discrete mathematics, but models in the "herd" style of Alglave and Maranget [36] can be expressed just as the conjunction of checks that various derived relations are acyclic or empty, and those derived relations can be defined using standard relational-algebra operations on binary relations, rather than pointwise definitions involving their elements. The "Cat" language [36] provides a concise concrete syntax for such relational algebra, and is used by various software tools, including `herd7` and (in a variant) `isla-axiomatic`.

The Cat notation for relations is very similar to what we have used so far (which is conventional discrete maths notation except that we have used arrows for all relations, to get a tight correspondence between the maths and diagrams), except (a) one omits the arrows for relations, e.g. writing just `po` for the program-order relation instead of $\xrightarrow{\mathsf{po}}$, and (b) the operations on relations are written as below.

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

Then a Cat file can contain a sequence of definitions of derived relations, e.g.

```
let pos = po & loc
```

defining `pos` to be the intersection of the program order relation `po` with the same-location relation `loc`, and acyclicity and emptiness checks, e.g.

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

checking $\mathrm{acyclic}\left(\xrightarrow{\mathsf{pos}} \cup \xrightarrow{\mathsf{rf}} \cup \xrightarrow{\mathsf{co}} \cup \xrightarrow{\mathsf{fr}}\right)$, and also naming this check `internal`.

With this, the x86-TSO axiomatic we have so far could be expressed as follows, introducing new names `obs`, `lob`, and `ob` for some of the derived relations.

```
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 *)

 (* expanding the above, ob = po \ ([W];po;[R]) | rfe | coe | fre *)

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

10.7 An x86-TSO axiomatic model, with LOCK'd instructions and `MFENCE`

Our operational x86-TSO model covered the x86 `MFENCE` barrier and LOCK'd read-modify-write instructions. To extend the above axiomatic model correspondingly, we have to add three things.

For `MFENCE`, operationally this restores order between writes and reads by waiting for the thread-local write buffer to drain before allowing subsequent instructions to go ahead. Axiomatically, that can be captured by including `MFENCE` events in candidate executions, and adding all write/read pairs separated in program order by an `MFENCE` to the locally-ordered-before relation:

```
(* Locally-ordered-before *)
let lob = po \ ([W]; po; [R])
        | [W]; po; [MFENCE]; po; [R] (* W/R pairs separated by an MFENCE *)
        | ...
```

where `[MFENCE]` is the identity relation on the set of all `MFENCE` events.

For atomic read-modify-write instructions, the LOCK'd instructions, operationally these were modelled with `L` and `U` transitions at the start and end of each such instruction. They have two effects: those transitions each require the local write buffer to be empty, thereby restoring order to write/read pairs around (or involving) the LOCK'd instruction; and while one thread is executing a LOCK'd instruction, any **RB: Read from buffer**, **RM: Read from memory**, and **DM: Dequeue write from write buffer to memory** by other threads are blocked.

The first of those can be captured axiomatically by adding additional write/read pairs to locally-ordered-before, where at least one of each pair comes from such a LOCK'd instruction (writing `X` for the set of all such events):

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

The second effect can be captured by ruling out all writes which are *coherence intervening* between the read and write of the LOCK'd instruction, requiring:

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

where `rmw` is a relation that relates the read and write of each LOCK'd read-modify-write instruction.

That completes this x86-TSO axiomatic model, lightly adapted for presentation from [31]:

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

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

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

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

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

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

(* External check *)
acyclic ob
```

10.8 Equivalence of the operational and axiomatic x86-TSO models

Durbaba's mechanised proof [67] of the equivalence of x86-TSO operational and axiomatic models, described in §10.5, also covers `MFENCE` and LOCK'd instructions. The definition of same thread-local behaviour has to be extended, to require:

- **Events:** The set of events of the pre-execution must be the same as the set of events of the trace when restricted to Read, Write and Barrier events only (excluding Propagates, Locks and Unlocks, but including MFENCEs).
- **Program-Order:** Two events in the same thread must appear in the same order in the trace as they appear in the pre-execution's po relation.
- **Read-modify-write:** Each RMW Read/Write pair r `rmw` w must be inside a pair of Lock and Unlock events, and every Read/Write pair inside a pair of Lock and Unlock events must be a RMW pair.
- **RMW Locks only:** Each Lock event in the trace must be followed by a Read, then a Write, and then a Propagate of that Write and finally an Unlock event.

then the theorem statement is essentially the same: that for any trace and any candidate execution with the same thread-local behaviour, the trace is allowed by the operational model iff the candidate execution is allowed by the axiomatic model.

10.9 Exercises

Exercise 10.1 *Show that SB is allowed by the axiomatic model of TSO.*

Exercise 10.2 *Show why Peterson and Dekker's are forbidden with the axiomatic model when suitably mfenced, and allowed otherwise.*

Exercise 10.3 *Show why the following MP variant is forbidden:*
W x 37; W x 42; W y 1 || R y 1; R x 37

Exercise 10.4 *Make a variant of SB that uses a single location. Is a weak outcome still allowed by TSO?*

Exercise 10.5 *Is the following test allowed?*
W x 1; W x 2; R y 1 || W y 1; W y 2; R x 1

Exercise 10.6 *Show that a compilation scheme with mfence in between every read/write enforces SC.*

Exercise 10.7 *Show that if a location is only accessed by one thread, then the fences between accesses to other locations and this one are not needed.*

Exercise 10.8 *Propose a more efficient scheme.*

Exercise 10.9 *Design an axiomatic model for PSO (see the exercise in the operational TSO chapter 7).*

Exercise 10.10 *Design an axiomatic model for a variant of PSO in which addresses equal modulo 2^n share a buffer.*

Exercise 10.11 *Show that the following algorithm implements mutual exclusion:*
W x 1; r1 = R x; if (r1 = 2) enter critical section || W x 2; r2 = R x; if (r2 = 1) enter critical section
where the critical sections can be represented by Wz1, and mutual exclusion means that there is at most one write of 1 to z

Chapter 11

Making the axiomatic models executable as a test oracle: the Herd and Isla-axiomatic tools

Historically, most formal semantic models, including much (though not all) previous work on relaxed memory models, have been expressed as pen-and-paper or LaTeX mathematics, not executable in any way. That makes it hard and error-prone to explore what behaviours they allow on examples, to compare that against the behaviour of real-world implementations, and to compare variant models against each other. Building on our previous experience developing semantics for real-world network protocols [118, 136, 51, 52, 49, 111, 50], we therefore emphasised from our first work in this area the need for tool support that makes relaxed-memory models executable as test oracles. Chapter 8 already briefly described the RMEM tool that lets one explore various operational models. Those models are expressed as nondeterministic transition systems, so for small examples one can compute the set of all model-allowed behaviour by an exhaustive backtracking search, computing allowed executions incrementally.

For axiomatic models, the problem is rather different: these models are expressed as predicates over candidate execution graphs: the memory events (and relations over them) of candidate *complete* executions; one can't easily incrementally build the allowed executions. Instead, the obvious strategy is to first enumerate all the candidate pre-executions (the E and `po`) that are consistent with the instruction semantics for each thread in isolation, then for each enumerate all the candidate execution witnesses (pairs of `rf` and `co` relations over those events that satisfy the well-formedness properties), then filter those by the consistency predicate of the model (e.g. the coherence and external acyclicity checks of the x86-TSO axiomatic model), to give the set of allowed candidate executions. This strategy scales badly in the size of the test[1], but because most litmus tests are very small, it often remains viable, and the close correspondence between the tool implementation and the intended mathematical semantics helps provide confidence in the tool.

A priori, any memory read in the thread-local semantics could read an arbitrary value (e.g. from $\{0, \ldots, 2^{64} - 1\}$), so to make this practical, even for small litmus-test examples, one either has to restrict the value domain to a very small set (e.g. just $\{0, 1\}$, or treat the thread-local instruction behaviour *symbolically*. For example, for a dynamic instance of an x86 quad-word 64-bit `movq (y),%rax` instruction, reading the memory value at location `y` into register `rax`, a thread-local symbolic semantics creates a fresh symbolic variable, say v, records that after this instruction `rax` contains v, and adds a read event R `y`=v to the candidate pre-execution, `po`-after all previous events by this thread. For an x86 `incq (y)` instruction, such a semantics creates a fresh symbolic variable, say v', and adds a read event R `y`=v' and a write event W `y`=$v' + 1$,

[1] For example, for tests with no control-flow choices, if there are L locations, each with W write events, and a total of R read events, there are a priori $(W!^L) \cdot (R^{(L \cdot W)+1})$ choices of the coherence orders and reads-from relations.

involving the symbolic expression $v' + 1$. For each candidate reads-from relation one adds symbolic constraints, for each `rf` edge, that the values of the writes and reads are equal. The tool then has to check for each candidate execution whether these constraints are all satisfiable, and also check the acyclicity conditions of the axiomatic model.

Several tools along these lines have been developed. We focus here on those we use in this text, deferring discussion of other related work to Part **??**. There are many interesting design decisions and differences between such tools, including:

- the range of architectures they support;
- the range of features they support (e.g. mixed-size, instruction-fetch, virtual memory, etc.);
- the range and completeness of the instruction-set semantics they support, e.g. just enough for simple litmus tests, or for typical concurrent algorithms, or a complete ISA;
- what format of litmus tests they support (it's highly desirable for all tools to support the same format, or have automated translators, so that tools and models can be experimentally compared);
- whether the instruction-set semantics has to be custom-written for this tool or can be re-used from an established and validated definition;
- the tension between a naive rendering of the mathematics into executable code, which is desirable for confidence that the tool does what is intended, and optimisations that may be necessary for sufficient performance;
- the relationship between the tool implementation and any mechanised mathematics version of the intended semantics;
- the choice of whether to use an ad hoc constraint solver or an external SMT solver;
- whether the axiomatic model is specified in code in the tool source, or in some external logical language, or in some external Cat or Cat-like relational algebra definition; and
- the user interface design, including command-line and/or web versions, preferably with good graph drawing for candidate executions.

The *memevents* tool, used first in development of the x86-CC and x86-TSO axiomatic models [114, 20, 101, 120] was developed from 2007 to 2014, initially by Susmit Sarkar and Peter Sewell, following discussions with Francesco Zappa Nardelli, and then also by Jade Alglave (in joint development of prototype IBM Power axiomatic models), and by Luc Maranget. Gabriel Kerneis contributed work on Power ISA semantics.

A 2013 re-engineering of memevents by Luc Maranget evolved into the *herd* tool [29] by Alglave et al. [36], and used initially for axiomatic models for SC, TSO, release-acquire C++, and IBM Power. This introduced the Cat language [25], for defining axiomatic models with conditions on derived relations defined using relational algebra (rather than the pointwise set-theoretic definitions used earlier), that we briefly described in §10.6.

Herd was extended by various authors for GPU concurrency [24], C11 and OpenCL [44], the Linux kernel memory model [32], and x86 non-writeback memory, non-temporal instructions, and persistent memory [108].

Later development led to the current (at the time of writing) *herdtools7* version of herd [30] by Alglave and Maranget. This is used by Alglave et al. within Arm for their axiomatic concurrency model specification [27]. Herd has a web interface at `http://diy.inria.fr/www/`.

The *isla-axiomatic* tool [38, 40, 41] has been developed since 2019, principally by Alasdair Armstrong, Brian Cambell, Ben Simner, and Thibaut Pérami. This uses the complete Arm-A

and RISC-V instruction semantics, expressed in the Sail ISA definition language of Armstrong et al. [39]. These are, respectively, automatically translated from the authoritative definition in their ASL language, and hand-written and adopted by RISC-V International as their reference formal model. Isla-axiomatic thus escapes the previous reliance on custom hand-crafted ISA models for relaxed-memory axiomatic exploration tools. To make these large definitions usable (the Arm-A ISA specification is several hundred thousand lines), isla-axiomatic uses the *isla* symbolic evaluation engine for Sail, and the Z3 SMT solver [63]. It combines the constraints from this intra-instruction execution with those of the axiomatic memory model, in a similar strategy to that of Alglave et al. [28]. It includes the litmus-test final condition constraints; isla-axiomatic checks whether there exists some execution that satisfies those, rather than computing the set of all allowed executions – in other words, it makes models executable as a test oracle, but not exhaustively executable. Isla-axiomatic has been used in the development of axiomatic models for Arm-A instruction-fetch and virtual memory, by Ben Simner et al. [125, 124], exploiting the fact that the Arm-A ISA specification includes all the details of the virtual memory configuration and address translation page-table walks. Isla-axiomatic has a web interface at `https://isla-axiomatic.cl.cam.ac.uk/`. However, it does not currently support x86. We'll describe it briefly here and return in more detail for the more relaxed Arm-A and RISC-V models later.

For completeness we mention here also two analogous tools for axiomatic models for programming-language concurrency. The early *cppmem* tool [45, 43] was developed around 2010–2012 by Mark Batty, Scott Owens, Jean Pichon-Pharabod, Susmit Sarkar, and Peter Sewell. It supports the C++11 relaxed memory model as formalised by Batty et al. [43], expressed in the Lem higher-order logic specification language [97, 96], together with an ad hoc symbolic semantics for a small fragment of C. It has a web interface at `http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/`.

The 2019 *Cerberus-BMC* [87, 88] tool, by Stella Lau, Victor Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell, adapts the Cerberus semantics by Memarian [94, 95], for a large fragment of C, to generate SMT constraints – analogous to, though actually preceding, the isla-axiomatic switch to full ISA semantics, but at the C level. Cerberus BMC combines those thread-local constraints with various relaxed models (substantial fragments of the C11, RC11, and Linux Kernel models) specified in a Cat-like relational algebra language, using an external SMT solver. It has a web interface at `http://cerberus.cl.cam.ac.uk/bmc.html`.

x86 Litmus tests The web interfaces for RMEM, herd, and isla-axiomatic each provide a library of litmus tests. These are not necessarily identical, though the individual tests largely follow a common naming scheme, and same-named tests should be essentially equivalent.

There's also a useful set of x86 litmus tests at `https://github.com/litmus-tests/litmus-tests-x86`, produced by Shaked Flur and Luc Maranget. For example, tests such as `SB`, `SB+mfences`, and `MP` can be found in the `tests/non-mixed-size/BASIC_2_THREAD` directory there, and various coherence tests in `tests/non-mixed-size/CO`.

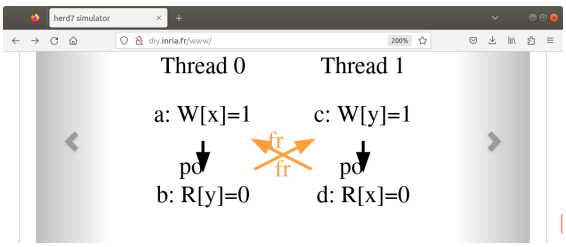
We'll discuss how tests can be automatically generated in the following chapters.

The herd web interface To use the herd web interface, go to `http://diy.inria.fr/www/`, select x86 from the central blue drop-down menu, and either select one of the basic x86 tests from the test-name drop-down menu on the right, or paste in some other test. The default x86-TSO Cat model is shown below in the interface – this is phrased slightly differently to the one we saw in §10.7, but it should be equivalent to that. Then one can run the tool by pressing the right-triangle at the bottom. This shows the textual output of herd, e.g.:

```
Test SB Allowed
States 4
```

```
0:EAX=0; 1:EAX=0;
0:EAX=0; 1:EAX=1;
0:EAX=1; 1:EAX=0;
0:EAX=1; 1:EAX=1;
Ok
Witnesses
Positive: 1 Negative: 3
Condition exists (0:EAX=0 / 1:EAX=0)
Observation SB Sometimes 1 3
Time SB 0.10
Hash=edd4722437d708675ed921e7607e77f0
```

This shows the list of final states allowed by the model, with the `Positive: 1` indicating that one of those satisfies the final condition of the test. Herd also renders the allowed candidate executions (there called "event structures"), e.g.



The herd command-line interface If you've installed `herdtools7` as in §1.5, cloned the above `litmus-tests-x86` repository into a directory `REPO_DIRECTORY`, and have the x86-TSO Cat file from §10.7 in a file`x86-tso.cat`, then:

```
herd7 -cat x86-tso.cat \
  $(REPO_DIRECTORY)/litmus-tests-x86/tests/non-mixed-size/BASIC_2_THREAD/SB.litmus
```

runs herd with that model on the basic SB test, producing a similar histogram and result. It differs slightly as this is a 64-bit version of the SB test.

```
herd7 -cat x86-tso.cat \
  $(REPO_DIRECTORY)/litmus-tests-x86/tests/non-mixed-size/BASIC_2_THREAD/SB.litmus
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;
Ok
Witnesses
Positive: 1 Negative: 3
Condition exists (0:rax=0 /\ 1:rax=0)
Observation SB Sometimes 1 3
Time SB 0.01
Hash=ac0b1f983ea53dfb4bc1a5cf3e493028
```

The isla-axiomatic web interface To use the isla-axiomatic web interface (for Arm-A and RISC-V), shown in Fig. 11.1, go to `https://isla-axiomatic.cl.cam.ac.uk/`. The upper left pane shows a litmus test; the middle pane shows the axiomatic model in use, in a mild variant

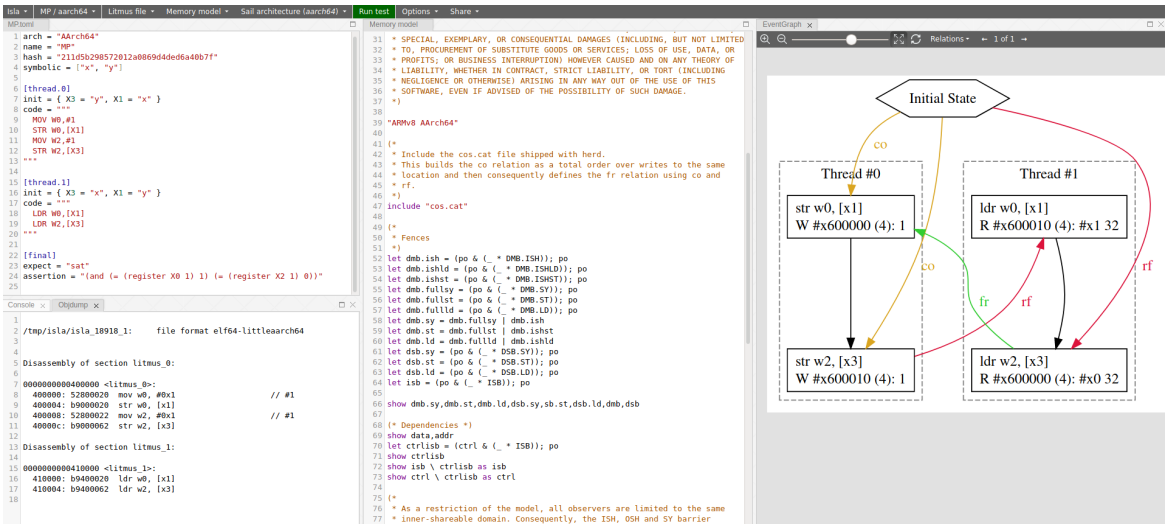


Figure 11.1: Isla-axiomatic web interface

of the Cat syntax used by herd; and the right pane shows candidate executions. One can select a new litmus test from the library in the "Litmus file" drop-down and select from a list of axiomatic models in the "Memory model" drop-down; both can be hand-edited. One can also choose from a range of ISA models in the "Sail architecture" drop-down. Isla-axiomatic uses a different concrete syntax for litmus tests, in a TOML format (suffix `.toml`) rather than the original ad hoc `.litmus` format, but old tests can be translated to the new format with a standalone tool in the repository. Because it uses complete ISA definitions, including decoding, Isla-axiomatic actually checks machine-code litmus tests rather than the assembly-like syntax used by RMEM and herd; it uses a conventional assembler to translate the assembly instructions of the test source into machine code before checking that. Clicking on "Run test" checks whether there are candidate executions satisfying the axiomatic model, the instruction semantics, and the final constraint of the test. If there are, it shows them in the right-hand pane; the "Relations" drop-down lets one toggle display of various relations.

The isla-axiomatic command-line interface To install and run isla-axiomatic locally (which will be necessary for larger examples), follow the repository instructions at `https://github.com/rems-project/isla`.

11.1 Exercises

Exercise 11.1 *Modify x86tso.cat to implement PSO, and identify an execution of a program that is allowed in PSO but forbidden in TSO.*

Chapter 12

Running tests on hardware: Litmus

In §1.5 we introduced the *litmus* tool, for experimentally running tests on hardware processor implementations, and Chapter 2 built on experimental results for various x86 tests. Much early work on relaxed memory models was based on discussions of microarchitecture or on vendor documentation, and the experimentally observable behaviour of hardware has not always been clear. Notable work that took a more empirical approach include among others that of Collier [57, 59], with his 1993 ARCHTEST [58], and Adir et al. [16]; other related work will be described in Part **??**.

The litmus tool was inspired in part by these and (as for model evaluation tooling) by the *experimental semantics* approach we used in modelling network protocols [50]. ARCHTEST comprises a small number of hard-coded experiments to probe specific aspects, e.g. with one thread writing an increasing sequence of values to a location while another reads them and checks they are non-decreasing, to probe one aspect of coherence. In contrast to this, we wanted a generic tool that would support testing of any litmus test. The challenge here is one of (more-or-less) black-box testing of a complex system: the observable behaviour of a multiprocessor for a concurrent example depends on many aspects of its microarchitectural design and of its dynamic internal state, including e.g. the exact timing between execution of different hardware threads, the states of the cache protocol, store buffers, and pipelines for each hardware thread, contention for specific execution units, and so on. In white-box testing of a specific hardware design during its development, one might have knowledge of the design for all those, and visibility of them during simulation, though it may still be challenging to drive the system into a good range of its internal states. For external testing of production processors, one does not have that visibility.

Our basic approach, dating back to initial experiments by Sarkar and Sewell in 2007, is to transform a litmus test, say that operates over memory locations `x` and `y`, into an indexed version, accessing elements of an array for `x` and an array for `y`. Executing many copies of the test, iterating over the array indices in a randomised order, randomising over choices of the assignment of threads of the test to hardware threads, and varying the precise synchronisation of the thread starts of each instance, usefully stresses the hardware implementation. The first version of the litmus tool was developed principally by Thomas Braibant and Francesco Zappa Nardelli from 2007–2008, with contributions from Alglave, Sarkar, and Sewell. This took arbitrary litmus tests as input, in a format much like those we present here, rather than the hard-coded litmus tests of our initial experiments.

Luc Maranget contributed from late 2008 onwards with substantial re-engineering, initially to harmonise the front-end with the memevents tool, and with another refactoring from 2013 onwards leading eventually to the current herdtools7 version [30]. This long development has tuned the test harness and added many options, and it has been used in many papers and in both IBM and Arm. An early version was described in [22], but one should consult the current documentation `https://diy.inria.fr/doc/litmus.html` for up-to-date details.

Experimental testing might be done for several subtly-different reasons:

- investigation of the behaviour of existing hardware implementations, to inform the construction of models;
- checking that a model is sound, with respect to the behaviour of existing hardware implementations; and
- checking that a hardware implementation is correct with respect to a model.

However, one always has to be aware of the limitations. Experimental testing can demonstrate that a specific hardware implementation can exhibit some particular behaviour (modulo any bugs in the test harness, of course), but it cannot demonstrate conclusively that a hardware implementation cannot exhibit a behaviour: if a behaviour isn't observed, it could just be that the test harness does not drive the implementation into an internal state that would exhibit it. Some interesting behaviours occur only very rarely, even when using carefully tuned configurations of the litmus test harness, so one might need to run tests for many iterations to observe them reliably – even up to 10^{12} iterations – and this can be time-consuming, especially for large batches of tests. Most fundamentally, experimental testing alone cannot distinguish between an intentionally allowed behaviour or a hardware bug: it does not tell one what the architectural intent is – though it can and does inform discussion of what the archtiectural intent should be.

Running a batch of tests on hardware using `litmus` In a simple usage one might run `litmus` on a single test, as we showed in §1.5, but typically one wants to run it on a batch of tests, and to tune the options to increase the likelihood of observing interesting results. For example, the directories of the `https://github.com/litmus-tests/litmus-tests-x86/tree/main/tests/non-mixed-size` repository of x86 litmus tests include `@all` files that list (hierarchically) all the tests they contain. One can run all of those in (for example) `BASIC_2_THREAD` with:

```
litmus7 -r 100 BASIC_2_THREAD/@all > run-hw.log
```

This runs each of those tests 10^7 times, logging to `run-hw.log`. For serious testing, one should increase that by 10–1000 and tune the parameters to the hardware at hand, and typically will be using many more tests. The generated log contains, for each test, the histogram of observed final states. It also records whether the identified final-state condition was observed or not. For example:

```
Test SB Allowed
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
```

The `Allowed` on the first line and the `0`ks in the output should be ignored – what matters is the histogram, showing how often each final state was observed, and the `Observation` line, showing whether (and how often) a final state satisfying the final condition of the test was observed. The $*$ in the histogram identifies those states.

Chapter 13

Test families and test generation: Diy

The early literature on relaxed memory sometimes illustrates specific points with a few small examples, like those we have seen. Collier recalls [56] that early engineers "made up all the test cases they could think of and then they ran those cases past their designs", without general theory, although there "were a lot of people, in industry and in academia, who produced such tests", e.g. what we know as SB by R. M. Smith.

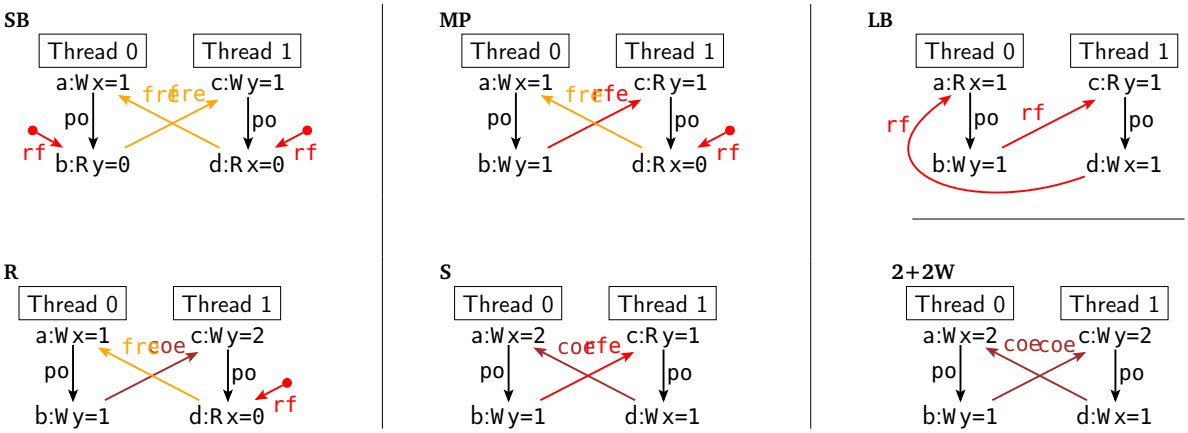
As mentioned already, the tests in the early literature are typically paper examples only, to illustrate some proposed microarchitecture or formal model, without discussion of testing on hardware or execution in model evaluation tools. To the best of our knowledge, such tests were first referred to as litmus tests in the 1992 Alpha Architecture Reference Manual [129], which included 11 tests. Other architecture manuals also include small numbers of tests. For example, the IBM Power manual historically included two [10, Book II, §1.7.1]. Intel added 10 tests to its x86 manual around 2008 [61], including some of those we saw in Chapter 2 (they were not in the Nov. 2006 version [60]). Arm produced a note containing around 10 user-concurrency tests (and more for cache and TLB maintenance) in 2008 [37].

Adir et al. [16] speak of a larger set of "about 40 litmus tests that cover all rules of the [PowerPC] model" (8 included in that paper), which they also used in the verification of a processor design. Our early work was in the same vein, taking existing tests from the literature and adding a few more, e.g. the 24 tests used for x86-TSO validation [101, 103, 120].

However, all these are ad hoc collections, of a relatively small number of hand-written interesting tests. Can one organise tests more systematically, and/or generate them automatically, and ideally so so in a way that is in some sense complete? The axiomatic model structure suggests ways to do these.

13.1 Organising tests

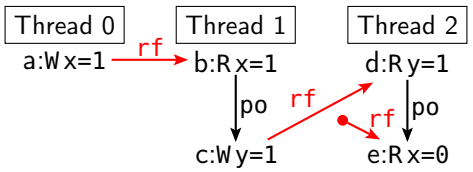
Tests can be useful in two overlapping ways: for exploring the allowed behaviour of models, and for experimentally investigating what behaviour a hardware implementation exhibits. For both, one is especially interested in tests with some non-SC execution, as the SC executions are allowed in any reasonable model. The axiomatic-model characterisation of SC (§9) thus suggests a way to organise tests systematically: one can look at tests with a cycle in $(\texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr})$, and especially at tests with a minimal such cycle, in which (for example) there are no adjacent `po` edges. The small-shape characterisation of coherence of §9.3 (Page 81) gives such cycles with two or three edges (all such cycles can involve only one location). For cycles with four edges, there are just six shapes that have a pair of different-location accesses on each of two threads.



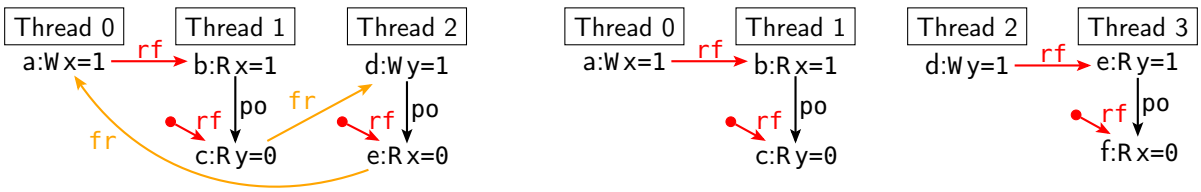
We have already seen SB, MP, and LB. Tests R and S can be thought of as variants of SB and MP respectively, but with a coherence edge from a write to a write in place of a from-reads edge from a read to a write (which means there is coherence edge from the write that that read reads from to the latter write). These are similar in some ways: on weaker architectures, each pair of variants needs the same additional synchronisation to restore SC. Test 2+2W involves just coherence and program order. Naming tests consistently and concisely turns out to be challenging, especially as more are used over time, involving more features. Several of those test names are rather arbitrary, but they are now well-enough established that it is best to keep using them.

For each test shape, one can then systematically consider the family of all tests that have the same shape but with some additional synchronisation. For example, tests MP+mfence+po and MP+po+mfence are strengthenings of MP with an x86 mfence between the two Thread 0 access or Thread 1 accesses respectively (and just program order on the other thread), and MP+mfences is the strengthening with an mfence on both. Many more kinds of strengthening will be used later, especially for more relaxed architectures.

For each of those 4-edge tests, one can consider possible 5-edge extensions in which a first write on one or both threads is moved to a new thread and replaced by a read from it. These are also similar to their base tests in that, on weaker architectures, each needs the same additional synchronisation to restore SC. For MP, this gives the WRC test (§2.7):



For SB, doing this for one write gives the RWC test [55], while doing it for both gives IRIW (§2.6):



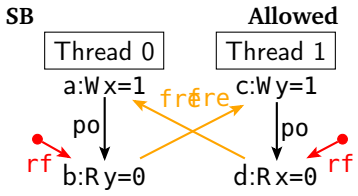
Tests S, R, and 2+2W can be extended similarly (R in two distinct ways), but we do not detail them here.

Moving to tests with three threads, three shared locations, and two reads or writes in each thread, there are 11 families, of which several are interesting for more relaxed architectures.

All this suggests grouping tests in a way loosely reminiscent of the periodic table, which we'll come back to later.

13.2 Generating single tests from cycles

Hand-writing tests is sometimes necessary, but if they are simple non-SC cycles, it is convenient and less error-prone to automatically generate them. The *diyone7* tool from Alglave and Maranget's `herdtools7` tool suite [30] can do this, for multiple architectures. For example, `SB`:



is essentially the cycle `a po b fr c po d fr a`. Annotating those `fr` edges to indicate that they are inter-thread (external, `e`), and annotating those `po` edges with additional data to indicate that they are between events to different locations (`d`) and from a write to a read event (`WR`), one has the cycle `Fre PodWR Fre PodWR` in the input language of the tool. Then:

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

generates the syntactic litmus test, including the assembly for each thread and a final condition that identifies the (unique here) candidate execution that contains that cycle:

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

For more details, see the documentation: `http://diy.inria.fr/doc/gen.html`, and [21, 35, 34]. In brief, program order edges have the syntax `Po(s|d)(R|W)(R|W)`, where `s` or `d` indicates that the two events are to the same or different location(s), and `R` or `W` indicates an event is a read or a write. For a program order edge separated by an x86 `mfence`, one can write `MFenceRR`, `MfenceRW`, etc. Reads-from, coherence, and from-reads edges must specify whether they are internal or external, e.g. `Rfi` and `Rfe`, and `Fri` and `Fre`. For historical reasons coherence edges are written `Wsi` and `Wse` here (for "write serialisation").

13.3 Generating families of tests

Inspired by the *critical cycles* (minimal non-SC cycles) of Shasha and Snir [121], and by their work on axiomatic models expressed using acyclicity conditions, Alglave and Maranget developed the *diy* tool [21, 35, 34] (and the diyone above), to generate families of tests with interesting relaxations. To use diy, one specifies a number of processors, a maximum cycle length, a set of edge kinds that are assumed "safe", i.e., expected to be respected by hardware, or included in the main acyclicity condition of a model, and a set (possibly empty) of "relaxed" edges that

might not be. The tool then generates (roughly) all the cycles up to that length, each of which contains just one "relaxed" edge (or none if that set was empty), and then generates a concrete litmus test for each such cycle.

For example, using 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
```

Running diy:

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

generates the six 2-thread 2-location critical-cycle tests above. Again, see the documentation: http://diy.inria.fr/doc/gen.html, and [21, 35, 34] for more details.

To try to observe some putative relaxation (some edge that we think should not be in `ob`), remove it from the `-safe` list and add it to `-relax`, then `diy7` will by default generate cycles of exactly one relaxed edge and some safe edges (from http://diy.inria.fr/doc/gen.html#sec52):

x86-rfi.conf

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

x86-podwr.conf

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

This test generation approach, combined with the modelling, model evaluation, and experimental testing infrastructure of the previous chapters, has been very effective for model development. It has identified a number of previously-unexpected but not erroneous hardware behaviours, and some bugs in, or differences between, models. It has also identified a number of errata in production processor implementations.

It lets one quickly generate substantial collections of interesting tests, and also to do so more-or-less uniformly for multiple architectures (modulo the inescapable differences between their fence instructions and suchlike). However, one should note that the generated tests are of a particular character. They cover interesting cycles of edges, but the assembly instructions of each test are uniformly generated from the given edges, rather than designed to thoroughly exercise the microarchitectural features of hardware processor implementations. For example, an important edge for more relaxed models is a data dependency via registers, from one memory read to the value of a later memory write. The generator will produce essentially identical assembly for every data dependency edge, without attempting (e.g.) to create contention for pipeline arithmetic units. There is surely scope for more refined test generation that does that, along the lines of hardware test generators.

That said, there is a trade-off. For model development, one wants to establish a library of reusable tests, that can be run on multiple hardware implementations, and in multiple models. But as one adds more synchronisation features, more threads, and more events per thread, the number of tests quickly explodes, and running each test on hardware for a sufficiently large

number of iterations becomes time-consuming. It's thus important to generate only "good" tests, as far as possible.

Chapter 14

Validating the model: why should one believe it?

The x86-TSO model[1] aims to be a good specification, for the concurrency aspects within its scope, of an existing abstraction at the heart of computing, the x86 architecture. The model is thus interesting and important to the extent that it does capture that contingent but central reality, rather than as a purely theoretical definition. Hence, one has to ask: why (and to what extent) should one believe it?

We described the properties a good architecture specification should have in §1.6. To establish reasonable confidence that x86-TSO has them, we have to *validate* the model in various ways. It is not possible, even in principle, to attain complete certainty for all these: some are mathematical facts, but several are not, or are in principle but not practically so. This perspective is similar to that adopted in our earlier work on network protocols [50], which was similarly an attempt to develop a good post facto specification for an existing pervasive abstraction, though the technical and social context for architectural relaxed memory are interestingly different to that for network protocols; we'll discuss that later.

14.1 Sound with respect to existing hardware: experimental validation

One can gain a useful level of confidence that a model is sound with respect to existing hardware processor implementations, i.e. that the model admits all the behaviour that those implementations exhibit, by experimental testing. This relies on:

1. making the model executable as a test oracle, e.g. with the RMEM tool for operational models (§8) and the herd and isla-axiomatic tools for axiomatic models (§11);
2. a test harness such as litmus for running tests on hardware implementations (§12), and
3. a good suite of tests, typically some hand-written and some automatically generated, e.g. by the diy tool (§13), that exercise the model and (with the test harness) exercise hardware implementations.

For example (on a small scale) one might generate the basic x86 4-edge tests with the diy configuration of §13:

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

[1] Given the proof of equivalence between the operational and axiomatic models, we can now speak of *the* x86-TSO model as the extensional model that they both define.

run them on hardware from the command line of an x86 machine:

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

compute the allowed behaviour in the operational model for them, running RMEM in exhaustive mode (and also fix up the case of register names in the log file to match herd):

```
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
```

and compute the allowed behaviour in the axiomatic model for them, using herd:

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

One can then compare the resulting hardware and model log files uing the *mcompare* tool of the herdtools7 suite:

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

The == entries show that, for these tests, the operational and axiomatic models both allow exactly the same behaviour as the hardware exhibits (in this case, the specific x86 processor used for that run of litmus), and allow the same as each other. The mcompare7 tool also has options `-pos <file>` and `-neg <file>` to output just the positive and negative differences. Normally we would check test hashes for safety, without the `-nohash` option, but at present they have temporarily diverged between the tools.

For each litmus test, any final state might be *observed* or *not observed* in experimental testing of some specific hardware implementation, and *allowed* or *forbidden* by some specific model.

In comparing models and implementations, one always has to be aware that experimental testing of implementations is not exhaustive: if one observes some particular final state, that provides definite evidence (assuming the test harness is correct), but if a state is not observed, that might be just because the test harness is not aggressive enough, or that it did not happen to

generate the right conditions to observe that state in the executed runs. For the models, on the other hand, the RMEM and herd tools compute the set of all model-allowed behaviours of each litmus test, and isla-axiomatic computing whether the specified interesting final state is allowed or not.

In comparing a model and a hardware implementation, one thus has four cases (for each litmus test and each final state):

model	experiment	conclusion
allowed	observed	ok
allowed	not observed	ok, but model is looser than this hardware, or testing is not aggressive
forbidden	observed	model is not sound w.r.t. this hardware, or the hardware has a bug
forbidden	not observed	ok

One can compare models and implementations either by looking at the sets of all allowed/observed behaviours for each test, or just whether the identified final state is allowed/observed. In general, the former is more discriminating, but some tools conveniently support only the latter, and the two almost always coincide.

The same approach can also be used to experimentally compare models against each other.

Experimental testing and validation are very important in model development and in model validation (developing confidence), but experiment is not enough by itself:

- the architectural intent is typically looser than any specific hardware design, so there is a danger of over-fitting;
- one can't always determine whether a strange observed behaviour is a hardware bug or not without asking the relevant architects – ultimately, that is their call; and
- it is hard to know whether there might be undiscovered phenomena that are not covered by any of one's current set of tests, especially if those (either automatically generated or hand-written) share some specific properties.

As mentioned in §1.6, in principle one would like to establish soundness of an architectural model with respect to current hardware mathematically, by proving that it is a sound implementation of the relevant hardware implementations (at the RTL level of abstraction). That is still out of reach.

14.2 Sound with respect to future hardware; loose enough to permit future microarchitectural innovation

These are hard in principle to assess, especially without discussion with leading processor design teams for the architecture in question. For x86, our understanding is that the existing body of code that implicitly relies on TSO-like behaviour effectively constraints future architectural change to be no more relaxed: that the de facto standards of the code assumptions and observable hardware behaviour are essentially tight against each other.

14.3 Opaque with respect to hardware implementation detail

Modern x86 processors are highly sophisticated designs with out-of-order and speculative execution, but the x86-TSO programmers model exposes very little of this – just the existence of store buffers.

14.4 Complete with respect to hardware

As noted in §1.6, this is not a goal for real architecture specifications. The x86-TSO model is not complete with respect to hardware in many ways, notably that it allows unbounded-size write buffers and unbounded differences in execution speed between hardware threads, while any specific hardware implementation will have some concrete bounds. This looseness is desirable for an architecture specification: one would not want software to be written on the assumption that the write buffers are (say) of size 16, as that would not be helpful for normal concurrent idioms, it could inspire exotic coding, and it would not be robust in the face of likely future microarchitectural change.

14.5 Strong enough for software

As we noted, this is hard to assess as one would like, against the corpus of software. For x86-TSO, there are theoretical results that give confidence of particular kinds. Owens [102] showed a generalised *data-race freedom* result, that for any program that does not exhibit certain kinds of races ("triangular races") in SC executions, every x86-TSO execution gives the same result as some SC execution. Building on this, he showed the correctness of an x86 spinlock and that any properly-locked program using it has the mutual exclusion property and SC behaviour with locks that one would expect.

Batty et al. [43] proved the correctness of the proposed compilation scheme from the C/C++11 programming language concurrency model down to x86-TSO.

Ševčík et al. [119] proved correctness of a compiler, extending CompCert [90], from a dialect of C with TSO-like concurrency to x86-TSO.

14.6 Precise and unambiguous

The original definitions of x86-TSO [101] were mechanised in a proof assistant (in HOL4 [**?**]), which is as precise and unambigous as a definition can be. The definitions presented here are in paper maths, backed up by (but without a precise connection to) executable operational definitions in RMEM and axiomatic definitions in the Cat language of Herd. The axiomatic model here is in a somewhat different style to the original.

14.7 Clear

Whether the x86-TSO operational and axiomatic models are clear is perhaps best for the reader to decide. We believe that the operational model, with its per-hardware-thread TSO write buffers, provides a decent mental model for thinking about program execution, and the axiomatic model a more concise definition of the allowed behaviour.

14.8 Executable as a test oracle

The RMEM and Herd tools make the model executable as a test oracle for small litmus tests.

14.9 Incrementally executable

The operational version of the model is incrementally executable. For testing larger software, one would want better tooling, such as a more efficient architecturally complete emulator.

14.10 Mathematically validated

The original x86-TSO paper [101, 103] exercised the definitions mathematically by proving, largely in the HOL4 mechanised proof assistant, the equivalence of operational and axiomatic versions of the model. That gives substantial assurance in the details of both. The paper used a separate OCaml implementation of the axiomatic model in the memevents tool to check the allowed behaviour of litmus tests, but it also defined an algorithmic version of the axiomatic-model consistent execution predicate in HOL4, proved that equivalent to the primary definition, and extracted code that could also run in memevents to check, for positive tests, that any execution that the OCaml implementation permits is really permitted.

As mentioned above, Owens [102] showed a generalised *data-race freedom* result, Batty et al. [43] proved the correctness of the proposed compilation scheme from the C/C++11 programming language concurrency model down to x86-TSO, and Ševčík et al. [119] proved correctness of a compiler with x86-TSO concurrency. There has been much other work about or above the model, including for example a rely-guarantee proof system by Ridge [110], and a separation logic by Sieczkowski et al. [122]. However, different usages often require rephrasing or transcribing the model into another form, e.g. into a different proof assistant, so there is no single definitive version which all of these boost assurance of. Durbaba [67] proved in Isabelle the equivalence of operational and axiomatic models based on the paper maths versions we describe here.

14.11 Authoritative

x86-TSO is in a particular position here. It was developed by an academic group without detailed interaction with the industry vendors of x86 processors (Intel, AMD, and VIA), but it provided a clear specification of the allowed concurrency behaviour at a time when the vendor documentation did not (and, to a lesser extent still does not). Of course one can and should compare in detail with what the vendor documentation does say, as described in Chapter 3. x86-TSO has not been incorporated into any vendor documentation, but it nonetheless does, to our understanding, capture the de facto standard.

Each other model we'll describe has it's own particular (and sometimes peculiar) status, in different ways.

14.12 Accurately capturing the architectural intent

Without detailed discussion with the architects, this is difficult to assess. The authors of [101, 103] attempted to establish such discussion at that time, with only limited success.

14.13 Consistency with the de facto standard

Our experimental testing, discussion in the community, and the limited discussion we had with x86 vendors, all strongly suggest that x86-TSO does accurately capture the de facto standard.

Part II

Arm-A, IBM Power, and RISC-V

Chapter 15

Introducing Arm-A, IBM Power, and RISC-V relaxed concurrency

The Arm-A, IBM Power, and RISC-V architectures are all substantially more relaxed than x86. They are broadly similar to each other in their relaxed-concurrency behaviour, though not identical. They are also more complex than that of x86, with a more delicate boundary between the behaviours that the architectures allow and forbid, and with additional mechanisms to strengthen the default relaxed behaviour where necessary. In this Part II, we describe the relaxed phenomena they exhibit and the operational and axiomatic models that have been developed to precisely specify them. As before, we focus just on the "user" architecture: normal loads and stores, and the synchronisation mechanisms provided to manage them, as used in concurrent algorithms (in either user or systems code). We defer discussion of the relaxed "systems" behaviour, such as instruction fetch, virtual memory, and exceptions.

We begin in this first chapter with context on the three architectures and an informal discussion of how their relaxed behaviour arises from microarchitectural optimisations, sketching an abstract microarchitectural view that will be useful for understanding the detailed phenomena. We also describe the litmus tests and candidate executions that we'll use.

Chapter 16 discusses the relaxed phenomena in detail, and the following chapters introduce and define the models that capture this.

15.1 Architectures and Implementations

For context, we start with some general background on these three architectures, their implementations as hardware processors, and their roles in the industry.

Each architecture specification exists in many versions, evolving over time as features are added and the specification is refined. Many of these changes are unrelated to the relaxed concurrency behaviour of the aspects we discuss, but sometimes there are new features that are specific to concurrency, e.g. the addition of new atomic instructions, or changes to the specified concurrency behaviour that are important here, e.g. the 2017 major shift by Arm to multicopy-atomic behaviour for Arm-A, and other more minor changes since.

Each architecture specification version may also have various optional features, such as the Arm-A FEAT_xxx features, and the RISC-V extensions. Subsets of these may be required in particular architecture versions (for Arm) or for particular profiles (for RISC-V). Sometimes the status of an optional feature can be discovered introspectively, e.g. by code querying a special register.

Each processor design should conform to some particular architecture version, with some particular set of optional features. The long timeline, of architecture development, processor and system-on-chip design, product design, and product lifespan, means that many processor designs and many architecture versions co-exist at any one time.

15.1.1 Arm-A

Arm develop architecture specifications and processor designs. They do not manufacture and sell processors themselves; instead, they license the specifications and designs to a large ecosystem of other vendors. These Arm partners either license specific Arm-designed processor cores to integrate into their own system-on-chips, or, for architecture licensees, design their own Arm-architecture cores to build into their own system-on-chips. Arm-architecture processors are dominant in mobile devices, and increasingly important in servers.

Architecture versions Arm define three main families of architectures. The *application-profile (A-profile)* architectures, which we focus on here, target high-performance systems such as phone main processors and datacentre servers. Arm also define microcontroller-profile (M-profile) and real-time profile (R-profile) architectures, along with the Morello research program architecture that adds hardware-capability security, but we do not cover any of those.

The A-profile architecture is defined in a pdf document, updated around twice per year:

Date	Document version	Architecture version(s)
2013	A.a	Armv8.0-A (first non-confidential beta)
...		
2016	A.k	Armv8.0-A (early-access release, EAC)
2017	B.a [12]	Armv8.1-A (EAC), Armv8.2-A (Beta) (simplification to MCA)
...		
2022	H.a	Armv8.8-A and Armv9.3-A (incorporating SVE)
...		
2024	K.a	Armv8.9-A and Armv9.4-A (incorporating MPAM and SME)
2024	L.a [14]	Armv9.5-A

Armv8-A introduced Arm's *A64* 64-bit instruction set (executing in its *AArch64* execution state). ARMv7 and earlier versions are still in use. For brevity, we write Arm-A to refer to all Armv8-A and Armv9-A architecture versions. We consider only the AArch64 execution state, not AArch32.

Processor implementations There are many Arm-A architecture processor implementations. Some are based on core designs by Arm, licensed to other vendors to incorporate into system-on-chips (SoCs) that they design and build. For example, the Arm-designed Cortex-A78 core, which implements the Armv8.2-A architecture and some extensions, has been used in the Samsung Exynos 2100 SoC, the MediaTek Dimensity 1200 and 8000 series, the NVIDIA DPU, and the HiSilicon Kirin 9000s[1]. Others are designed by Arm architecture partners, for example the Armv9.2-A architecture core, designed by Apple, of the Apple M4 SoC. The Wikipedia pages `https://en.wikipedia.org/wiki/List_of_ARM_processors` and `https://en.wikipedia.org/wiki/Comparison_of_ARMv8-A_cores` maintain lists of Arm-A cores of various architecture versions.

Documented instruction-set architecture The Arm-A instruction semantics is defined in their ASL specification language, made machine-readable by Reid et al. [109]. The ASL definitions of instruction behaviour are included in the Arm architecture manual and are also available in XML. The Arm-A ASL can be automatically translated into Sail [39, 47], which supports translation into theorem provers and other tools.

[1] `https://en.wikipedia.org/wiki/ARM_Cortex-A78`

Documented relaxed concurrency architecture Early versions of the Arm specification included prose memory model descriptions, broadly similar to those of IBM Power, that did not clearly define what behaviour was allowed. From March 2017 [12] (version B.a of the documentation, for versions Armv8.1-A and Armv8.2-A of the architecture), this was replaced by a prose version of an axiomatic model [64]. This was developed principally by Deacon, then at Arm, expressed in the Cat style of Alglave et al.'s herd tool [36, 29]. It was co-developed with a corresponding operational model, principally by Flur, Pulte, Sarkar, and Sewell, that was proved equivalent by Pulte [106, 105]. Recent versions of the Arm specification include a prose description [14, B.2] automatically generated from a successor to this axiomatic model, by Alglave, Nikoleris, and Khyzha [26, 65, 91]. All of this has been in discussion with Grisenthwaite, Arm Chief Architect.

Further academic work by Armstrong et al. integrated a version of those axiomatic models with the full Arm-A instruction semantics [39, 40, 41], and work by Simner et al. developed extensions for some systems aspects: instruction fetch [125], virtual memory [124], and exceptions [123], again in collaboration with Grisenthwaite.

15.1.2 IBM Power

IBM Power is the architecture of a line of high-end IBM server and supercomputer processors, descending from the 1990s IBM RS/6000, IBM POWER and Apple/IBM/Motorola PowerPC architectures. In contrast to the Arm-A architecture and multi-partner ecosystem, Power processors have historically been developed by IBM, with corresponding architecture specifications in lock-step, to fabricate and incorporate into IBM server products. The Power architecture is now managed by the OpenPOWER Foundation[2], part of the Linux Foundation.

Power is especially interesting from the relaxed-memory point of view because these have long been aggressive high-performance implementations, supporting many hardware threads with coherent shared memory.

Architecture versions and processor implementations

Date	Architecture version	Processor
2004	Power ISA 2.03	POWER5
2007	Power ISA 2.03 [7]	POWER6
2010	Power ISA 2.06	POWER7
2014	Power ISA 2.07	POWER8
2017	Power ISA 3.08B	POWER9
2021	Power ISA 3.1	POWER10
2021	Power ISA 3.1B	
2024	Power ISA 3.1C [73]	

For example, POWER8 supported up to 192 cores, each with up to 8 hardware threads, POWER9 supported 96 hardware threads per die, and POWER10 supports 240 hardware threads per socket. High-level descriptions of the processor implementations are in published papers, e.g. [100, 132, 128, 89, 84]. There are now some open-source Power processor designs, in addition to these IBM servers.

Documented instruction-set architecture The Power instruction-set behaviour is described in an informal (non-mechanised) pseudocode in the architecture manual. Part of this was semi-automatically translated into a previous version of Sail [77].

[2] https://en.wikipedia.org/wiki/OpenPOWER_Foundation

Documented relaxed concurrency architecture The Power architecture specification includes a prose description of its relaxed concurrency architecture [73, Book II, Chapter 1]. This prose has been lightly extended but never substantially revised – much dates back to at least the 2.03 version of the mid-2000s [7] – and it does not clearly define what relaxed behaviour is allowed. The examples, discussion, and operational model that we present are some of the results of an extended line of work, mainly by Alglave, Flur, Maranget, Pulte, Sarkar, Sewell, and others, variously in collaboration or separately, that developed a series of axiomatic and operational models for IBM Power [20, 21, 116, 34, 23, 115, 92, 93, 36, 77, 71]. Much of this was in collaboration or close discussion with Derek Williams, one of the senior IBM designers. Based on that, and on extensive experimental testing of various POWER machines, we are reasonably confident that what we describe is a good de facto standard model for Power.

15.1.3 RISC-V

The RISC-V architecture is an open standard architecture specification, available under royalty-free open-source licenses. Originating as a 2010 academic project by Asanović, Waterman, and Lee, the architecture is now managed by the non-profit RISC-V International (formerly the RISC-V Foundation), a consortium with many members.

Architecture versions The RISC-V architecture is structured as a base integer instruction set (in variants for 32 and 64 bits, and non-embedded/embedded) with many extensions. These are from time to time collected into new versions of the unprivileged and privileged architecture documents:

- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [113]
- The RISC-V Instruction Set Manual Volume II: Privileged Architecture [112]

Newly ratified extensions are typically documented stand-alone, and later incorporated into those.

Processor implementations RISC-V International does not itself design or build processors – that is done by its many members, and we will not attempt to describe them here.

Documented instruction-set architecture The RISC-V instruction behaviour is described in prose in their architecture manuals and formally in their Sail reference model [39, 18].

Documented relaxed concurrency architecture The RISC-V Foundation established its RISC-V memory model working group in 2017-03, chaired by Daniel Lustig [13]. Flur, Maranget, Pulte, Sarkar, and Sewell contributed substantially from 2017-07, together with several others, to what became the RISC-V weak memory ordering model (RVWMO), which was ratified in 2018-07. This is very similar (though not identical) to the multi-copy atomic Arm-A model. The RISC-V formal axiomatic and operational models were incorporated into the RISC-V unprivileged specification [74, 113, Chapter 17, Appendix A, Appendix B]. There is also a *Ztso* extension which strengthens the model to a version of TSO [113, Chapter 18].

15.2 Relaxed behaviour and abstract microarchitecture, informally

The first stored-program electronic computers, such as the EDVAC, EDSAC, and Manchester Baby, executed one machine instruction at a time, in program order, and did so relatively slowly. For example, the EDSAC operated at about 650 instructions per second, with a memory initially

of 512 17-bit words. The 75+ years between then and now have seen many orders of magnitude improvement in performance and capacity. Much of this has been driven by advances in the underlying devices and their fabrication: for many years up to the mid-2000s, shrinking transistor sizes meant that more could be used *and* that they could be switched faster, at the same power density (Dennard scaling).

Other performance gains have come from a range of *microarchitectural* innovations, also enabled by increased transistor counts, and it is some of these that give rise to observable relaxed-memory behaviour.

15.2.1 Microarchitecture optimisations and relaxed architecture specifications

The architectural interface has remained basically unchanged over these 75+ years: programs are still expressed as sequences of machine instructions stored at consecutive addresses, with occasional branches to other addresses. Many microarchitectural optimisations exploit the *instruction-level parallelism* and *locality* that is implicit in such a program. The basic ideas include:

- **pipelining:** splitting each instruction into steps, e.g. the instruction fetch, decode, arithmetic, memory accesses, and register writes, each of which can be done in parallel for different instructions.
- **superscalar execution:** executing multiple instructions simultaneously by dispatching to multiple decode and execution units.
- **out-of-order execution:** allowing execution of (parts of) instructions before program-order-earlier instructions, e.g. if the latter are awaiting results from memory, or if there is contention on some arithmetic unit.
- **speculative execution:** predicting (for example) whether a conditional branch will be taken or not, and speculatively executing the following instructions based on that – rolling back and discarding those parts of the execution if it turns out that the prediction was wrong.
- **hierarchical memory:** sophisticated buffering and cache hierarchies, with some memory close to the processor execution units (and hence fast, but necessarily small), backed up by larger but slower memory further away.

For all of these, the hardware designers' goal has been to increase performance (initially focussed on speed, but more recently on power efficiency) while preserving an illusion of in-order sequential execution, at least for execution within a single thread. That is sometimes achieved by blocking some operation until it will definitely not violate the sequential model, and sometimes by letting operations go ahead speculatively, but detecting and restarting the relevant part of the instruction stream execution, if that speculative execution is later found to violate the model.

Since the mid-2000s, transistor counts continued to increase (albeit more slowly), but the limits of Dennard scaling and power density, and the increasing challenge of extracting more implicit parallelism from sequential instruction streams, lead to an increased focus on explicit parallelism. This takes several forms, including multicore designs, vector extensions in the instruction set of conventional CPUs, and distinct GPUs and machine-learning accelerators (which have somewhat different programmers' models). Our focus here is on the former, and we do not discuss the latter two further, though there is a body of recent research on their relaxed concurrency models.

In multicore designs, the hardware directly supports multiple hardware threads of execution: with multiple independent cores on the same chip (multicore in a strict sense); and/or on multiple chips (multiprocessor systems); and/or with simultaneous multithreading (SMT), in which

the execution units of a core are shared between multiple threads. The distinctions between these are not very important for the programmers' view that we consider here: in each case, each hardware thread has its own architectural register state, but all appear to be executing above a common shared memory. Software threads are typically mapped onto these hardware threads by an operating system, which at any one time uses the available hardware threads to run that number of software threads, with a scheduler to manage context switches between them every so often. Multicore designs are now ubiquitous, except for small microcontrollers.

Multicore hardware implementations give programmers more ways to observe the effects of the above optimisations than one has in the single-core case, as multiple hardware threads can be attempting to access the same memory locations. That creates important choices for the design of the architecture specification – the intended programmers model – as one has to decide which of these effects are allowed to be programmer-observable, and which will be hidden from the programmer (modulo performance effects) by appropriate blocking or restarts. All modern high-performance processor implementations include all the above optimisations, and many more, but the different architectures have made different choices about what relaxed behaviour is exposed like this.

For normal memory accesses, the Intel and AMD x86 architectures choose to expose only the TSO effects of FIFO store buffering, while Arm-A, IBM Power, and RISC-V all choose to expose much more relaxed behaviour.

15.2.2 The pros and cons of relaxed architecture specifications

Intuitively, a relaxed architecture specification should permit hardware designs with less hardware-design complexity, improved performance, and improved scalability to many-core systems. However, assessing these quantitatively is a challenge, as each family of processor designs, architecture specification, and software corpus has now been co-developed and tuned for many years – it is hard to isolate the effects of the choices of how much relaxed behaviour to expose.

The downside of a more relaxed architecture is that it makes the programmers' model more complex, though this too is debatable – it is also arguable that the more relaxed models encourage programmers to more explicitly express the synchronisation that they actually need, rather than (perhaps accidentally) relying on the strength of TSO.

However, for our purposes here, we do not need to come to a conclusion on any of those questions: we are not aiming to prescribe or advocate whether architectures should in general be relaxed or not, but rather, given that much of the world's software has to execute above architectures that are relaxed, to show how they can be made precisely defined and well-understood.

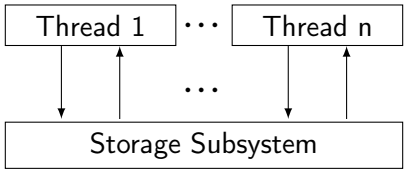
That said, for specific architecture design choices this work does sometimes provide good arguments one way or another, e.g. if a proposed architectural relaxation can be shown to be hard to implement higher-level language concurrency models above, or an architecture is found to be stronger than hardware implementations that it is intended to cover, or an architecture requires excessively complex models to precisely define.

15.2.3 Abstract microarchitecture – structure

Relaxed behaviour of hardware emerges from microarchitectural optimisations, and could in principle be understood and characterised by looking at them in detail, but that is not what we do, or what we should do, here. Microarchitecture design and hardware implementation is a whole field in itself, that of Computer Architecture, as described for example in the well-known textbooks of Hennessy and Paterson [79, 104] (note that this is a different sense of "architecture" to the *architecture specification* we focus on here). As noted in §1.6, the internal structure of processor implementations does not serve as a usable programming model: it is far too complex, it is generally commercially confidential, and it is too specific to each particular implementation.

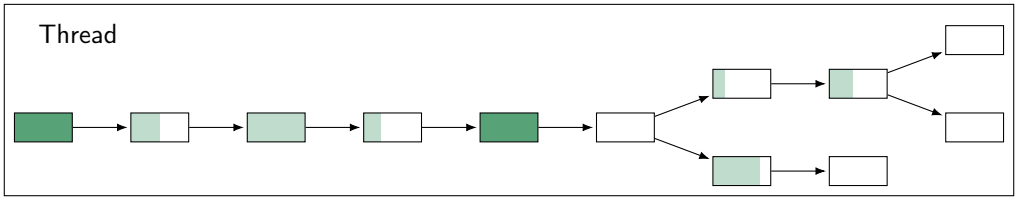
To define a good architecture specification, in contrast, we want to understand and capture *as little as possible* about the underlying microarchitectural implementations – just the aspects that give rise to observable relaxed behaviour, while abstracting from all the other details. It's useful to do this first entirely informally, with cartoon microarchitecture that conveys high-level ideas of what's going on in real implementations, and how programmers might think of those – just as we did with the cartoon diagrams of possible store-buffer configurations in Chapter 2. These can then be made precise, as architectural specifications in an abstract-microarchitectural style, e.g. with explicit speculative execution, but abstracting from all the details of any concrete pipeline design. That has typically been done by defining abstract-microarchitectural operational models, as we did for x86-TSO in Chapter 5, and as we'll do for Arm-A, RISC-V, and Power later in this Part. One could also define abstract-microarchitectural axiomatic models, but that is less common.

We factor our cartoon microarchitectures into a component for each hardware thread, executing instruction instances, above a common storage subsystem:



The interface between them consists roughly of the memory reads and writes done by each thread.

Thread semantics 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 program-order-predecessor instructions that are not yet known not to raise an exception, or program-order-predecessor instructions that might access the same address in a way that would violate coherence. We can think of such executions in terms of per-thread trees of instruction instances:



The diagram illustrates a snapshot of the history and current state of a single hardware thread, in which each rectangle represents a single instruction instance. The arrows show the program-order successor relation between instruction instances. The leftmost instruction instance is that at the start of the machine execution, with no predecessors. Some instruction instances have a single possible successor, while as-yet-unresolved conditional branches have two, and unresolved computed branches would have many. When a conditional branch is fully resolved, any not-taken subtree can be discarded. The rightmost instruction instances are those for which their successors have not yet been fetched.

Each instruction instance might have completed more or less of its intra-instruction behaviour, as shown by the shading. Some are finished, in solid dark green. These are not subject to roll-back or restart; they correspond in microarchitectural terms to retired instructions. Others are in flight, with light green shading indicating their progress through their intra-instruction behaviour. An out-of-order and speculative processor might execute parts of any of the in-flight instructions in any order, constrained in various ways that we will discuss. Such hardware will check, and block, roll back, or restart as needed, to ensure that this does not

violate the architected guarantees about observable sequential per-thread execution, coherence, or synchronisation.

This abstract view of out-of-order and speculative execution, in terms of the per-thread trees of instruction instances, will let us capture aspects of the underlying hardware optimisations that are relevant for the programmers' model of their behaviour, while abstracting from a mass of hardware-implementation detail. The latter is important for the underlying hardware implementations but not relevant for the programmers' model.

Instruction semantics To a first approximation, one can think of the intra-instruction behaviour of each instruction instance as a sequence of the register and memory actions that it does. Later, we will see that this is not always true – intra-instruction concurrency is required in some cases to give a good architectural specification.

The intra-instruction behaviour is defined relatively precisely by the vendors for each of the three architectures we consider, in broadly similar pseudocode languages. Arm provide definitions in their ASL specification language [109], RISC-V in Sail [39], and Power in a non-mechanised pseudocode. The Arm-A ASL can be automatically translated into Sail [39, 47], and part of the Power definition was semi-automatically translated into a previous version of Sail [77].

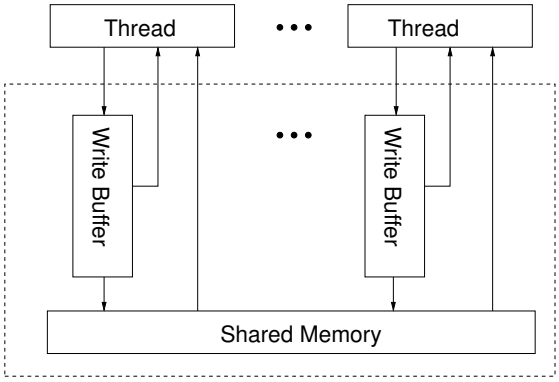
For example, a simplifed version of the Arm-A *Load Register (register)* instruction `LDR Xt, [Xn,Xm]`, where `t`, `n` and `m` are general purpose register IDs between 0 and 30, reads 8 bytes from the memory location that is the sum of the values in registers `Xn` and `Xm`, and writes that value to register `Xt`. Its intra-instruction semantics can be described with the following Sail definition:

```
function clause execute ( LDR((t, n, m)) ) = {
  /* Register read: ask for the value of register Xm and record it in local variable offset */
  offset    : bits(64) = rX(m);
  /* Register read: ask for the value of register Xn and record it in local variable base_addr */
  base_addr : bits(64) = rX(n);
  /* Compute the address */
  addr : bits(64) = base_addr + offset;
  /* Memory read: ask for the eight-byte value in memory starting from location addr and record
   *  it in local variable data */
  data : bits(64) = rMem(addr);
  /* Register write: ask for the value of data to be written to register Xt */
  wX(t) = data;
}
```

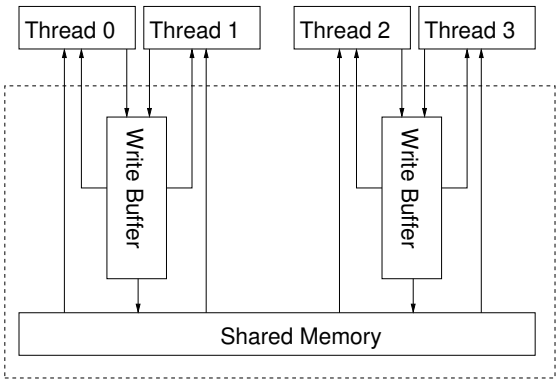
This defines the intra-instruction execution behaviour of this `LDR` instruction with sequential code, that uses Sail functions `rX()` and `wX()` to read and write registers, and `rMem()` to read memory. In a relaxed architecture, register and memory reads do not simply read from some global state – instead, the possible return values of those functions are provided by the thread and storage subsystem semantics.

Storage subsystem semantics 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:



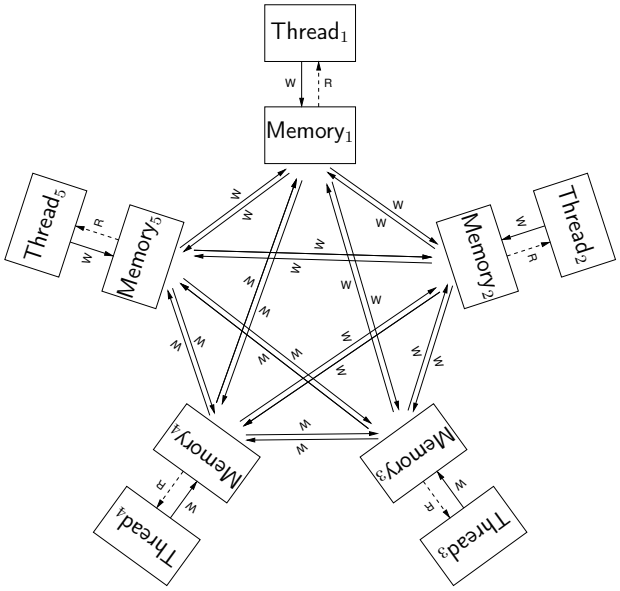
One can also have observably hierarchical buffering, as we discussed for IRIW in §2.6:



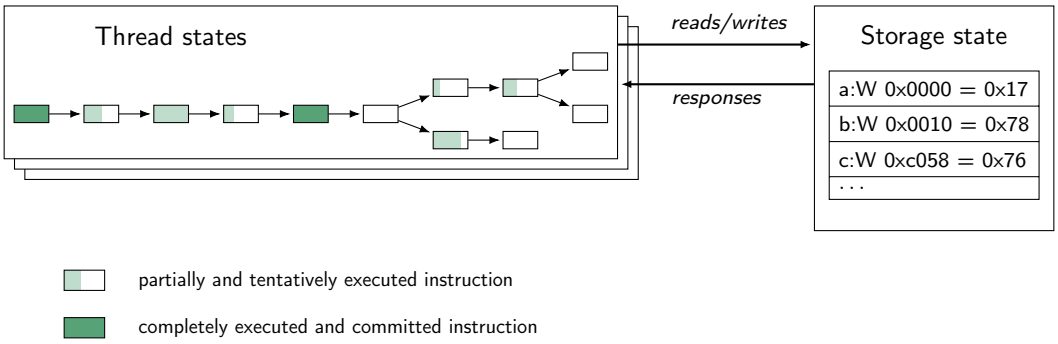
or non-FIFO buffers, or buffering of read requests in addition to writes, either together with writes or separately. High-performance interconnects might have separate paths for different groups of addresses; high-performance cache protocols might lazily invalidate cache lines; and certain atomic RMW operations might be done "in the interconnect" rather than in the core. One can capture all of these with various abstract storage subsystems.

For example, the Power architecture can be described abstractly in terms of a model in which writes and barriers can be propagated separately to a notional copy of memory for each hardware thread, and in which coherence choices are made incrementally [116], with an abstract microarchitecture as below. Note that in this model all the threads are symmetric, abstracting from the typically asymmetric hardware implementation. The hierarchical structure of the latter leads to non-uniform memory access time (NUMA) effects, and for performance optimisations

programmers may need to be aware of it, but for correctness, they should not need to.



Interestingly, some programmer-observable phenomena can be seen as arising either from thread or storage effects – then we can choose whether to include one, the other, or both. For Arm-A and RISC-V, it turns out that one can view all the storage subsystem relaxed effects as subsumed by thread relaxed effects, leading to a combined cartoon microarchitecture as below, and the "Flat" operational models, in which the storage subsystem is just a flat memory holding the most recent write to each address.



15.2.4 Abstract microarchitecture – behaviour

Having sketched the possible static structures of an abstract microarchitectural view, we now turn to the dynamic behaviour. The execution of an instruction in a concrete hardware implementation might involve many different steps, which we can abstract to the following. One doesn't always have to think about all of these, some are specific to particular architectures or particular models, and sometimes model design is informed by lower-level things – but these are usually a detailed enough view to understand the behaviour.

All instructions involve most of the following:

- **Fetch instruction** This represents the fetch and decode of a new instruction instance, as a program-order successor of a previously fetched instruction instance, or at the initial fetch address for a thread. In user models, that leave aside the systems issues of self-modifying code and address translation, we assume that instructions are fetched from a fixed program. Later work on instruction-fetch semantics generalises this to fetch instructions from the mutable shared memory.

- **Register read** A read of a register value from a register write by the most recent program-order preceding instruction instance that writes to that register. In an out-of-order machine, that preceeding instruction might not yet have reached that register write, in which case this will normally be blocked until it does.
- **Register write** A register write makes the value available for program-order succeeding instruction instances to read from.
- **Internal step** This covers internal computation of the intra-instruction semantics: arithmetic, auxiliary function calls, etc.
- **Finish instruction** At this point the instruction semantics execution is completed, the instruction cannot be restarted or discarded, and all memory effects have propagated to the storage subsystem. For a conditional branch, any non-taken subtrees of the instruction tree are discarded. This abstracts the microarchitectural notion of *retiring* an instruction.

Load instructions will also do the following:

- **Initiate memory reads of load instruction** At this point the memory footprint of the load is provisionally known – "provisionally" because some earlier instruction that feeds into the register reads determining that footprint might need to be restarted. The memory access is split to multiple single-copy atomic writes as required. There might be more than one memory read per load instruction if the address is misaligned, or for instructions such as the Arm Load-Pair. The individual reads can then start being satisfied.
- **Issue a memory read request** This abstracts the issuing of a memory read request by the thread to the storage subsystem.
- **Satisfy memory read from memory** Partially or entirely satisfy a single read from writes that have previously been propagated to memory.
- **Satisfy memory read by forwarding from writes** Partially or entirely satisfy a single read by forwarding writes from program-order preceding store instructions.
- **Complete load instruction (when all its reads are entirely satisfied)** At this point all the reads of the load have been entirely satisfied and the instruction semantics can continue execution.

All of these might be discarded and restarted, so the read(s) of a load instruction could be satisfied multiple times, with just the last ending up in the resolved execution.
Store instructions will also do the following:

- **Initiate memory writes of store instruction, with their footprints** At this point the memory footprint of the store is provisionally known. The memory access is split to multiple single-copy atomic writes as required. Note that the values for the writes may not be not available yet.
- **Instantiate memory write values of store instruction** At this point the store's writes have known values, and program-order succeeding reads can be satisfied by forwarding from them.
- **Commit store instruction** At this point the store is guaranteed to happen (it cannot be restarted or discarded), and its writes can start being propagated to the storage subsystem.
- **Propagate a memory write to another thread** Propagates a single write to the abstract copy of memory seen by another thread (this is used only for the Power architecture).
- **Propagate a memory write to storage** Propagates a single write to memory.
- **Complete store instruction** At this point all writes have been propagated to the storage subsystem, and the instruction semantics can continue execution.
- **Partial coherence commitment** (For the Power architecture only) The storage subsystem can internally commit to a more constrained coherence order for a particular address, adding an arbitrary edge, between a pair of writes to that address that have been seen already that are not yet related by coherence.

Barrier instructions will also do the following:

- **Commit barrier** At this point all the operations of instructions that precede the barrier and are relevant to the barrier have been performed.
- **Propagate barrier to another thread** (For the Power architecture only).

With address translation, each of the above memory accesses may need one or two stages of address translation, each of which might involve several more memory accesses, and involves further ordering constraints – all of which we ignore in this Part.

15.3 Relaxed architecture design principles and choices

The following chapters examine many relaxed-architecture design questions in detail, discussing relaxed behaviours that an architecture might allow or forbid, and how those choices can be formally captured in precise models.

Sequential behaviour for disjoint threads Threads that each operate on a disjoint part of the memory should appear to each execute sequentially.

No cross-thread speculation: speculative writes are not propagated to other threads

Branch speculation but no observable value speculation

15.4 Litmus tests

The Arm-A, IBM Power, and RISC-V architectures are not identical, but they share much behaviour. Many questions and litmus tests make sense for all three, though obviously the tests have to be expressed in the instructions of each architecture. We will typically show the common candidate execution together with the assembly-syntax litmus test for each architecture. For example, Fig. 16.3 presents the SB litmus test that we first discussed for x86.

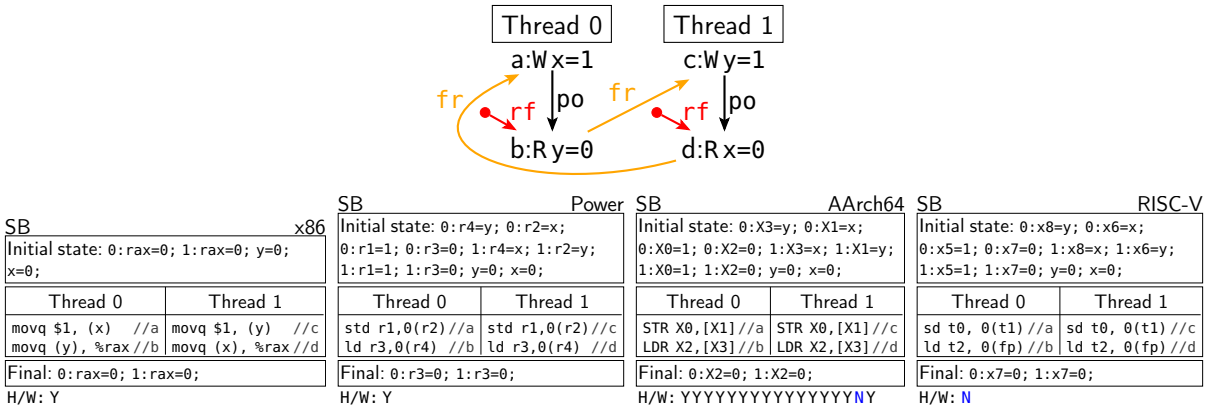


Figure 15.1: Litmus test SB

This shows a single candidate execution diagram and a sub-figure for each architecture, showing the architecture name in the top right of the sub-figure, the assembly code for that architecture, and more information that will be explained below. Some figures will not include all the architectures, as some are not relevant for particular litmus tests.

The litmus test name can be slightly different for each architecture when the test includes features that are named differently by the architectures. For example, each architecture has barrier

instructions of different strengths with different names, hence the name for SB with strong barriers in Arm's AArch64 is SB+dmb.sys, in Power it is SB+syncs, in RISC-V it is SB+fence.rw.rws, and in x86 it is SB+mfences. We use a generic name fen (fence), for the architecture-specific strong barrier, naming this generic litmus test SB+fens. The generic name also appears in the figure's caption.

The top box in the architecture sub-figure, under the architecture and litmus name, presents the assembly code for each thread. The comment at the end of some lines (all comments start with `//`), includes a single letter that is used in the execution diagram to identify memory accesses that are associated with the assembly instruction in that line.

The bottom box in the architecture sub-figure presents the initial state for the test, and a constraint on the final state's register and memory values, which might be allowed or forbidden by the architecture. In Fig. 16.3, in the initial state of the AArch64 sub-figure, `0:X3=y` means that register `X3` of Thread 0 is initialised with the address of a shared memory variable y. Shared memory variables are assumed to be distinct, and have 8 bytes of allocated memory each, starting from a properly aligned address. The final state constraint in the sub-figure is prefixed with "Allowed", and coloured in green, when the architecture model allows hardware implementations to exhibit a final state satisfying the constraint, and is prefixed with"Forbidden", and coloured in red, otherwise. Below each architecture sub-figure appears a sequence of Y, N, and - characters, labelled "HW observations". Each character represent experimental results from running the litmus test on a specific machine. The order of the characters corresponds to the order of the machines in **??**. The letters Y and N in the position corresponding to a machine, indicates that the litmus test was ran on the machine and the final state was exhibited, or not exhibited, respectively, by the machine. The - character indicates that this document was built without data from running the litmus test on the corresponding machine.

If the litmus test is forbidden by the architecture model but it was exhibited by the machine the letter Y will appear in red. This indicates that the hardware is incompatible with the architecture model. This could mean that the architecture model needs to be refined, or that the hardware has a bug. If the litmus test is allowed by the architecture model but it was not exhibited by the machine, the letter N will appear in blue. This may happen when a hardware implementation is stricter than the architecture, or when the litmus tool, that runs the tests on the machine, is not sufficient to elicit the interesting behaviour. In any case, this typically does not indicate anything wrong in either the hardware or the model. To summarise:

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

15.4.1 Candidate executions

The execution diagram above represents a possible complete execution of the litmus test, that starts from the initial state and ends in the final state. The diagram is a directed graph where the nodes are (single-copy atomic, see §16.2.3) memory access events (W for a memory write, and R for a memory read), and the edges are relations over the memory accesses as detailed below.

For example, the node a:Wx=1 in Fig. 16.3, represents the memory write event that is associated with the store instruction of Thread 0 (commented `//a` in the assembly). The x=1 indicates that the value 1 is written to the memory location associated with the shared variable x. Similarly, the node d:Rx=0 represent the memory read event that is associated with the load instruction of Thread 1 (commented `//d`). Here, the x=0 indicates that the value 0 is read from the memory location associated with the shared variable x.

In addition to the memory accesses that emerge from the instructions in the litmus test, and appear in the diagram, there are implicit memory writes that initialise all the memory locations. Those will appear in the diagram as a red dot, if some read is reading from them. If a specific value is assigned to a shared variable in the initial state part of the litmus test (in the architecture sub-figures) the memory location of the shared variable will be initialised with that value, otherwise it will be initialised with 0. Those implicit memory writes are assumed to be visible to all threads when the litmus test start executing. This can be seen in Fig. 16.3 as the two red dots above the events b, and d, with the red rf edges linking them to b, and d.

To determine the final memory state, additional implicit memory reads must be issued. Those are assumed to be executed after all the memory accesses of the test have taken their full effect. These implicit memory reads do not appear in the execution diagram.

Candidate executions now include more relations than for x86-TSO, to express various dependency relations and fences between memory access events. In the explanation below, r will denote memory read events, w will denote memory write events, and m will denote events that could be either memory read or write.

po **(program-order):** A po edge from m_1 to m_2 indicates that the instruction that generated m_1 precedes the instruction that generated m_2 in program order. Program order is the unfolding of the program instructions: regular instructions are program-order-before the following instruction memory location, and branch instructions are program-order before all their potential branch targets. The events of each thread in the diagram are always ordered from top to bottom according to program order, omitting some po edges to reduce clutter.

ctrl **(control dependency):** A ctrl edge from r to m indicates that the execution of the instruction that generated m is conditional and depends (through registers) on the value returned by r. In most cases r is a memory read (it can also be a potential memory write of a store-exclusive) that feeds (through registers) into a conditional or indirect branch instruction, b, and the instruction that generated m succeeds b in program-order. In the diagram we only draw the ctrl edge to the first such instruction, to prevent clutter.

In architectures that support delay slots, such as MIPS and SPARC, a conditional or indirect branch instruction can be succeeded by some instructions that will be executed irregardless of which branch target is taken (i.e. the delay slots). The architectures that we discuss in detail do not have delay slots, and the execution of every instruction that succeeds a conditional or indirect branch, in program-order, is conditional.

data **(data dependency):** A data edge from r to w indicates that the value returned by r feeds (through registers) into the value w writes to memory. In most cases r is a memory read (it can also be a potential memory write of a store-exclusive), and w is always a memory write.

addr **(address dependency):** An addr edge from r to m indicates that the value returned by r feeds (through registers) into the memory location m accesses. In most cases r is a memory read (it can also be a potential memory write of a store-exclusive).

fen **(fence/barrier):** A fen edge from m_1 to m_2 indicates that there is a strong fence/barrier instruction (`dmb sy/sync/fence rw,rw/mfence`) that succeeds m_1 and precedes m_2 in program-order. To prevent clutter we will only include such edges between the nearest m_1 and m_2, for every fence/barrier instruction (there are various kinds of fences, as we shall see). In addition, an instruction fence/barrier (`isb/isync`) is denoted ifen.

The edges po, ctrl, data, addr, and fen can only appear between instructions from the same thread. The following edges can appear between instructions either from the same or from different threads.

rf **(read from):** An rf edge from a write w to a read r indicates that r reads its value from w. Note that in order to make the litmus tests easier to understand, in all the tests used in this text, two store instructions that write to the same location will never write the same value. This makes pairing a read with the write it reads from simple.

co **(coherence):** A co edge from a write $w1$ to a write $w2$ indicates that $w1$ appears before $w2$ in the coherence-order of the execution (see §9.3 and §16.1.1).

fr **(from read):** An fr edge from a read r to a write w indicates that w succeeds the write from which r reads from, in the coherence-order of the execution [**?**]. In relational algebra $\mathsf{fr} = \mathsf{rf}^{-1} \circ \mathsf{co}$, where $^{-1}$ and $\circ$ are the converse and composition operations respectively.

Chapter 16

Arm-A, IBM Power, and RISC-V phenomena

16.1 Non-mixed-size Phenomena

16.1.1 Coherence

In relaxed architectures like those we consider, many of the effects of out-of-order execution and hardware optimisations are programmer visible, but to provide a reasonable programmers' model, architecture specifications must provide some guarantees, which constrain the hardware design and behaviour in various ways. The most fundamental of these is *coherence*, as introduced in §9.3. All the architectures we consider guarantee coherence for normal memory accesses. Coherence constrains the accesses to each memory location independently. In architecture-specification terms, it means:

1. in any execution, for each location, there is a total order over all the writes and reads to that location, which is consistent with each thread's program order, and in which each read reads from the most recent write.

For any execution and each location, the *coherence order*, `co`, is the restriction of that total order to the write events. Coherence prohibits the behaviours in Fig. 16.1 and Fig. 16.2, which show Arm-A, RISC-V, and IBM Power versions of the coherence litmus tests we saw in §9.3:

CoRW1 a read reading from a program-order-later write;

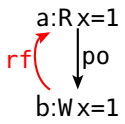
CoWW a write coherence-after a program-order-later write;

CoWR0 a read reading from a coherence-predecessor of a program-order-earlier write (which should hide that coherence-predecessor as far as this read is concerned);

CoRR two program-order-related reads reading from distinct writes in the opposite order to their coherence order; and

CoRW2 a write coherence-before the write that a program-order predecessor read from.

As we saw, forbidding these shapes is equivalent to the general characterisation of coherence above, and to the other formulations in §9.3. Importantly, coherence does not constrain memory accesses to disjoint locations in any way, as in each execution there is a separate order over the accesses to that location, with no constraints between them. This definition thus allows the relaxations that we'll see in the rest of this chapter. We discuss what happens for overlapping but distinct locations in §16.2, where we consider mixed-size effects. Note also that coherence does not constrain the ordering of two reads on the same thread which read from the same write. As



CoRW1 AArch64

Initial state: 0:X2=1; 0:X1=x; 0:X0=0; x=0;

```
LDR X0,[X1]          //a
STR X2,[X1]          //b
```

Final: 0:X0=1;

H/W: NNNNNN - NNNNNNNNNN

CoRW1 RISC-V

Initial state: 0:x7=1; 0:x6=x; 0:x5=0; x=0;

```
ld t0, 0(t1)         //a
sd t2, 0(t1)         //b
```

Final: 0:x5=1;

H/W: N

CoRW1 Power

Initial state: 0:r2=x;

```
lwz r1,0(r2)         //a
li r3,1
stw r3,0(r2)         //b
```

Final: 0:r1=1

H/W: -

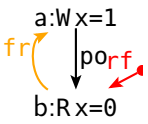
CoRW1 x86

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

```
movq (x), %rax       //a
movq $1, (x)         //b
```

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

H/W: -



CoWR0 AArch64

Initial state: 0:X1=x; 0:X0=1; 0:X2=0; x=0;

```
STR X0,[X1]          //a
LDR X2,[X1]          //b
```

Final: 0:X2=0;

H/W: NNNNNN - NNNNNNNNNN

CoWR0 RISC-V

Initial state: 0:x6=x; 0:x5=1; 0:x7=0; x=0;

```
sd t0, 0(t1)         //a
ld t2, 0(t1)         //b
```

Final: 0:x7=0;

H/W: N

CoWR0 Power

Initial state: 0:r2=x; 0:r1=1; 0:r3=0; x=0;

```
std r1,0(r2)         //a
ld r3,0(r2)          //b
```

Final: 0:r3=0

H/W: -

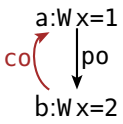
CoWR0 x86

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

```
movq $1, (x)         //a
movq (x), %rax       //b
```

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

H/W: -



CoWW AArch64

Initial state: 0:X2=2; 0:X1=x; 0:X0=1; x=0;

```
STR X0,[X1]          //a
STR X2,[X1]          //b
```

Final: x=1;

H/W: NNNNNN - NNNNNNNNNN

CoWW RISC-V

Initial state: 0:x7=2; 0:x6=x; 0:x5=1; x=0;

```
sd t0, 0(t1)         //a
sd t2, 0(t1)         //b
```

Final: x=1;

H/W: N

CoWW Power

Initial state: 0:r2=2; 0:r1=1; 0:r5=x;

```
stw r1,0(r5)         //a
stw r2,0(r5)         //b
```

Final: x=1

H/W: Y

CoWW x86

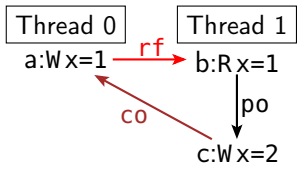
Initial state: x=0;

```
movq $1, (x)         //a
movq $2, (x)         //b
```

Final: x=1;

H/W: -

Figure 16.1: Single-threaded coherence litmus tests



CoRW2 AArch64

Initial state: `0:X1=x; 0:X0=1; 1:X2=2; 1:X1=x; 1:X0=0; x=0;`

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

Final: `1:X0=1; x=1;`

H/W: NNNNNN - NNNNNNNNNN

CoRW2 RISC-V

Initial state: `0:x6=x; 0:x5=1; 1:x7=2; 1:x6=x; 1:x5=0; x=0;`

Thread 0		Thread 1	
`sd t0, 0(t1)`	`//a`	`ld t0, 0(t1)`	`//b`
		`sd t2, 0(t1)`	`//c`

Final: `1:x5=1; x=1;`

H/W: N

CoRW2 Power

Initial state: `0:r2=x; 0:r1=1; 1:r3=2; 1:r2=x; 1:r1=0; x=0;`

Thread 0		Thread 1	
`std r1,0(r2)`	`//a`	`ld r1,0(r2)`	`//b`
		`std r3,0(r2)`	`//c`

Final: `1:r1=1; x=1;`

H/W: -

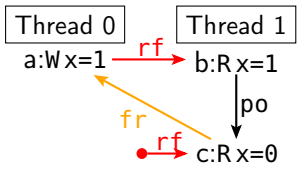
CoRW2 x86

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

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

Final: `1:rax=1; x=1;`

H/W: -



CoRR AArch64

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

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

Final: `1:X0=1; 1:X2=0; x=1;`

H/W: NNNNNN - NNNNNNNNNY

CoRR RISC-V

Initial state: `0:x6=x; 0:x5=1; 1:x6=x; 1:x5=0; 1:x7=0; x=0;`

Thread 0		Thread 1	
`sd t0, 0(t1)`	`//a`	`ld t0, 0(t1)`	`//b`
		`ld t2, 0(t1)`	`//c`

Final: `1:x5=1; 1:x7=0; x=1;`

H/W: N

CoRR Power

Initial state: `0:r2=x; 0:r1=1; 1:r2=x; 1:r1=0; 1:r3=0; x=0;`

Thread 0		Thread 1	
`std r1,0(r2)`	`//a`	`ld r1,0(r2)`	`//b`
		`ld r3,0(r2)`	`//c`

Final: `1:r1=1; 1:r3=0; x=1;`

H/W: -

CoRR x86

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

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

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

H/W: -

Figure 16.2: Multi-threaded coherence litmus tests

discussed in §16.1.8, out-of-order execution of such reads can be programmer-observable, some architectures allow it, and some hardware implementations take advantage of it.

A hardware implementation may have to take care to avoid coherence violations in multiple places. For example, in terms of the abstract microarchitectures of §15.2.3, potential coherence violations could arise both from out-of-order execution in the thread, or in the buffering and cache protocol of the storage subsystem. The former must constrain observable out-of-order execution of pairs of memory accesses that are to the same address (or which *might be* to the same address), either by blocking potentially violating steps, or by detecting potential violations and restarting as needed. The latter arises, even for fairly simple storage hierarchies, because caching creates multiple copies of memory data, to bring the data close to the hardware threads that are acting on it, but the coherence order (for each location) is global.

Coherence and cache coherence Discussions of coherence in hardware-design ("Computer Architecture") contexts often focus on *cache coherence*, speaking of the thread-semantics issues as particular *hazards*, but from an architecture-specification point of view, one has to consider the observable properties of the whole design.

It's hard to be sufficiently precise about this in prose without an underlying mathematical specification. For example, Hennessy and Patterson [79, p378] define:

> A memory system is coherent if
>
> 1. A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
> 2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
> 3. Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

In a relaxed setting, only parts of this really make sense. The third condition is straightforward, captured by the CoRR test and clearly implied by our definition above. The first appeals to a write by another processor "occuring between" a write and a read by P, but it's unclear what that is supposed to mean – it presumably refers to the global time of hardware execution, but, in all the menagerie of high-performance hardware optimisations, writes and reads are processed in many steps, and which step should be deemed to be the point at which they "occur", would be very specific to each design. The second condition leaves "sufficiently separated" vague, and also suffers from the same problem as the first.

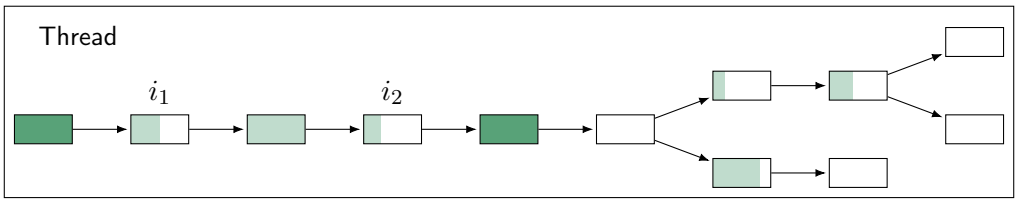
16.1.2 Out-of-order execution

For accesses to different locations, all three relaxed architectures by default permit observable out-of-order execution of normal memory accesses, for all four pairs of reads and writes: read program-order before read, write/write, write/read, and read/write. This enables the later instruction to make progress while the former instruction is blocked for some reason, for example, if it is waiting for some dependency to be resolved, or for space in a store buffer, or to gain sufficient ownership of the relevant cache line, or for its address translation to complete.

In abstract-microarchitectural terms, observable out-of-order execution can arise from thread or storage-subsystem effects. From the programmers' point of view, it is often not important to distinguish between these. In defining architectural models, sometimes one does, to retain a

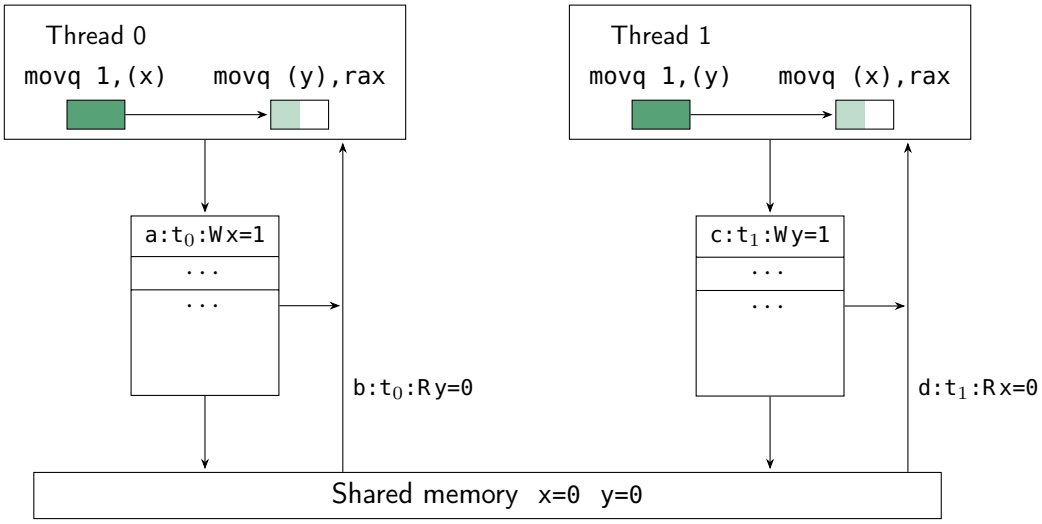
clear hardware-implementation intuition, and sometimes one abstracts from them, for model simplicity.

For an intuition of out-of-order thread execution arising from thread pipeline behaviour, consider the thread state below, in which instruction instances i_1 and i_2 are load or store instructions, and both have executed enough of their intra-instruction semantics to reach their memory accesses.



A hardware implementation, or an abstract-microarchitectural model, might let i_2 continue execution with its memory access before i_1 does its access. To avoid violating coherence, before i_2 is finally committed, it has to check that the two are to distinct addresses, and, as we discuss in the next section, certain dependencies also have to be respected.

On the storage-subsystem side, we saw for x86-TSO how per-thread write buffering gives rise to observable out-of-order execution of writes program-order before reads, even if the thread semantics executes instructions in-order and atomically: a program-order-later read can read from memory before a program-order-earlier write (to a different address) leaves the buffer and becomes visible to other threads. In the abstract-microarchitecture machine state below, the finished instructions are a prefix of the instruction instance tree for each thread, and both threads have finished executing their store instructions, but the load instructions can both read from the initial value of memory, before the writes are dequeued from the store buffers to memory.



In concrete hardware implementations, one can also have buffering of read requests, interconnects that are partitioned among sets of addresses (so the program-order-earlier access might hit congestion while the program-order-later access does not), and many other optimisations. In abstract microarchitectural models, we abstract from the details of those, for example, for Power, by regarding writes as propagating separately to each other thread.

SB: write/read reordering In terms of litmus tests, the basic SB test shows out-of-order execution of write/read pairs. This is often observed (the lack of observation on the RISC-V core tested is because that was a relatively simple non-optimised design).

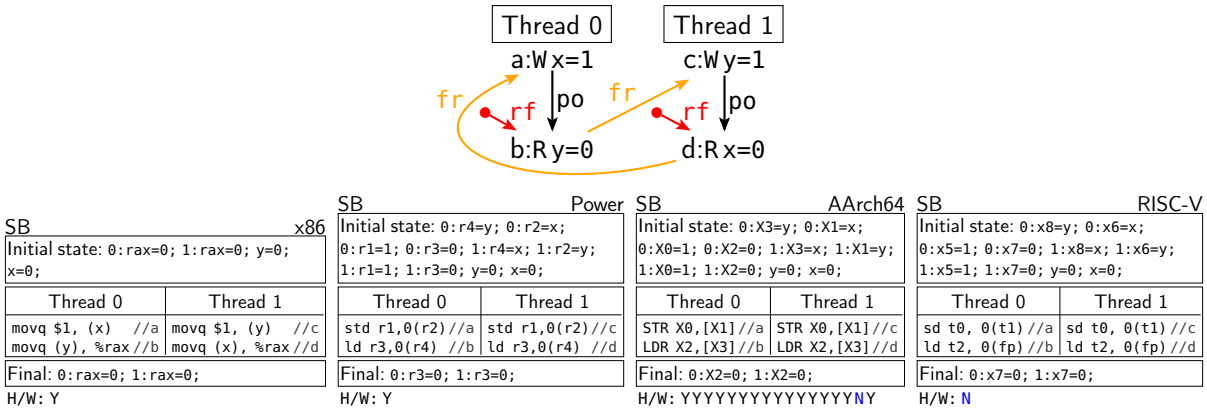


Figure 16.3: Litmus test SB

MP: write/write and read/read reordering The MP test could arise from out-of-order execu-

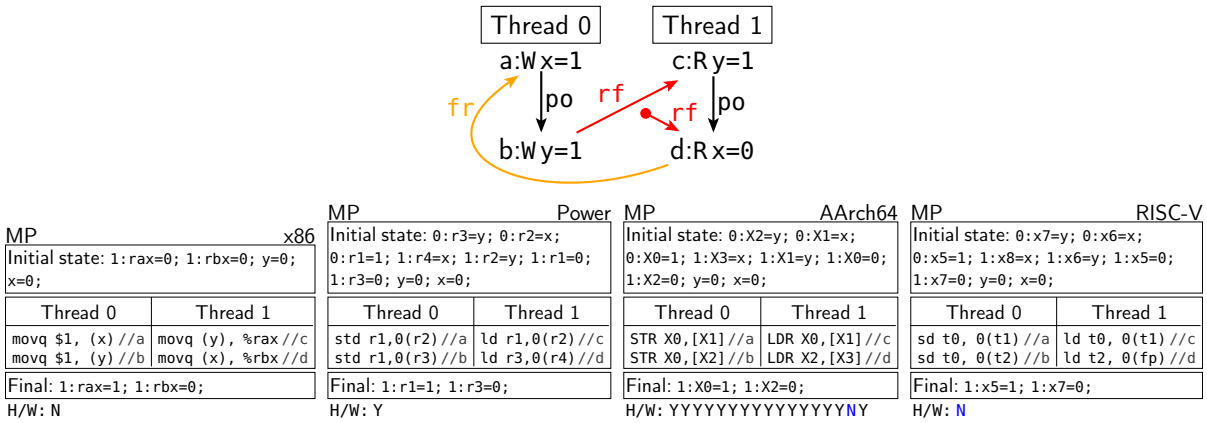


Figure 16.4: Litmus test MP

tion of the Thread 0 write/write pair, or from out-of-order execution of the Thread 1 read/read pair, or from non-FIFO write buffering or write propagation. We can isolate the write/write and read/read effects by adding a strong memory fence to one of the threads, which forces the hardware implementation to make any execution appear as if all the memory access instructions that are program-order before the fence have finished before any of the memory access instruction after the fence have started to execute. The litmus tests MP+po+fen and MP+fen+po do this for write/write and read/read pairs, respectively. Both of these tests are architecturally allowed and often observed.

LB: read/write reordering The LB test shows out-of-order execution of read/write pairs. Interestingly, this is allowed by all three architectures, but it has not been observed in practice on Power implementations, or on most Armv8-A and Armv9-A implementations (we lack test data on recent high-performance RISC-V implementations): a case where architectures are intentionally looser specifications than most implementations.

The reasons for this are interestingly intertwined with the asymmetry between reads and writes and the semantics of exceptions, as we discuss in [123, §4]. Writes are not made visible to other threads until they are known to be non-speculative, to avoid the need for cross-thread rollback. For LB to be observable, both writes must go ahead early, and be visible to the other thread, before the program-order-previous reads are satisfied – so both writes must be known to be non-speculative before the reads are satisfied. At first sight, this is no problem: the address translations of both accesses can complete early, and the reads and writes are to manifestly

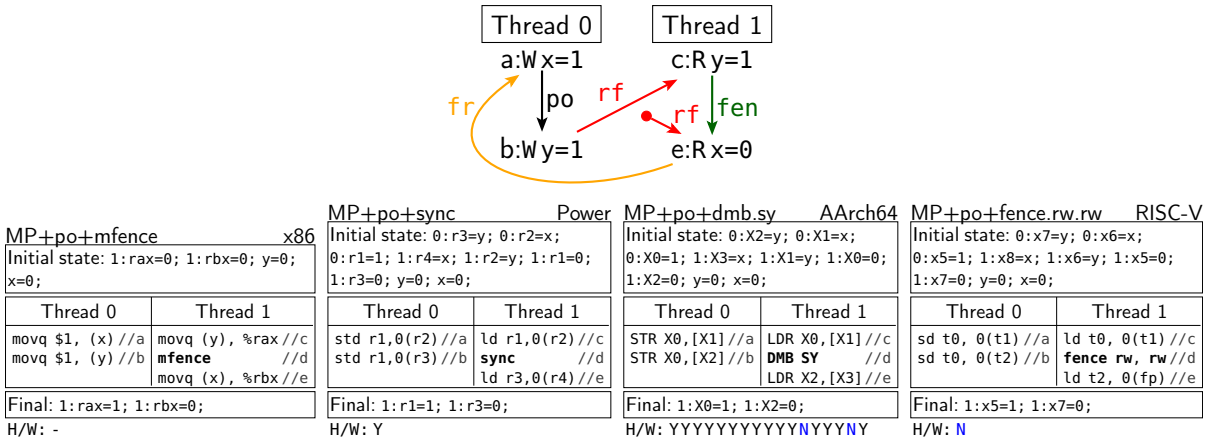


Figure 16.5: Litmus test MP+po+fen

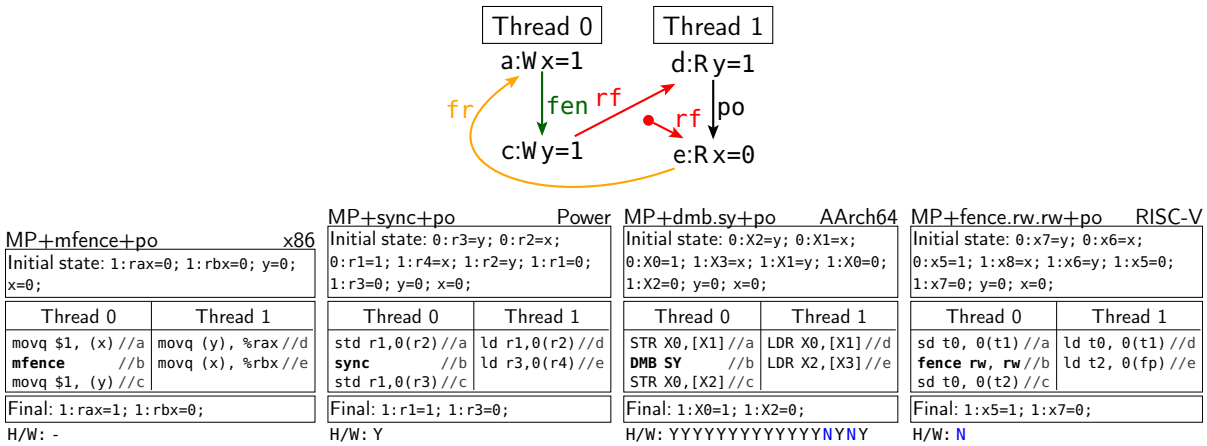


Figure 16.6: Litmus test MP+fen+po

different addresses, so there is no potential coherence violation. However, the memory system may detect errors such as data corruptions, e.g., using parity bits or error correcting codes, and signal those errors to software with an exception (in Arm-A, this class of exceptions is called "external aborts"). In systems with strong Reliability, Availablility, and Serviceability (RAS) goals (typically server processors) one wants these exceptions to be *precise*, i.e. (roughly), to appear as if at clean points in the instruction stream, with program-order previous instructions completed. In such a system, the possibility that one of the reads in LB might generate such an exception means that the program-order-later write must be considered speculative until it is known that that will not happen, which is basically when the read is satisfied – and thus LB will never be observable. In systems without strong RAS goals (e.g. mobile device processors), there might be no fine-grained recovery from such an exception – for example, one might just kill the entire process – and so it suffices for the exception to be imprecise. In that case, the write in LB can be considered non-speculative as soon as the address translations for the read and the write have completed, even if there might still be an (imprecise) external abort later – and thus LB could be observable.

Current architectures do not provide support for processor implementations to announce to software which of these cases applies, but that would be useful.

The fact that these architectures permit LB, combined with compiler optimisation, makes it very hard to define sensible relaxed concurrency models for programmming languages, that avoid the "thin-air problem". We'll return to this later.

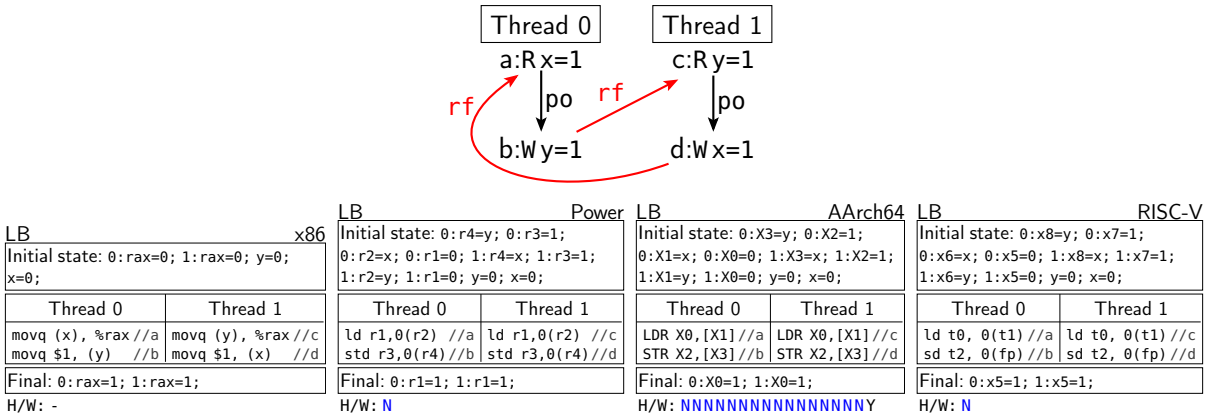


Figure 16.7: Litmus test LB

2+2W: write/write reordering and coherence The MP+po+fen test shows out-of-order execution of writes, as observed by a pair of reads. A related but different question is captured by test 2+2W, in which out-of-order execution of writes establishes coherence-order edges that

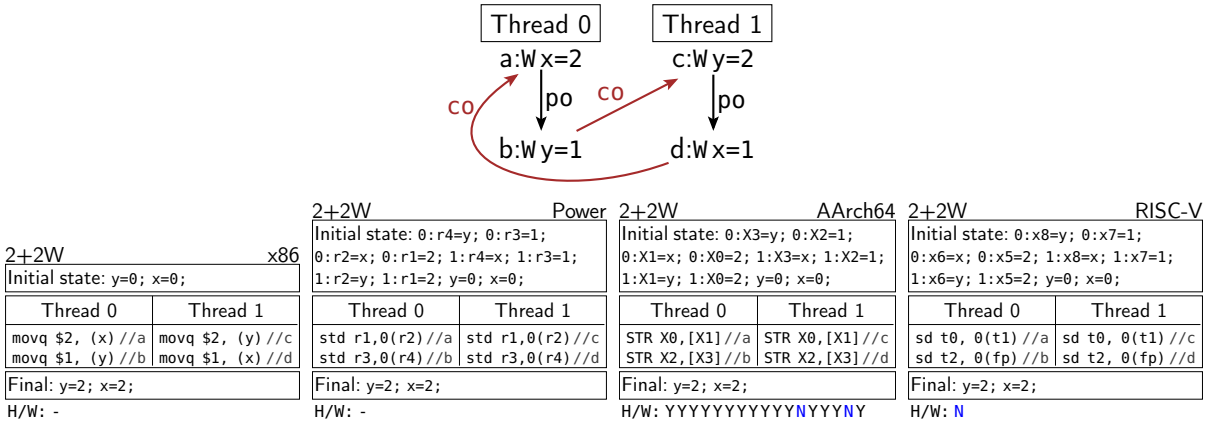


Figure 16.8: Litmus test 2+2W

form a cycle when combined with program order of the two threads.

This is also architecturally allowed and routinely observed (though not on exactly the same hardware implementations as MP+po+fen). This test becomes interesting when we come to consider weaker barriers.

R and S: the other four-edge two-thread two-location tests We saw in §13.1 that there are just six interesting (non-SC) test shapes that have a pair of accesses on each of two threads. We've already discussed four of them: SB, MP (with two variants), LB, and 2+2W. The other two shapes, R and S, are in most respects similar to SB and MP in terms of the allowed behaviour with added synchronsation: they are variants of SB and MP with a write-to-write coherence edge instead of a read-to-write from-reads edge. Without additional synchronisation, both are architecturally allowed on Arm-A, Power, and RISC-V.

16.1.3 Dependencies

The Arm, Power, and RISC-V architectures place some restrictions on observable out-of-order execution of memory accesses with *address and data dependencies*. In fact, the Arm Architecture Reference Manual [**?**, p. 90] lists the use of dependencies as a means to avoid an excessive number of barriers. Furthermore, the RISC-V Instruction Set Manual [**?**, p. 173] recognises

```
ldr X1,[X2]  // instruction i1; r1 is X1
add X3,X1,#1 // instruction i2; r2 is X1; r2' is X3
str X3,[X4]  // instruction i3; r3 is X3
```

Figure 16.9: AArch64 example of transitivity of data-flow through registers from the write to `X1` of the load instruction, to the read of `X3` of the store instruction.

that "modern code sometimes intentionally uses such dependencies as a particularly lightweight ordering enforcement mechanism". And the Power ISA book [**?**, p. 915] states that "...an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer".

Address and data dependencies are instances of data-flow between instructions of the same thread. Recall that in §16.1.1 (Coherence) it was mentioned that, in order to preserve sequential semantics of single-threaded programs, out-of-order execution must not appear to break data flow between instructions from the same thread. Address and data dependencies emerge from data-flow through registers, whereas coherence emerged from data-flow through memory.

An architecture will typically have a set of general purpose registers (GPRs) with sequential read and write semantics. The sequential semantics guarantees that an instruction that reads a GPR (or a portion of it) will see the value that is the combination of the GPR bits as they were written, each by the nearest program-order preceding instruction that writes to that GPR bit (or some initial bit value if no such instruction exists). Typically, special-purpose registers, such as stack pointer, and flag registers also have sequential semantics. In some architectures certain system registers require additional synchronisation to ensure sequential semantics [**?**]. In the following, a *normal register* refers to any register for which the architecture guarantees sequential semantics.

Here, for simplicity, it is assumed that for every register read, the most recent preceding register write to that register, writes to all the register bits that the read reads from. Later, §16.2.4 discusses instructions that can access specific bits of special-purpose registers, breaking the simplifying assumption, and how this affects dependencies. In addition, some architectures have a different semantics for the PC register: the implicit reads and writes for instruction fetch and branches might not respect dependencies, and there might not even be explicit reads and writes to the PC register. This will not be discussed here.

There is a data-flow through registers between two instructions if the latter instruction, in program-order, reads a value from a normal register that is computed from the value of a normal register that the former instruction writes to, and all the intermediate values of the computation are passed by instructions only through normal registers. That is, the computation does not involve writing an intermediate value to memory, or a non-normal register, and later reading it back in order to continue the same computation.

More precisely, the most simple case of data-flow through registers between two instructions is that in which an instruction reads the value of a normal register that was written to that register by a program-order preceding instruction. This is transitive (with some exceptions): if there is a data-flow through registers from instruction i_1's write to register r_1, to instruction i_2's read from register r_2, and there is a data-flow through registers from instruction i_2's write to register r'_2, to instruction i_3's read from register r_3, then there is also a data-flow through registers from instruction i_1's write to register r_1, to instruction i_3's read from register r_3. Fig. 16.9 shows an instance of transitivity of data-flow through registers.

The transitivity of data-flow through registers has two exceptions. The first one is where the intermediate instruction i_2 is a load instruction and the output register r'_2 is the register to which the load writes the loaded value from memory. Although this exception affects the dependencies that are defined below, which are an important part of the models of Arm, Power, and RISC-V,

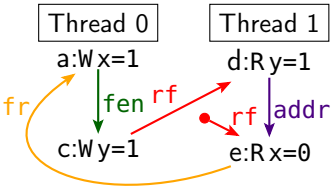
```
ldr X1,[X2]  // instruction i1; r1 is X1
blr X1       // instruction i2; r2 is X1; r2' is X30
ldr X3,[X30] // instruction i3; r3 is X30
```

Figure 16.10: AArch64 example of the second exception to data-flow through registers. There is no data-flow through registers from the write to `X1` of the first load instruction, to the read of `X30` of the second load instruction.

that will be discussed later, the models allow exactly the same behaviours with or without this exception.

The second and more significant exception is where the instruction semantics of the intermediate instruction i_2 has no data-flow from the input register r_2 to the output register r'_2. For example, in Fig. 16.10 there is no data-flow through registers from the write to `X1` of the first load instruction, to the read of `X30` of the second load instruction. This is because the AArch64 instruction `blr` (branch with link register) writes to register `X30` the value of the `PC` register plus 4, which does not feed from the value it reads from register `X1`. Because `blr` is an indirect branch instruction, there is a control dependency from the load instruction that precedes it, to the load instruction that succeeds it (discussed later, in §16.1.4), but this is different from data-flow and has different effects on the semantics of the code. Similar examples can be given for Power (using the `bctrl` instruction), and RISC-V (using the `jalr` instruction).

Data-flow through registers from a memory load instruction to a memory access instruction, where the register read of the latter instruction feeds into the memory address which the latter instruction access is called *address dependency*. Similar data-flow through registers, where the register read of the latter instruction feeds into the value that the latter instruction writes to memory is called *data dependency*.



MP+dmb.sy+addr AArch64

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X4=x; 1:X1=y; 1:X0=0; 1:X3=0; y=0; x=0;`

Thread 0	Thread 1
`STR X0,[X1] //a`	`LDR X0,[X1] //d`
`DMB SY //b`	`EOR X2,X0,X0`
`STR X0,[X2] //c`	`LDR X3,[X4,X2] //e`

Final: `1:X0=1; 1:X3=0;`

H/W: - - NNNN - N - NNNNNNNN

MP+fence.rw.rw+addr RISC-V

Initial state: `0:x7=y; 0:x6=x; 0:x5=1; 1:x9=x; 1:x6=y; 1:x5=0; 1:x8=0; y=0; x=0;`

Thread 0	Thread 1
`sd t0, 0(t1) //a`	`ld t0, 0(t1) //d`
`fence rw, rw //b`	`xor t2, t0, t0`
`sd t0, 0(t2) //c`	`add a0, s1, t2`
	`ld fp, 0(a0) //e`

Final: `1:x5=1; 1:x8=0;`

H/W: N

MP+sync+addr Power

Initial state: `0:r3=y; 0:r2=x; 0:r1=1; 1:r5=x; 1:r2=y; 1:r1=0; 1:r4=0; y=0; x=0;`

Thread 0	Thread 1
`std r1,0(r2) //a`	`ld r1,0(r2) //d`
`sync //b`	`xor r3,r1,r1`
`std r1,0(r3) //c`	`ldx r4,r3,r5 //e`

Final: `1:r1=1; 1:r4=0;`

H/W: Y

Figure 16.11: Litmus test MP+fen+addr

MP+fen+addr has two loads from different memory locations, with an address dependency between them. In all three architectures this dependency is enough to forbid observable read-read reordering of the loads, making the behaviour of the litmus test forbidden by the architecture. Similarly, the data dependencies in LB+datas forbid observable read-write reordering in all three architectures.

Although dependencies restrict out-of-order execution, other reorderings in the storage subsystem may still cause non-SC behaviour. For example, the Arm, Power, RISC-V, and x86 architectures all allow SB+rfi-addrs, and the Power architecture also allows IRIW+addrs. Those

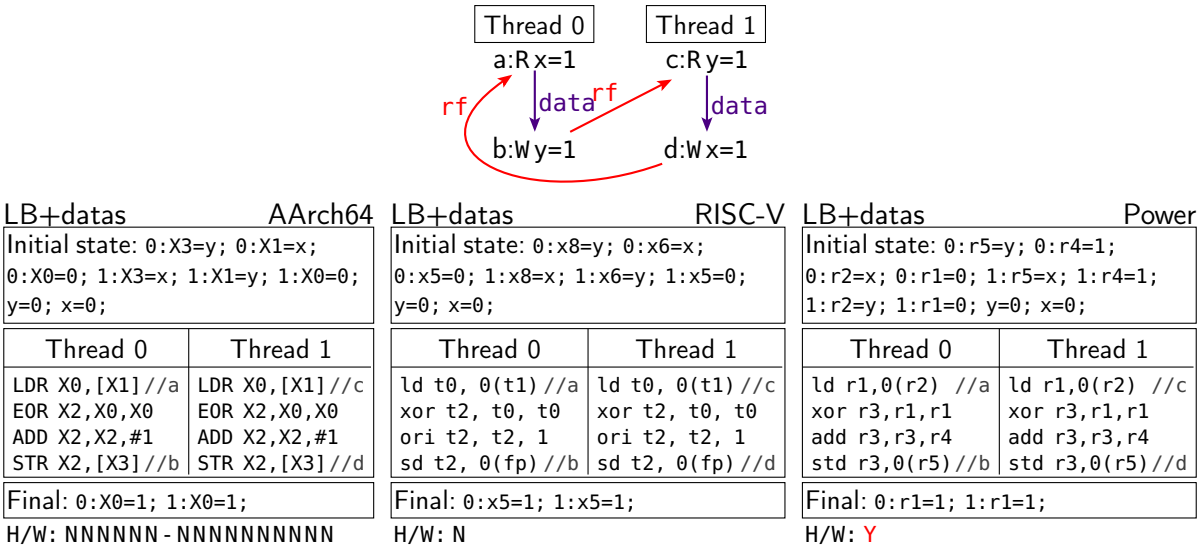


Figure 16.12: Litmus test LB+datas

are instances of non-multi-copy atomicity and variations of it, which will be discussed later in §16.1.14.

Value Speculation

The restrictions that Arm, Power, and RISC-V place on observable out-of-order execution of address and data dependencies effectively forbid observable value speculation of normal registers. That is, a hardware implementation that implements one of those architectures is not allowed to observably speculate the value of a register that is required during the execution of an instruction, in order to proceed with the execution. Value speculation is different from obtaining a value by speculative execution of a load instruction, as discussed later in §16.1.4.

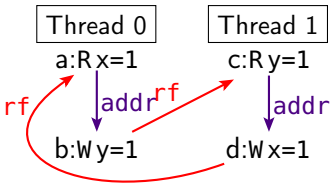
For example, if a hardware implementation were allowed to speculate register values, when executing MP+fen+addr, the hardware implementation could immediately guess the value of the register that feeds into the address of the second load (from x) of Thread 1 and execute that load before the first load (from y), and before any of the values that Thread 0 writes to memory take effect. The second load (from x) in this case could return the value 0, while the first load (from y) return 1. Hence, such value speculation would allow the hardware implementation to exhibit MP+fen+addr.

Note that value speculation does not break sequential semantics of single threaded code if the hardware implementation takes the necessary precautions. In particular, after speculating a value, and when the necessary information becomes available, the hardware implementation could validate the speculation and if it turns out to be wrong, perform a roll-back of the affected instructions, and re-execute them as necessary. Yet, such behaviour is forbidden by the Arm, Power, and RISC-V architectures, as it observably breaks the ordering guarantees of address and data dependencies.

The Alpha architecture [**?**, p. 5-15] notoriously does not strictly respect dependencies, which has the effect of allowing some value speculation. In particular, from the model given in the Alpha Architecture Reference Manual [**?**] it is clear that the Alpha architecture allows MP+fen+addr, though it does not allow LB+datas and LB+addrs.

Artificial dependencies

The Arm, Power, and RISC-V architectures forbid hardware implementations to observably optimise away address and data dependencies, even when the computation that generates the



LB+addrs AArch64

Initial state: 0:X4=y; 0:X3=1; 0:X1=x; 0:X0=0; 1:X4=x; 1:X3=1; 1:X1=y; 1:X0=0; y=0; x=0;

Thread 0	Thread 1
LDR X0,[X1] //a	LDR X0,[X1] //c
EOR X2,X0,X0	EOR X2,X0,X0
STR X3,[X4,X2] //b	STR X3,[X4,X2] //d

Final: 0:X0=1; 1:X0=1;

H/W: NNNNNN - NNNNNNNNNN

LB+addrs RISC-V

Initial state: 0:x9=y; 0:x8=1; 0:x6=x; 0:x5=0; 1:x9=x; 1:x8=1; 1:x6=y; 1:x5=0; y=0; x=0;

Thread 0	Thread 1
ld t0, 0(t1) //a	ld t0, 0(t1) //c
xor t2, t0, t0	xor t2, t0, t0
add a0, s1, t2	add a0, s1, t2
sd fp, 0(a0) //b	sd fp, 0(a0) //d

Final: 0:x5=1; 1:x5=1;

H/W: N

LB+addrs Power

Initial state: 0:r5=y; 0:r4=1; 0:r2=x; 0:r1=0; 1:r5=x; 1:r4=1; 1:r2=y; 1:r1=0; y=0; x=0;

Thread 0	Thread 1
ld r1,0(r2) //a	ld r1,0(r2) //c
xor r3,r1,r1	xor r3,r1,r1
stdx r4,r3,r5//b	stdx r4,r3,r5//d

Final: 0:r1=1; 1:r1=1;

H/W: N

Figure 16.13: Litmus test LB+addrs

dependency is redundant.

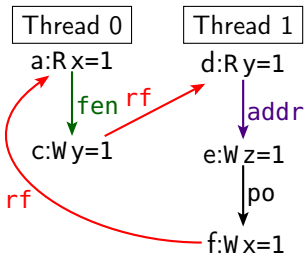
For example, the address dependency in the AArch64 litmus test MP+dmb.sy+addr in Fig. 16.11 is created by XORing the value that the first load (from y) reads from memory with itself, and using the result, which is always 0, as an offset from x in the second load (from x). If a hardware implementation could optimise away dependencies, it could compute the result of the `EOR` instruction before obtaining the value of the input register, as the XOR of any value with itself is always 0, and execute the second load (from x) of Thread 1 before the first load (from y), and before any of the values that Thread 0 writes to memory take effect. The second load (from x) in this case would return the value 0 while the first load (from y) return 1. Hence, such optimisation would allow the hardware implementation to exhibit MP+fen+addr. This kind of behaviour is forbidden by the three architectures.

In the example above, and similar cases with bitwise AND (when one of the operands is 0) and bitwise OR (when all the bits of one of the operands are 1s), it is obvious that the result of the computation is independent of the input register, and therefore the dependency is *artificial*. Therefore, when designing an architecture, one might consider not to respect artificial dependencies, as Arm did in earlier versions of the Arm architecture. However, defining precisely artificial dependencies that the architecture may not respect (or alternatively, defining precisely just the real dependencies that the architecture does respect) inevitably involves bounding the information available to the optimisation, which is not very appealing. Moreover, the optimisation is with respect to the semantics, which itself involves dependencies. This, if not defined carefully, creates circular reasoning in the semantics which can easily give rise to out-of-thin-air behaviour. Moreover, one might expect that compilers would in any case be able to do such optimisations, though there remains the problem of how the programmer should be able to indicate that dependencies should be respected in the generated code where required.

Address-induced control dependency

Address dependencies have side effects that data dependencies do not. A memory access can only be speculative until all the address dependencies of preceding instructions are resolved. This is due to two reasons. First, while the address dependencies of preceding instructions are not resolved, the hardware implementation can not guarantee that the succeeding memory access will not violate coherence (in the case where a preceding memory access turns out to have an overlapping memory footprint, after its address dependency is resolved). Second, when the address dependency is resolved an address translation is performed (when enabled), which

might trigger an exception if something goes wrong with the translation. Hence, address translation is similar to conditional branching, in the sense that the succeeding instruction might be either the instruction in location PC+4, or the instruction in the relevant exception handler location. Therefore, before the address dependency is resolved the hardware implementation must regard any succeeding instruction as speculative. The result of this is that LB+fen+addr-po is



LB+dmb.sy+addr-po AArch64

Initial state: 0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X5=x; 1:X4=z; 1:X3=1; 1:X1=y; 1:X0=0; z=0; y=0; x=0;

Thread 0	Thread 1
LDR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X2,[X3] //c	STR X3,[X4,X2] //e
	STR X3,[X5] //f

Final: 0:X0=1; 1:X0=1;

H/W: - - - - - - - - - - - - - - - - -

LB+fence.rw.rw+addr-po RISC-V

Initial state: 0:x8=y; 0:x7=1; 0:x6=x; 0:x5=0; 1:x11=x; 1:x9=z; 1:x8=1; 1:x6=y; 1:x5=0; z=0; y=0; x=0;

Thread 0	Thread 1
ld t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t2, 0(fp) //c	add a0, s1, t2
	sd fp, 0(a0) //e
	sd fp, 0(a1) //f

Final: 0:x5=1; 1:x5=1;

H/W: -

LB+sync+addr-po Power

Initial state: 0:r4=y; 0:r3=1; 0:r2=x; 0:r1=0; 1:r6=x; 1:r5=z; 1:r4=1; 1:r2=y; 1:r1=0; z=0; y=0; x=0;

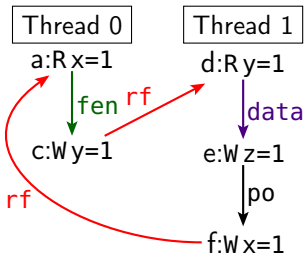
Thread 0	Thread 1
ld r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r3,0(r4) //c	stdx r4,r3,r5//e
	std r4,0(r6) //f

Final: 0:r1=1; 1:r1=1;

H/W: -

Figure 16.14: Litmus test LB+fen+addr-po

forbidden, while LB+fen+data-po is allowed, by all three architectures.



LB+dmb.sy+data-po AArch64

Initial state: 0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X5=x; 1:X4=1; 1:X3=z; 1:X1=y; 1:X0=0; z=0; y=0; x=0;

Thread 0	Thread 1
LDR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X2,[X3] //c	ADD X2,X2,#1
	STR X2,[X3] //e
	STR X4,[X5] //f

Final: 0:X0=1; 1:X0=1;

H/W: - - - - - - - - - - - - - - - - -

LB+fence.rw.rw+data-po RISC-V

Initial state: 0:x8=y; 0:x7=1; 0:x6=x; 0:x5=0; 1:x10=x; 1:x9=1; 1:x8=z; 1:x6=y; 1:x5=0; z=0; y=0; x=0;

Thread 0	Thread 1
ld t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t2, 0(fp) //c	ori t2, t2, 1
	sd t2, 0(fp) //e
	sd s1, 0(a0) //f

Final: 0:x5=1; 1:x5=1;

H/W: -

LB+sync+data-po Power

Initial state: 0:r4=y; 0:r3=1; 0:r2=x; 0:r1=0; 1:r6=x; 1:r5=z; 1:r4=1; 1:r2=y; 1:r1=0; z=0; y=0; x=0;

Thread 0	Thread 1
ld r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r3,0(r4) //c	add r3,r3,r4
	std r3,0(r5) //e
	std r4,0(r6) //f

Final: 0:r1=1; 1:r1=1;

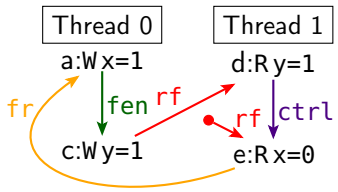
H/W: -

Figure 16.15: Litmus test LB+fen+data-po

16.1.4 Speculative execution - branching

In order for a hardware implementation to execute instructions out-of-order, it has to fetch instructions before the execution of previous instructions has completed. When one of these incomplete instructions is a conditional or indirect branch instruction the hardware implementation does not know which location should be fetched from next, as that location is computed by the incomplete branch instruction. An optimised hardware implementation often use *branch prediction*, a technique where the next fetching location is guessed using heuristics, and proceed with out-of-order execution based on that guess. Some hardware implementations may even proceed with more than one guess for the same branch instruction at the same time. Execution of instructions following a branch prediction is called speculative execution.

Speculative execution makes it possible to exhibit the MP+fen+ctrl litmus test, where



MP+dmb.sy+ctrl AArch64

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

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	CBNZ X0,LC00
STR X0,[X2] //c	LC00:
	LDR X2,[X3] //e

Final: 1:X0=1; 1:X2=0;

H/W: YYYYYY - YYYYYYNYNY

MP+fence.rw.rw+ctrl RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x8=x; 1:x6=y; 1:x5=0; 1:x7=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	bne t0, zero, LC00
sd t0, 0(t2) //c	LC00:
	ld t2, 0(fp) //e

Final: 1:x5=1; 1:x7=0;

H/W: N

MP+sync+ctrl Power

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r4=x; 1:r2=y; 1:r1=0; 1:r3=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	cmpw r1,r1
std r1,0(r3) //c	bc 12,2,4
	LC00:
	ld r3,0(r4) //e

Final: 1:r1=1; 1:r3=0;

H/W: Y

Figure 16.16: Litmus test MP+fen+ctrl

Thread 0 is forced to execute in-order by a fence, and Thread 1's load from x (event e) is preceded by a conditional branch instruction, the condition of which depends on the value that the load from y (event d) returns. A hardware implementation with speculative execution may guess, before the load from y is performed, that the load from x will follow the conditional branch, and so the hardware implementation may execute the load from x before any of the values that Thread 0 writes to memory take effect. The load from x in this case would return the value 0, while the load from y return 1. Hence, speculative execution can allow a hardware implementation to exhibit MP+fen+ctrl.

The relation between the load from y and the load from x in the example above is another form of dependency, a *control dependency*. There is a control dependency from a load instruction to an instruction that succeeds it in program-order if: there is a conditional/indirect branch instruction between them; and there is data flow through normal registers from the load instruction to the branch instruction, that feeds into the computation of the branch target.

As demonstrated by MP+fen+ctrl, control dependency in the Arm, Power and RISC-V architectures, does not restrict read-read reordering. However, control dependencies in all three architectures do restrict read-write reordering to some extent, as demonstrated by the LB+ctrls litmus test. If a hardware implementation would allow a speculative write (as in read-write reordering with control dependency) to be observed by another thread, as would be required in order to exhibit LB+ctrls (assuming it is the architecture's intention to allow it), the hardware implementation would have to coordinate a roll-back between multiple hardware threads, and the associated caches and memory, in the case where the speculated write turns out to be on

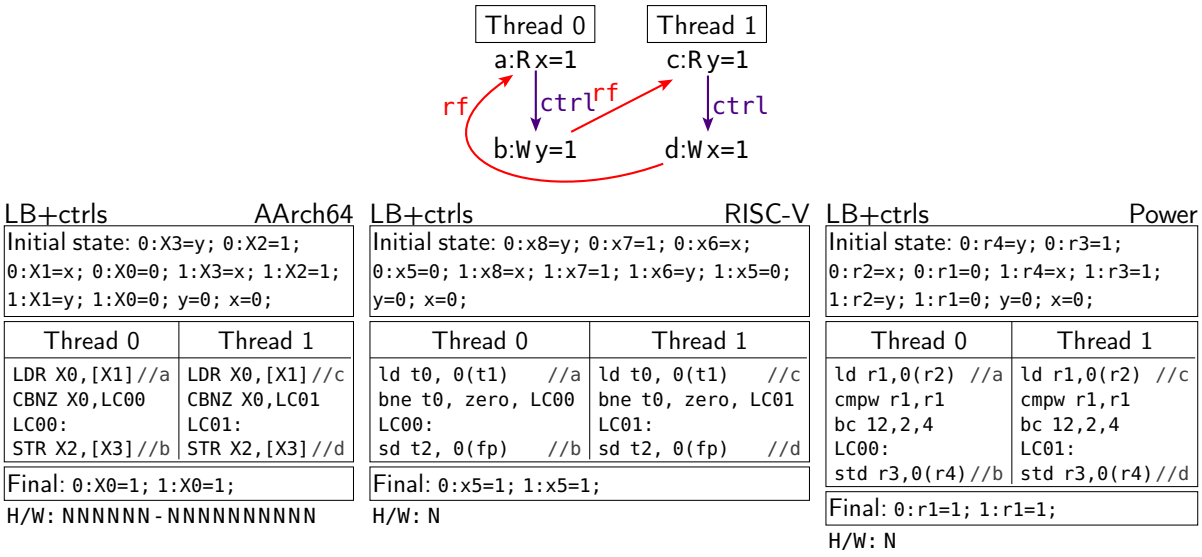


Figure 16.17: Litmus test LB+ctrls

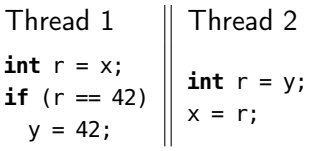


Figure 16.18: Out-of-thin-air values

an untaken branch, which would violate the usual hardware-implementation desire for each module to be as local as possible. Moreover, such behaviour would also allow executions where the speculated write feeds into the condition that determines if the branch on which the write is on would be taken or not, creating a causality loop. This behaviour would give rise to out-of-thin-air values, as demonstrated in Fig. 16.18. An execution that speculates the condition of the **if** in Thread 1 to be true, and allows the speculative assignment of 42 to y to be observed by Thread 2, can continue with the execution of Thread 2, in-order, and assign 42 to x, which would validate the speculation of Thread 1. Allowing out-of-thin-air values is undesirable, as it makes reasoning about programs very hard.

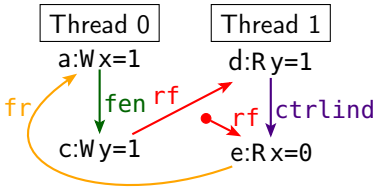
Observing a speculative write by the same thread is easier to roll-back if the speculation turns out to be wrong, and it cannot lead to causality loops, hence some architectures allow it. This case is discussed later in §16.1.6.

Note that in all the litmus tests here that include a conditional branch, both targets of the conditional branch are the same (whether the branch is taken or not), which could be considered an artificial control dependency. Similar to address and data dependencies, the Arm, Power, and RISC-V architectures forbid hardware implementations to observably optimise away control dependencies, even when the computation that generates the dependency is artificial, or when both targets are the same.

When speculating the target of a conditional branch, a hardware implementation is effectively doing a value speculation of one bit, that determines which of the branch targets is taken. The hardware implementation does not need to speculate the target locations, as those are encoded in the instruction semantics: if the condition does not hold, the target is the instruction that immediately follows the branch instruction in memory; and if the condition does hold the target is computed by adding to the program-counter an offset that is encoded in the branch instruction machine code. One might think that the architectures allow this sort of one bit value speculation, and forbid a more general value speculation as discussed in §16.1.3, because a one bit speculation is easier to deal with, but this is not the case. The architectures also allow spec-

ulative executions of indirect branch instructions, in which the target of the branch is computed from a value held in a register, that might not be readily available at the time of speculation. In this case the architecture is effectively allowing hardware implementations to speculate the value of the register holding the branch target, but only for the purpose of computing the target and not for any other computation.

For example, MP+fen+ctrlind, where the control dependency in Thread 1 is created using



MP+dmb.sy+ctrlind AArch64

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

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X4,X0,X0
STR X0,[X2] //c	ADR X5,LC00
	ADD X6,X5,X4
	BR X6
	LC00:
	LDR X2,[X3] //e

Final: 1:X0=1; 1:X2=0;

H/W: - - - - - - - - - - - - - - - - -

MP+fence.rw.rw+ctrlind RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x9=LC00; 1:x8=x; 1:x6=y; 1:x5=0; 1:x7=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor a0, t0, t0
sd t0, 0(t2) //c	add a0, a0, s1
	jalr zero, a0, 0
	LC00:
	ld t2, 0(fp) //e

Final: 1:x5=1; 1:x7=0;

H/W: -

MP+sync+ctrlind Power

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r4=x; 1:r2=y; 1:r1=0; 1:r3=0; 1:r5=P1:LC00; y=0; x=0;

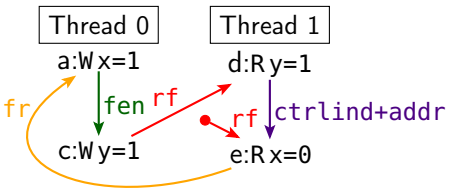
Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r6,r1,r1
std r1,0(r3) //c	add r6,r6,r5
	mtlr r6
	blr
	LC00:
	ld r3,0(r4) //e

Final: 1:r1=1; 1:r3=0;

H/W: -

Figure 16.19: Litmus test MP+fen+ctrlind

an indirect branch instruction, is allowed by the Arm, Power and RISC-V architectures, but the similar MP+fen+ctrlindaddr litmus test, where the register holding the target of the indirect



MP+dmb.sy+ctrlindaddr AArch64

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

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X4,X0,X0
STR X0,[X2] //c	ADR X5,LC00
	ADD X6,X5,X4
	BR X6
	LC00:
	EOR X7,X6,X6
	LDR X2,[X3,X7] //e

Final: 1:X0=1; 1:X2=0;

H/W: - - - - - - - - - - - - - - - - -

MP+fence.rw.rw+ctrlindaddr RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x9=LC00; 1:x8=x; 1:x6=y; 1:x5=0; 1:x7=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor a0, t0, t0
sd t0, 0(t2) //c	add a0, a0, s1
	jalr zero, a0, 0
	LC00:
	xor a1, a0, a0
	add a2, a1, fp
	ld t2, 0(a2) //e

Final: 1:x5=1; 1:x7=0;

H/W: -

MP+sync+ctrlindaddr Power

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r4=x; 1:r2=y; 1:r1=0; 1:r3=0; 1:r5=P1:LC00; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r6,r1,r1
std r1,0(r3) //c	add r6,r6,r5
	mtlr r6
	blr
	LC00:
	xor r7,r6,r6
	ld r3,r7,r4 //e

Final: 1:r1=1; 1:r3=0;

H/W: -

Figure 16.20: Litmus test MP+fen+ctrlindaddr

branch is also used to create an address dependency, is forbidden by those architectures.

Modelling the speculation of indirect branch instructions, in a way that also allows exhaustive enumeration of all valid executions of reasonable litmus tests, in finite time, is challenging. See discussion later in **??**.

16.1.5 Instruction Barrier

As fetching an instruction involves reading it from memory, it can be expected that some synchronisation will have to be used to guarantee the fetch includes the intended instruction (e.g. the code most recently produced by a just-in-time (JIT) compiler). Such synchronisation has to handle cases where the code was loaded from a different thread than the one executing it. In addition, to allow data and instructions to be efficiently cached separately, synchronisation might also be needed even if the code is executed by the same thread that loaded it. The synchronisation needed to guarantee the execution of newly loaded code was investigated and modelled by Simner et al. [**?**], and is beyond the scope of this chapter. However, the instruction fetching synchronisation includes an *instruction barrier* instruction (`ISB/isync/fence.i`) which has interesting side effects that affect load instructions, which will be covered in this subsection.

In the Arm and Power architectures (but not in RISC-V), when an instruction barrier is placed after a conditional or indirect branch, it prevents observable read-read reordering of any load that feeds into the branch instruction with any load that follows the barrier in program-order. This is because the instruction barrier effectively stops fetching until the barrier is not speculative, as part of the fetching synchronisation. This is demonstrated by MP+fen+ctrlifen,

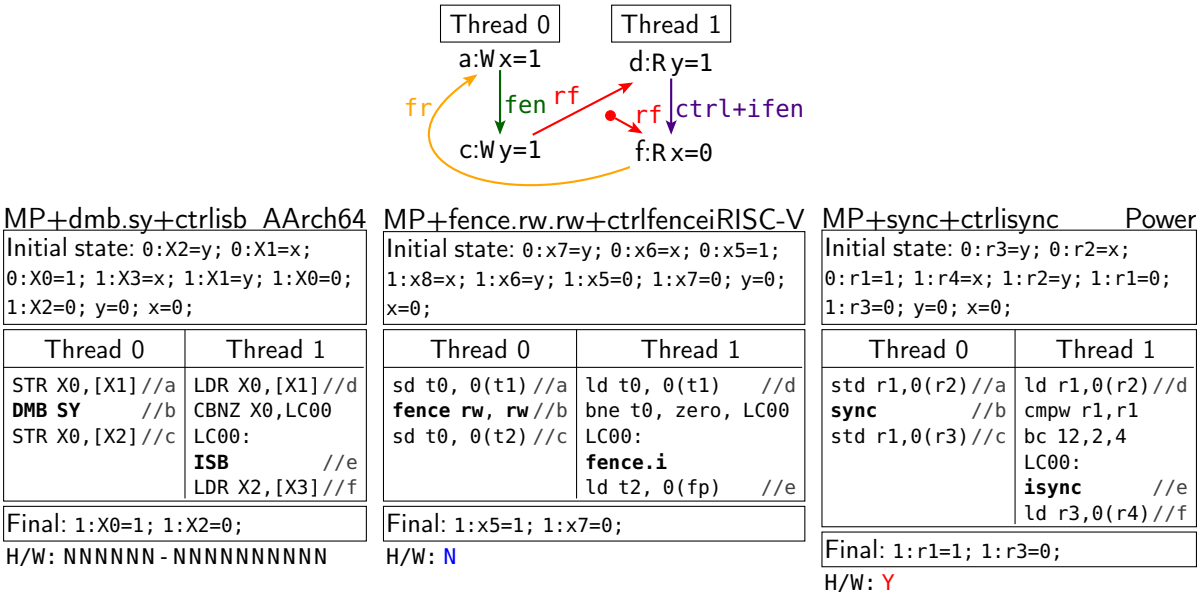


Figure 16.21: Litmus test MP+fen+ctrlifen

where Thread 1 has an instruction barrier. Recall, from the previous subsection, that without the instruction barrier (MP+fen+ctrl) this behaviour is allowed by the Arm and Power architectures. There, the second load of Thread 1 (from x) can be speculatively executed before the first load of Thread 1 (from y), which results in reading 0 from x, and later reading 1 from y. In MP+fen+ctrlifen, because of the instruction barrier, the second load of Thread 1 cannot be fetched before the first load of Thread 1 is completed. Hence, by the time the second load is executed, the stores and memory barrier of Thread 0 has already been performed, and therefore Thread 1 must read 1 from x.

The RISC-V architecture does not guarantee the read-read ordering of the instruction barrier instruction (`fence.i`), and therefore MP+fen+ctrlifen is allowed by the architecture. This decision, made by the RISC-V memory model task group, was motivated by the fact that this effect

is not the original intention of the `fence.i` instruction, and it was speculated that it is possible to implement RISC-V hardware implementations without this side effect. Moreover, the RISC-V architecture includes the `fence r,r` instruction which explicitly forces order between loads, and should be used instead, when such order is desired. The Arm and Power architectures do not include a barrier similar to the RISC-V `fence r,r`.

16.1.6 Write forwarding

As discussed in §16.1.4, the Arm, Power, and RISC-V architectures do not allow speculative writes to be observable by other threads. However, they do all allow speculative writes to be observed by the same thread, as discussed in this subsection.

Consider PPOCA (preserved program order control address, canonical name MP+fen+ctrl-

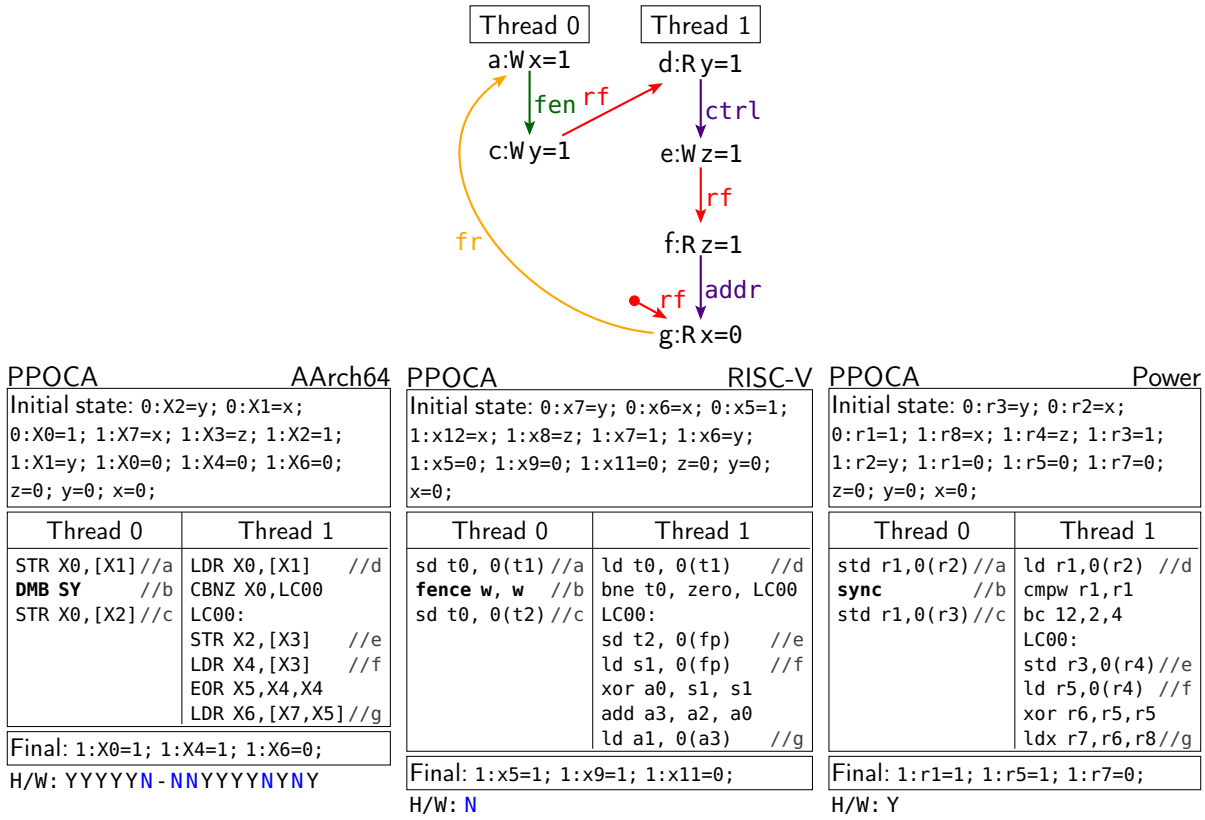


Figure 16.22: Litmus test PPOCA

rfi-addr, Fig. 16.22). In this version of MP, Thread 0 is forced to execute in-order by a fence, and Thread 1's loads from y (event d) and x (event g) are separated, in program-order, by a store to z (event e) and a load from z (event f) on which g has an address dependency. In addition, all the memory access instructions of Thread 1, except for the load from y, are on a conditional branch, the condition of which is determined by the value loaded from y (i.e. the memory accesses to z and x have a control dependency from the load of y).

A hardware implementation may speculate that the condition of the conditional branch in Thread 1 will be successful, and satisfy the load from z with the write of 1 to z by the store that precedes it (*write forwarding*). As the load from z is by the same thread as the store to z, and as they are both on the same speculated branch, discarding these memory accesses, if needed, is relatively simple (i.e., it does not involve coordinating a roll-back between multiple modules). The hardware implementation can then, speculatively, resolve the address dependency from the load of z to the load of x, speculatively perform the load from x, before any of the memory writes of Thread 0 take effect, and satisfy it with the initial value 0. The execution can continue with

Thread 0 executing in-order (as required by the fence), followed by satisfying the load from y of Thread 1 with the value that Thread 1 wrote to this location, which validates the speculation of the conditional branch. This behaviour is indeed allowed by Arm, Power, and RISC-V.

In contrast to PPOCA, the very similar PPOAA (preserved program order address address,

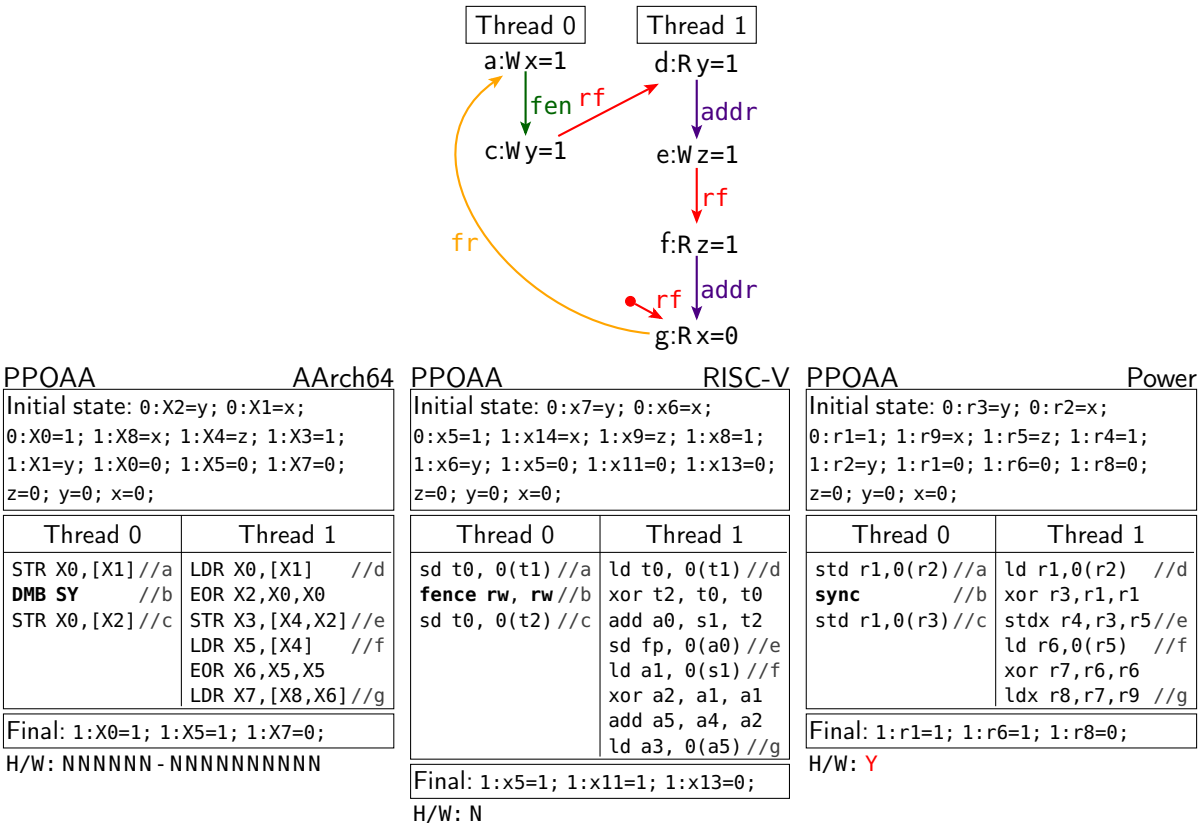


Figure 16.23: Litmus test PPOAA

canonical name MP+fen+addr-rfi-addr, Fig. 16.23) is forbidden by Arm, Power, and RISC-V. In PPOAA there is no control dependency from the load from y to all the memory accesses that succeed it in program-order, instead there is an address dependency from the load from y to the succeeding store to z. As observable value speculation is forbidden by the three architectures, the write to z is not allowed to be speculated in this case, which prevents the load from z from being satisfied early, which prevents the address dependency from that load to the load from x from being resolved, which prevents the load from x from executing out-of-order with the load from y.

Write forwarding from store-exclusive instructions is discussed later in §16.1.15

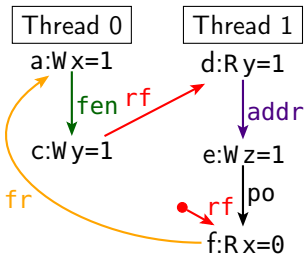
16.1.7 Speculative execution - restarts

§16.1.4 discussed speculative execution of instructions as a result of conditional or indirect branches. In those cases, when the target of the branch instruction is finally computed, any speculated instruction that is not on the correct branch is discarded. In the kind of speculation this subsection discusses, miss-speculated instructions are not discarded (as they are on the correct execution path), they are instead rolled-back (restarted). For those instructions, what is being speculated is the validity of doing their execution out-of-order with previous instructions.

It is a property of the Armv8-A, Power, and RISC-V architectures that a restart can not be observed directly. That is, when a hardware implementation restarts an instruction it has to leave no trace of the spurious execution, except perhaps for debug and performance-counter facilities. This is reflected in the operational models of those architecture by the fact that every

behaviour that is allowed by the models can be exhibited by an execution that does not restart any instruction.

Although the architectures do not allow restarts to be observed directly, the fact that a hardware implementation can restart an instruction is observable, as demonstrated by the MP+fen+addr-po litmus test, in which the two loads of Thread 1 are separated in program-



MP+dmb.sy+addr-po AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X6=x; 1:X4=z; 1:X3=1; 1:X1=y; 1:X0=0; 1:X5=0; z=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X0,[X2] //c	STR X3,[X4,X2] //e
	LDR X5,[X6] //f

Final: 1:X0=1; 1:X5=0;

H/W: NYY - - - - NNYYY - NY - Y

MP+fence.rw.rw+addr-po RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x12=x; 1:x9=z; 1:x8=1; 1:x6=y; 1:x5=0; 1:x11=0; z=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t0, 0(t2) //c	add a0, s1, t2
	sd fp, 0(a0) //e
	ld a1, 0(a2) //f

Final: 1:x5=1; 1:x11=0;

H/W: -

MP+sync+addr-po Power

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r7=x; 1:r5=z; 1:r4=1; 1:r2=y; 1:r1=0; 1:r6=0; z=0; y=0; x=0;

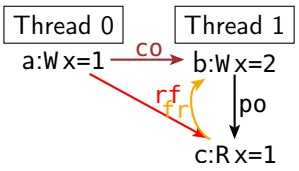
Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r1,0(r3) //c	stwx r4,r3,r5 //e
	ld r6,0(r7) //f

Final: 1:r1=1; 1:r6=0;

H/W: -

Figure 16.24: Litmus test MP+fen+addr-po

order, by a store to z (event e) with an address dependency from the load of y (d). As the load from x has no dependency, a hardware implementation can execute this load out-of-order with the preceding store to z and load from y. However, such execution of the load from x must be speculative, as the hardware implementation does not yet know the memory location that the preceding store will access (due to the address dependency). If the location that the preceding store accesses turns out to be x, the out-of-order execution of the load from x would be a violation of coherence, similar to CoWR, and would have to be re-executed (i.e. restarted). Hence,



CoWR AArch64

Initial state: 0:X1=x; 0:X0=1; 1:X1=x; 1:X0=2; 1:X2=0; x=0;

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

Final: 1:X2=1; x=2;

H/W: YNNNNN - NNNNNNNNNN

CoWR RISC-V

Initial state: 0:x6=x; 0:x5=1; 1:x6=x; 1:x5=2; 1:x7=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	sd t0, 0(t1) //b
	ld t2, 0(t1) //c

Final: 1:x7=1; x=2;

H/W: Y

CoWR Power

Initial state: 0:r2=x; 0:r1=1; 1:r2=x; 1:r1=2; 1:r3=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	std r1,0(r2) //b
	ld r3,0(r2) //c

Final: 1:r3=1; x=2;

H/W: Y

Figure 16.25: Litmus test CoWR

exhibiting MP+fen+addr-po by a hardware implementation makes the fact that the hardware implementation can do such speculation observable.

Soft restarts

When a hardware implementation determines that an instruction has to be restarted, in most cases in RISC architectures, it does not matter (i.e. it is unobservable) whether the instruction is completely discarded and re-fetched, or if parts of the spurious execution, that were not inconsistent, are kept. For example, if a load instruction has to be re-executed because it was executed out-of-order with a preceding store that turned out to be to the same location, the part of the execution of the load instruction that computes the memory location that the load reads from would result in the same location. Hence, discarding this part of the execution and re-executing it, or keeping the result of the computation from the spurious execution would have the same effect.

However, in some cases, keeping parts of the spurious execution, even if those are not inconsistent, is observable. For example, the Arm and Power architectures include a load instruction that can read multiple values from adjacent locations in memory, and store each value in a separate register (AArch64 `ldp`, and Power `lmw`). Consider MP+fen+pos-si124-ctrlifen. In this version

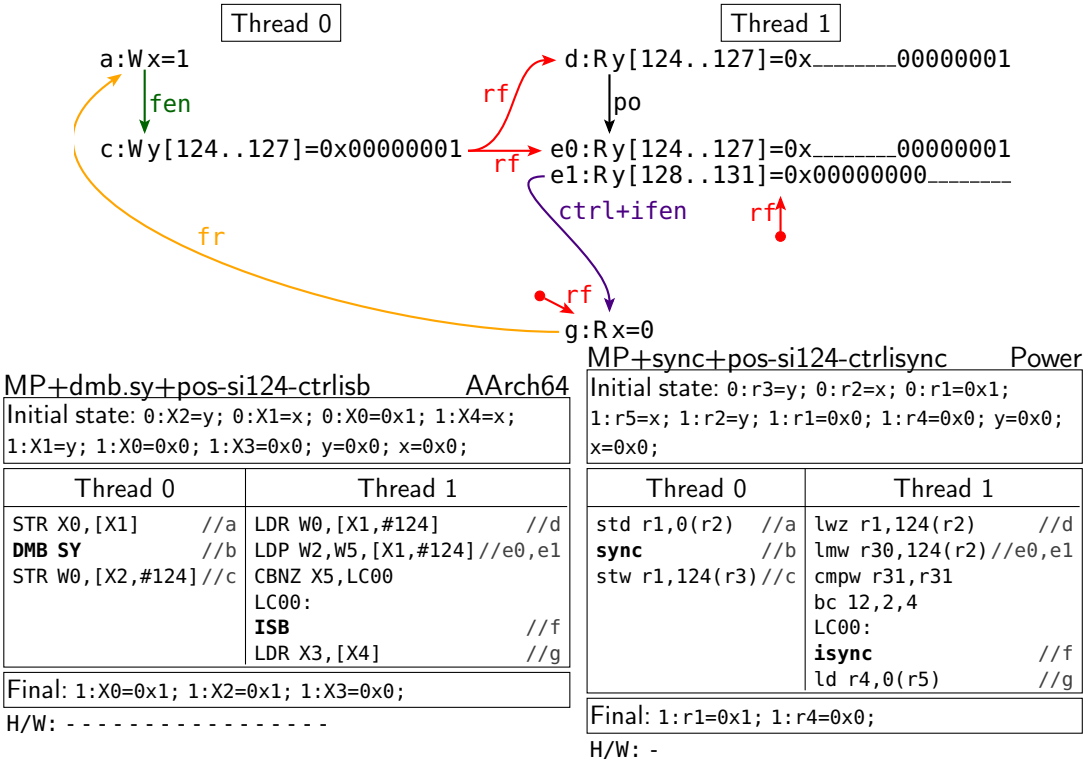
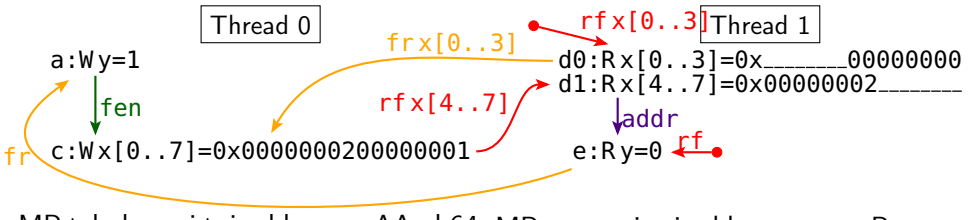


Figure 16.26: Litmus test MP+fen+pos-si124-ctrlifen

of MP the y location is allocated such that $y[0]$, the first byte of the allocated region, is exactly the beginning of a cache line, and the size of the allocated region spans multiple cache lines. The y location is then used in the test to access two consecutive 4-byte words, $y[124..127]$ and $y[128..131]$. Those words are properly aligned (see §16.2.3) and, assuming 128 is a multiple of the cache line size (which holds for all the machines we tested, but is not architecturally mandated), each word is in a different cache line. Note that we take care to place those words in this fashion (on a cache boundary) to make it more likely for a hardware implementation to exhibit the interesting behaviour; the architectures allow this behaviour even when the words are not on a cache boundary.

With that, Thread 0 is forced to execute in-order by a fence, and Thread 1's loads from $y[124..127]$ (event d) and x (event g) are separated, in program-order, by a load instruction that access two locations: $y[124..127]$ (event $e1$) and $y[128..131]$ (event $e0$), a conditional branch that is determined by $e0$, and an instruction fence (event f).

The MP+fen-si+si-addr litmus test demonstrates that hardware implementations may per-



MP+dmb.sy-si+si-addr AArch64

Initial state: 0:X2=0x200000001; 0:X1=0x1; 0:X6=y; 0:X5=x; 1:X6=y; 1:X5=x; x=0x0; y=0x0;

Thread 0	Thread 1
STR X1,[X6] //a	LDP W1,W2,[X5] //d0,d1
DMB SY //b	EOR W3,W1,W1
STR X2,[X5] //c	LDR X4,[X6,W3,SXTW] //e

Final: 1:X1=0x0; 1:X2=0x2; 1:X4=0x0;

H/W: - - - - - - - - - - - - - - - -

MP+sync-si+si-addr Power

Initial state: 0:r5=y; 0:r4=x; 0:r1=0x3; 0:r2=0x100000002; 1:r5=y; 1:r4=x; x=0x0;

Thread 0	Thread 1
stw r1,0(r5) //a	lmw r30,0(r4) //d0,d1
sync //b	xor r3,r30,r30
std r2,0(r4) //c	lwzx r2,r3,r5 //e

Final: 1:r2=0x0; 1:r30=0x0; 1:r31=0x2;

H/W: -

Figure 16.27: Litmus test MP+fen-si+si-addr

form the different memory reads of load-pair/multiple separately, and make the first-satisfied value available in the appropriate register before reading the other memory location. For a hardware implementation to exhibit MP+fen+pos-si124-ctrlifen it must also be able to perform only part of the load-pair/multiple in a non-speculative way, while the other part was not performed yet, or performed speculatively.

For a hardware implementation to exhibit MP+fen+pos-si124-ctrlifen it must perform the part of the intermediate load of Thread 1 that reads from $y[128..131]$ before the memory writes of Thread 0 take effect, as the load from x has a control dependency on this part of the load, and there is an instruction fence between the load from x and the conditional branch (see Instruction Barrier above). In addition, the hardware implementation must perform the firs load of Thread 1 after the memory writes of Thread 0 take effect, and the part of the intermediate load that reads from $y[124..127]$ can only be speculative before that, as it access the same location as the first load.

Performing a misaligned load (see §16.2.3) is somewhat similar to load-pair/multiple. The key difference is that the latter loads into multiple distinct parts of the register state, that can be depended or independent. Since a misaligned load loads into a single register, soft restart is not observable, as the value that is loaded from memory is only available to other instructions (i.e. written to the register) after all the memory reads of the load instruction are completed.

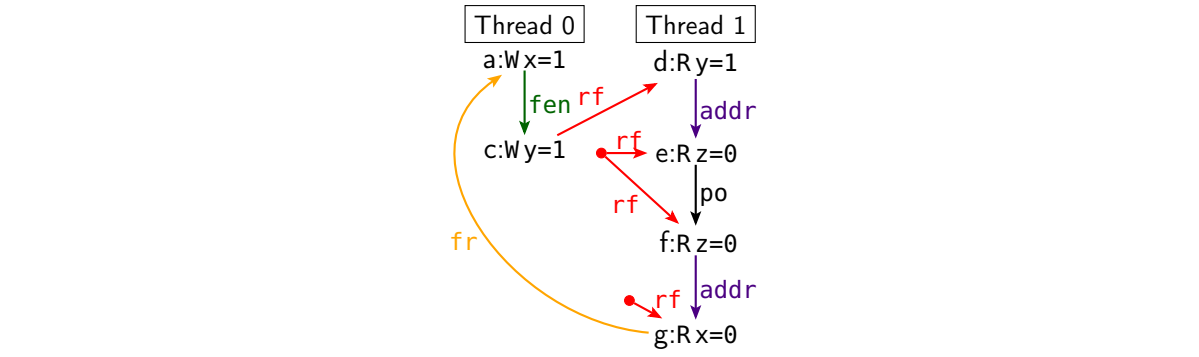
16.1.8 Satisfy same address reads out-of-order

As was mentioned in §16.1.1, it is possible to execute out-of-order two load instructions that access the same memory location, without violating coherence. RSW (read same write, Fig. 16.28) discriminates between architectures that allow such reordering and those that do not.

RSW is a variant of MP in which the stores of Thread 0 are forced to execute in-order by a fence, and the loads from y and x of Thread 1 are separated by two loads from z, of which the first has an address dependency from the load from y, and the second has an address dependency to the load from x.

The RSW behaviour is coherent, with the total orders over x, y, and z, in Fig. 16.29.

A hardware implementation may exhibit this behaviour by satisfying the second load from z (f) with the initial value 0 very early, calculating the address of the load from x (g) (i.e. resolving the address dependency from f to g) and satisfying the load from x with the initial value 0, before the memory writes of Thread 0 take effect. The hardware implementation can



RSW AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X8=x; 1:X4=z; 1:X1=y; 1:X0=0; 1:X7=0; z=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X0,[X2] //c	LDR X3,[X4,X2] //e
	LDR X5,[X4] //f
	EOR X6,X5,X5
	LDR X7,[X8,X6] //g

Final: 1:X0=1; 1:X7=0;

H/W: YYYYYN-NNYYYYNYNY

RSW RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x14=x; 1:x9=z; 1:x6=y; 1:x5=0; 1:x13=0; z=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t0, 0(t2) //c	add a0, s1, t2
	ld fp, 0(a0) //e
	ld a1, 0(s1) //f
	xor a2, a1, a1
	add a5, a4, a2
	ld a3, 0(a5) //g

Final: 1:x5=1; 1:x13=0;

H/W: N

RSW Power

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r9=x; 1:r5=z; 1:r2=y; 1:r1=0; 1:r8=0; z=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r1,0(r3) //c	ldx r4,r3,r5 //e
	ld r6,0(r5) //f
	xor r7,r6,r6
	ldx r8,r7,r9 //g

Final: 1:r1=1; 1:r8=0;

H/W: Y

Figure 16.28: Litmus test RSW

$$init(x) < g < a$$
$$init(y) < c < d$$
$$init(z) < e < f$$

Figure 16.29: Coherence orders of RSW. Here, $init(L)$ is the notional write that sets the initial value for location L.

then complete the execution by performing all the instructions of Thread 0 in-order, satisfying the load from y (d) with the value written by the store to y (c), and satisfying the first load from z (e) with the initial value 0. Note that even though that e and f are to the same location (z), they were satisfied out-of-order. The hardware implementation can be sure that the coherence of z is not violated by allowing this behaviour only when Thread 1's cache line that holds z has not been released between the two reads of z (f and e).

Tracking the cache line in this way, in order to guarantee that there is no coherence violation, excludes behaviours like RDW (read different writes, Fig. 16.30), which is similar to RSW, except that the two reads of z return values from different writes, even though RDW does not violate coherence. Moreover, when the second load, in a sequence of two loads from the same location, is satisfied first, it must be regarded as speculative until the first load is satisfied.

16.1.9 Write forwarding from a non-speculative write

In §16.1.6 PPOCA was used to demonstrate write-forwarding of writes to program-order-following reads. There, a speculative write that must not be observed by other threads while speculative, is allowed to be observed by the thread that executes it (even while it is speculative). In such forwarding, the read to which the write is being forwarded to must also be regarded by the hardware implementation as speculative, at least until the write that is being forwarded to

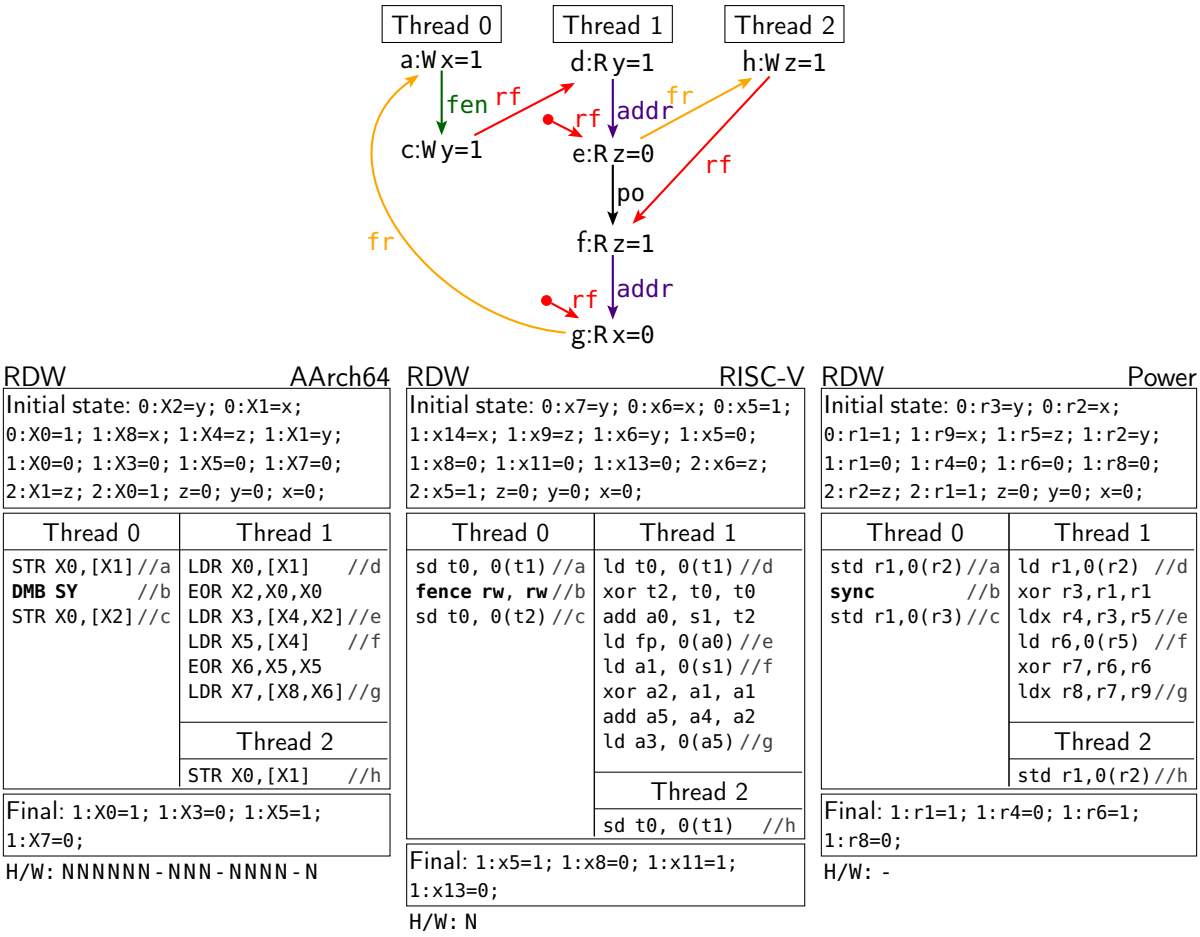


Figure 16.30: Litmus test RDW

it is determined to be non-speculative. In PPOCA this is obvious, as both the write and the read are on the same conditional branch, but in other cases the write could be speculative due to other reasons that do not affect the read (see §16.1.7), other than by the forwarding from the write.

A hardware implementation might require to prevent a write from being observed by other threads at a given state for reasons other than the write being speculative, most notably, to prevent coherence violations. In such circumstances a hardware implementation may be able to forward the write without having to regard the read to which the write is forwarded to as speculative. Consider MP+fen+data-wsi-rfi-ctrlifen. The first write to z (e) has a data dependency on the preceding load from y and therefore that write can not be performed before the the load from y is completed. The second write to z (f) has no dependencies, but the hardware implementation can not perform it before the first write to z, as that can lead to violation of coherence, similar to CoWW. Although the second write to z can not be performed before the first write to z, it can be forwarded to the succeeding load from z, without violating coherence, because the second write to z obscures the first write to z for the succeeding load. Unlike the case of write-forwarding in PPOCA, here the write that is being forwarded is not speculative, as it does not follow a speculative branch and is not amenable for restarts. Hence, after forwarding f to g the hardware implementation can compute the condition of the conditional branch that feeds from g, and it can be sure that the result will not change due to speculations. As the conditional branch is resolved the instruction-fence that succeeds the conditional branch no longer has any effect (in the context of this execution) and the succeeding read m can be satisfied,

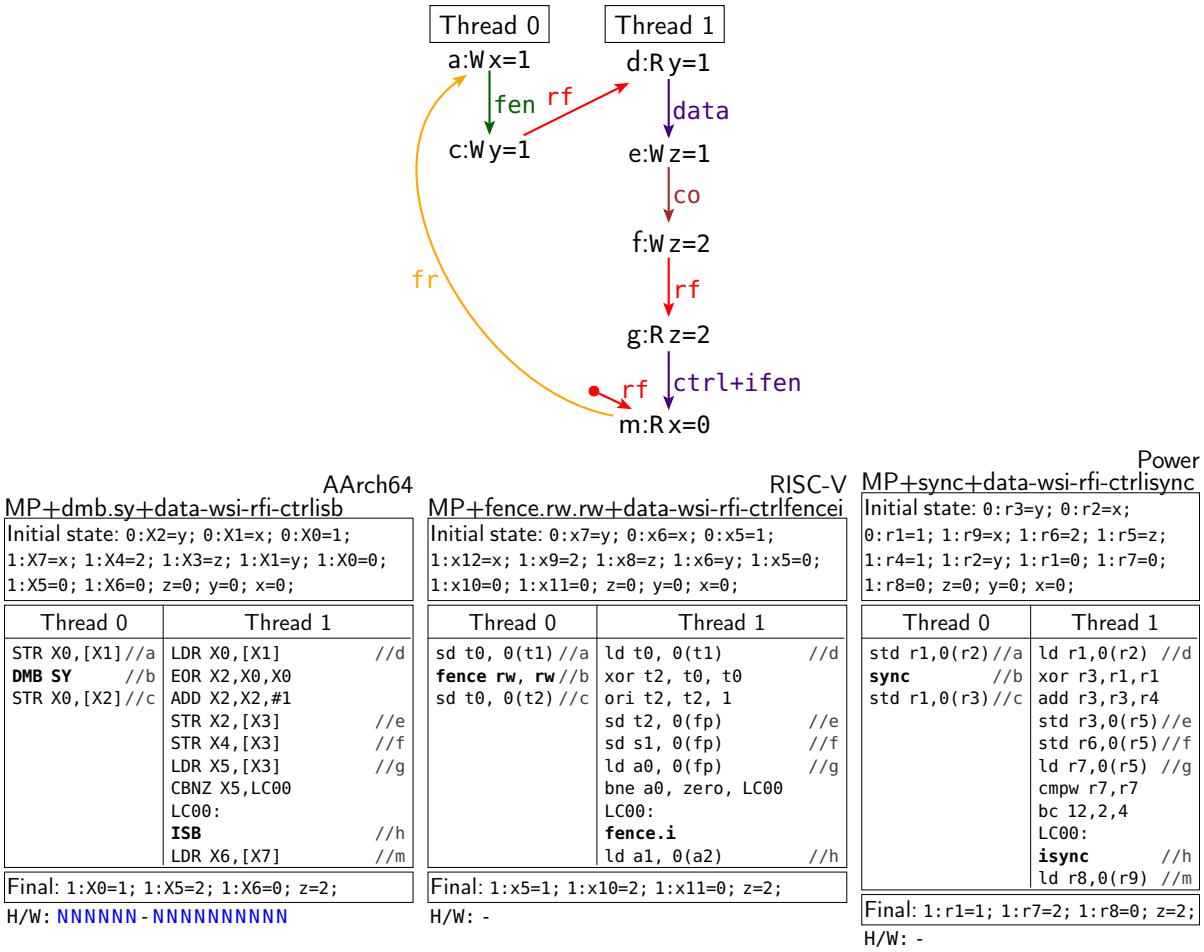


Figure 16.31: Litmus test MP+fen+data-wsi-rfi-ctrlifen

before any of the stores of Thread 0 take effect.

This behaviour is allowed by the Arm and RISC-V architectures, though since the RISC-V instruction-fence is weaker than the Arm instruction fence, for RISC-V this is not much different from PPOCA. That is, the same behaviour can be explained for RISC-V without using the fact that forwarding of f to g is non-speculative. The similar test S+fen+data-wsi-rfi-ctrl, which eliminates the instruction-fence, and changes the last memory access of MP+fen+data-wsi-rfi-ctrlifen from a load to a store, is allowed by both RISC-V and Arm because they allow the control dependency in Thread 1 to be resolved by non-speculative forwarding.

Sarkar et al.'s PLDI11 memory model of the Power architecture [**?**] does not allow the behaviour of MP+fen+data-wsi-rfi-ctrlifen (and the similar S+fen+data-wsi-rfi-ctrl), as none of the Power hardware implementations exhibit this behaviour and it is not clear whether there is any benefit in allowing it. If need be, the PLDI11 model could be adapted to allow this behaviour by relaxing the condition of the Thread transition "Commit in-flight instruction" [**?**, p. 7]. In particular, condition 4 of this transition can be changed to exclude loads if the load is satisfied by forwarding from a store that cannot be restarted and all program-order previous instructions which might access the load's address are committed.

An alternative explanation for MP+fen+data-wsi-rfi-ctrlifen is discussed later, in §16.1.12.

16.1.10 Multi-step read satisfaction

In 2012, while investigating the relationship between the Arm architecture and Sarkar et al.'s PLDI11 memory model of the Power architecture [**?**], the authors of [**?**, **?**] discovered some

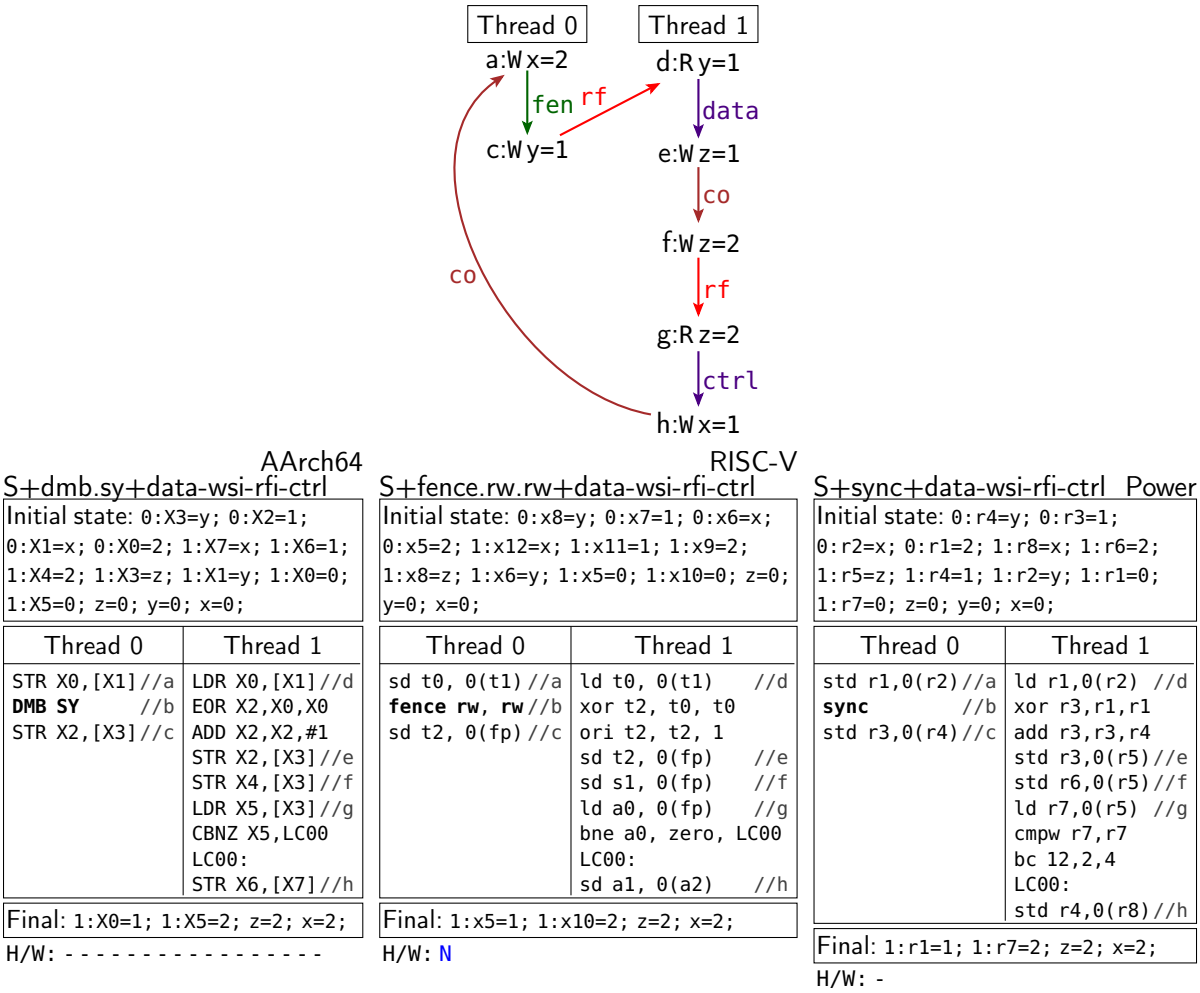
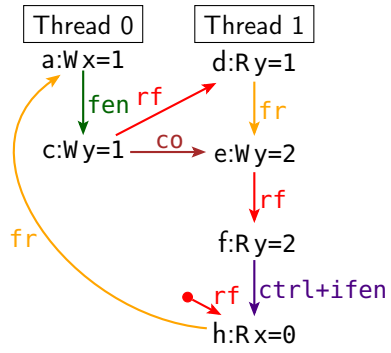


Figure 16.32: Litmus test S+fen+data-wsi-rfi-ctrl

inconsistencies between that model and the observed behaviour of some Armv7-A hardware implementations. Most notably, the Qualcomm APQ8060 SoC (dual-core Scorpion CPU architecture, Armv7-A) exhibited MP+fen+fri-rfi-ctrlifen, that is not allowed by the PLDI11 model. This behaviour, and other litmus tests that the APQ8060 exhibited and are not allowed by the PLDI11 model, were discussed with Arm and Qualcomm designers. Most of those behaviours were concluded to be bugs in the APQ8060 implementation (e.g. a coherence violation, of CoRR), but the MP+fen+fri-rfi-ctrlifen behaviour was deemed to be an intended allowed behaviour of the Armv7-A architecture, and also, later, of the Armv8-A architecture. In those discussions, the behaviour of MP+fen+fri-rfi-ctrlifen was explained by the use of a hardware implementation queue that can hold write events and unsatisfied read request events, that are kept in order if they access the same location (similar to a write buffer, with the addition of read request events). When such a hardware implementation is performing a memory access, an event is placed in the queue, which is served (and removed from the queue) when the appropriate cache line is made available. In addition, a read that is immediately preceded by a write to the same location, can be satisfied by that write. If such a read is completely satisfied by the write, it is removed from the queue, but if it is not completely satisfied (see §16.2), the hardware implementation has to ensure single-copy atomicity (see §16.2.3), which can be enforced in the model by, for example, swapping the position in the queue of the write with the read that it partially satisfied.

This queue can be used to explain MP+fen+fri-rfi-ctrlifen as follows. The hardware implementation places the read d in the queue; it then forwards the write e to the succeeding read f, and places e in the queue. The fact that the queue preserves the order of events to the same



AArch64

MP+dmb.sy+fri-rfi-ctrlisb

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X5=x; 1:X2=2; 1:X1=y; 1:X0=0; 1:X3=0; 1:X4=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	STR X2,[X1] //e
STR X0,[X2] //c	LDR X3,[X1] //f
	CBNZ X3,LC00
	LC00:
	ISB //g
	LDR X4,[X5] //h

Final: 1:X0=1; 1:X3=2; 1:X4=0; y=2;

H/W: NNNNNN - NNNNNNNNNY

RISC-V

MP+fence.rw.rw+fri-rfi-ctrlfencei

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x10=x; 1:x7=2; 1:x6=y; 1:x5=0; 1:x8=0; 1:x9=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	sd t2, 0(t1) //e
sd t0, 0(t2) //c	ld fp, 0(t1) //f
	bne fp, zero, LC00
	LC00:
	fence.i
	ld s1, 0(a0) //g

Final: 1:x5=1; 1:x8=2; 1:x9=0; y=2;

H/W: -

Power

MP+sync+fri-rfi-ctrlisync

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r6=x; 1:r3=2; 1:r2=y; 1:r1=0; 1:r4=0; 1:r5=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	std r3,0(r2) //e
std r1,0(r3) //c	ld r4,0(r2) //f
	cmpw r4,r4
	bc 12,2,4
	LC00:
	isync //g
	ld r5,0(r6) //h

Final: 1:r1=1; 1:r4=2; 1:r5=0; y=2;

H/W: N

Figure 16.33: Litmus test MP+fen+fri-rfi-ctrlifen

location guarantees that d will always precede e in the queue, and therefore coherence will not be violated by d getting its value from e, or a coherence-after write. Hence Thread 1 can regard e as if it has been performed non-speculatively, even though it is still in the queue and cannot yet be observed by Thread 0, and even though the program-order preceding read from the same location (which is also in the queue) has not been satisfied yet. At this point the hardware implementation can compute the condition of the conditional branch that feeds from f, and be sure that the result will not change due to speculations (there are no potential coherence violations that will require restarts). As the conditional branch is resolved, the instruction-fence that succeeds the conditional branch no longer has any effect (in the context of this execution) and the succeeding read h can be satisfied, before any of the stores of Thread 0 take effect. This hardware implementation explanation is the inspiration behind the storage subsystem of the Flowing memory-model that will be discussed in **??**.

MP+fen+fri-rfi-ctrlifen is another behaviour that the PLDI11 memory model of the Power architecture does not allow, as none of the Power hardware implementations exhibit this behaviour and it was not clear whether there was any benefit in allowing it. If need be, this model could be adapted to allow this behaviour by adding to each thread a queue that can hold read and write events before they are propagated to the storage.

As Alglave et al. [**?**, p. 53-54] observed, MP+fen+fri-rfi-ctrlifen can also be explained by write-forwarding from a non-speculative write (discussed above §16.1.9), from e to f. However, other behaviours that the queue enables, such as MP+fen+fri-[ws-rf]-ctrlifen (for non-multi-copy-atomic architectures), and the mixed-size LB+data+pod-rfi-pos-data+MIX1, can not be explained by write-forwarding from a non-speculative write. And the opposite is also true, write-forwarding from a non-speculative write enables MP+fen+data-wsi-rfi-ctrlifen, which can not be explained by the queue.

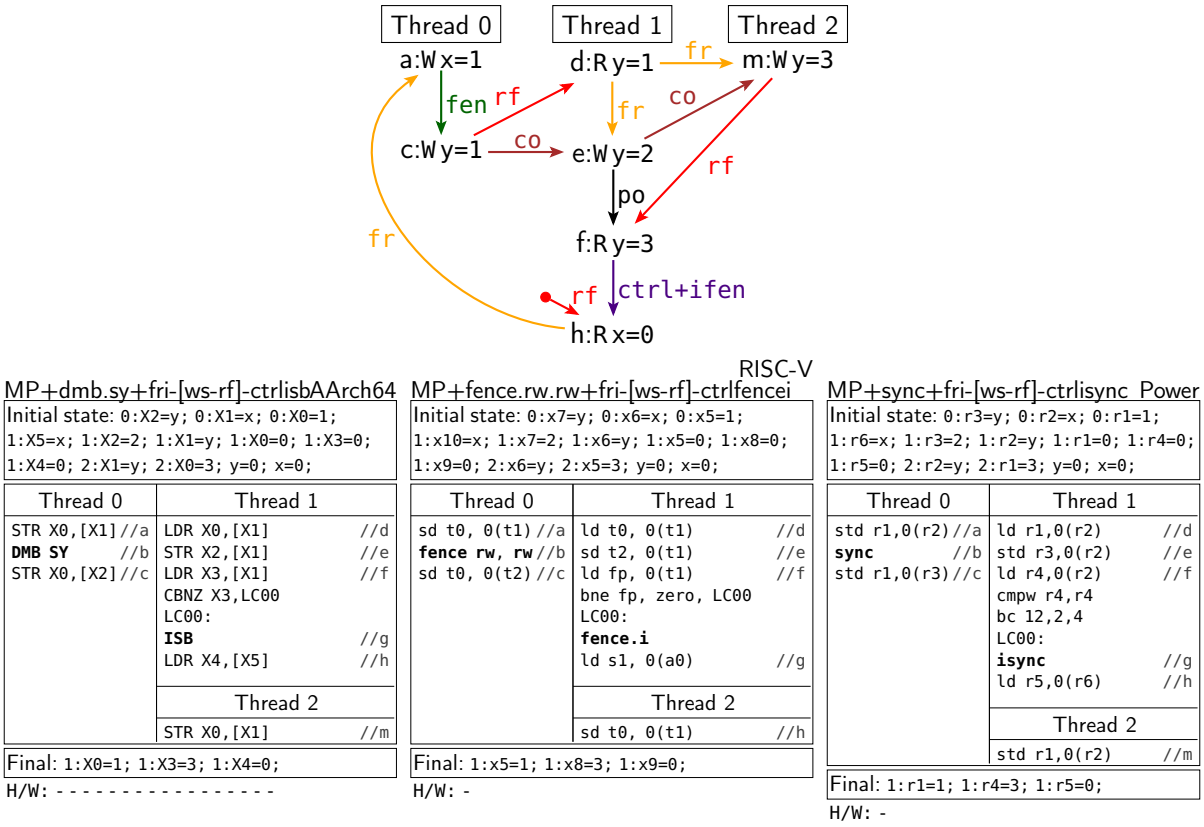


Figure 16.34: Litmus test MP+fen+fri-[ws-rf]-ctrlifen

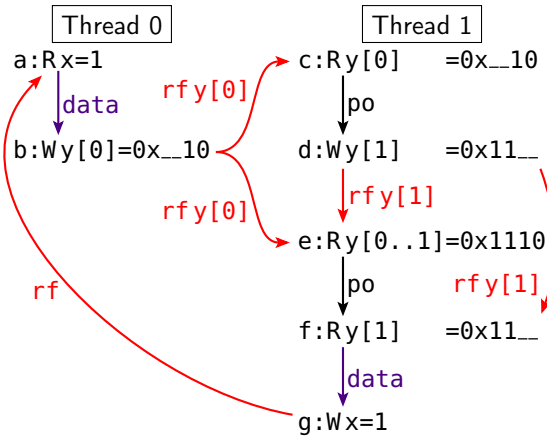
16.1.11 Detour

In MP+fen+fri-[ws-rf]-ctrlifen, from the previous subsection, the memory accesses e and f of Thread 1 are ordered by the external event m from Thread 2. This is a relatively simple case of a *detour*: two events from the same thread that are ordered by external events [**?**]. A more complex example is MP+fen+fri-[rf-addr-rf]-addr. Here, the memory accesses e and f of Thread 1, are to different locations, and are ordered by memory accesses from Thread 2, that are ordered by an address dependency between themselves. This can get even more complicated if the events of Thread 2 are also ordered by any number of detours.

Recursive detours can be tricky to get right axiomatically, in a non-multi-copy atomic architecture, and in fact were one of the reasons given by Arm staff for changing to a multi-copy atomic architecture.

16.1.12 Write subsumption

It is possible for a hardware implementation to detect two (or more) memory stores to the same location, and discards the program order preceding one, if there are no loads from the same location, from the same thread, between the stores (i.e., the succeeding write subsumes the preceding one). For example, consider S+fen+data-wsi, where Thread 1 performs two adjacent stores to x. Without write subsumption this behaviour cannot be exhibited because the write of 2 to x (f) can only be performed after the write of 1 to x (e) has been performed, to ensure that those two writes are not observed out of order; in addition, the first write of 1 to x can only be performed after the read from y (d), that feeds into its data, has been performed, and the read can get its value from the write to y of Thread 0 (c) only after the write of 3 to x of Thread 0



LB+data+pod-rfi-pos-data+MIX1 AArch64

Initial state: 0:X4=0x10; 0:X2=y; 0:X1=x; 0:X0=0x1; 1:X6=0x0; 1:X2=0x11; 1:X3=0x0; 1:X4=x; 1:X1=y; x=0x0; y=0x0;

Thread 0	Thread 1
LDR X0,[X1] //a	LDRB W0,[X1] //c
EOR X3,X0,X0	STRB W2,[X1,#1] //d
ADD X3,X3,X4	LDRH W3,[X1] //e
STRB W3,[X2] //b	LDRB W5,[X1,#1] //f
	EOR X6,X5,X5
	ADD X6,X6,#1
	STR X6,[X4] //g

Final: 0:X0=0x0; 1:X0=0x10; 1:X3=0x1110; 1:X5=0x11; y=0x1110;

H/W: - - - - - - - - - - - - - - - -

LB+data+pod-rfi-pos-data+MIX1 RISC-V

Initial state: 0:x4=0x10; 0:x8=y; 0:x6=x; 0:x5=0x0; 1:x11=x; 1:x7=0x11; 1:x6=y; 1:x5=0x0; 1:x8=0x0; y=0x0; x=0x0;

Thread 0	Thread 1
ld t0, 0(t1) //a	lb t0, 0(t1) //c
xor t2, t0, t0	sb t2, 1(t1) //d
or t2, t2, tp	lh fp, 0(t1) //e
sb t2, 0(fp) //b	lb s1, 1(t1) //f
	xor a0, s1, s1
	ori a0, a0, 1
	sd a0, 0(a1) //g

Final: 0:x5=0x1; 1:x5=0x10; 1:x8=0x1110; 1:x9=0x11; y=0x1110;

H/W: -

LB+data+pod-rfi-pos-data+MIX1 Power

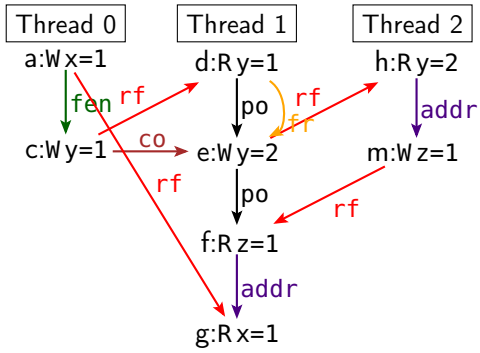
Initial state: 0:r5=y; 0:r4=0x10; 0:r2=x; 0:r1=0x0; 1:r8=x; 1:r7=0x1; 1:r3=0x11; 1:r2=y; 1:r1=0x0; 1:r4=0x0; y=0x0; x=0x0;

Thread 0	Thread 1
ld r1,0(r2) //a	lbz r1,0(r2) //c
xor r3,r1,r1	stb r3,1(r2) //d
add r3,r3,r4	lhz r4,0(r2) //e
stb r3,0(r5) //b	lbz r5,1(r2) //f
	xor r6,r5,r5
	add r6,r6,r7
	std r6,0(r8) //g

Final: 0:r1=0x1; 1:r1=0x10; 1:r4=0x1110; 1:r5=0x11; y=0x1110;

H/W: -

Figure 16.35: Litmus test LB+data+pod-rfi-pos-data+MIX1



MP+dmb.sy+fri-[rf-addr-rf]-addr AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X7=x; 1:X4=z; 1:X2=2; 1:X1=y; 1:X0=0; 1:X3=0; 1:X6=0; 2:X4=z; 2:X3=1; 2:X1=y; 2:X0=0; z=0; y=0; x=0;

Thread 0	Thread 1	Thread 2
STR X0,[X1] //a	LDR X0,[X1] //d	LDR X0,[X1] //h
DMB SY //b	STR X2,[X1] //e	EOR X2,X0,X0
STR X0,[X2] //c	LDR X3,[X4] //f	STR X3,[X4,X2] //m
	EOR X5,X3,X3	
	LDR X6,[X7,X5] //g	

Final: 1:X0=1; 1:X3=1; 1:X6=1; 2:X0=2; y=2;

H/W: - - - - - - - - - - - - - - - -

MP+fence.rw.rw+fri-[rf-addr-rf]-addr RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x12=x; 1:x9=z; 1:x7=2; 1:x6=y; 1:x5=0; 1:x8=0; 1:x11=0; 2:x9=z; 2:x8=1; 2:x6=y; 2:x5=0; z=0; y=0; x=0;

Thread 0	Thread 1	Thread 2
sd t0, 0(t1) //a	ld t0, 0(t1) //d	ld t0, 0(t1) //h
fence rw, rw //b	sd t2, 0(t1) //e	xor t2, t0, t0
sd t0, 0(t2) //c	ld fp, 0(s1) //f	add a0, s1, t2
	xor a0, fp, fp	sd fp, 0(a0) //m
	add a3, a2, a0	
	ld a1, 0(a3) //g	

Final: 1:x5=1; 1:x8=1; 1:x11=1; 2:x5=2; y=2;

H/W: -

MP+sync+fri-[rf-addr-rf]-addr Power

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r8=x; 1:r5=z; 1:r3=2; 1:r2=y; 1:r1=0; 1:r4=0; 1:r7=0; 2:r5=z; 2:r4=1; 2:r2=y; 2:r1=0; z=0; y=0; x=0;

Thread 0	Thread 1	Thread 2
std r1,0(r2) //a	ld r1,0(r2) //d	ld r1,0(r2) //h
sync //b	std r3,0(r2) //e	xor r3,r1,r1
std r1,0(r3) //c	ld r4,0(r5) //f	stdx r4,r3,r5 //m
	xor r6,r4,r4	
	ldx r7,r6,r8 //g	

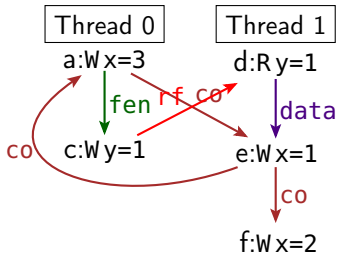
Final: 1:r1=1; 1:r4=1; 1:r7=1; 2:r1=2; y=2;

H/W: -

Figure 16.36: Litmus test MP+fen+fri-[rf-addr-rf]-addr

(a) has been performed, because of the fence between the two writes, and therefore the write of 3 to x (a) cannot be coherence-after the write of 2 to x (f).

A hardware implementation with the write subsumption optimisation could notice that eventually the write of 1 to x (e) would be overwritten (subsumed) by the program-order succeeding



S+dmb.sy+data-wsi AArch64

Initial state: 0:X3=y; 0:X2=1; 0:X1=x; 0:X0=3; 1:X4=2; 1:X3=x; 1:X1=y; 1:X0=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X2,[X3] //c	ADD X2,X2,#1
	STR X2,[X3] //e
	STR X4,[X3] //f

Final: 1:X0=1; y=1; x=2;

H/W: NNN - - - - NNNNN - NN - N

S+fence.rw.rw+data-wsi RISC-V

Initial state: 0:x8=y; 0:x7=1; 0:x6=x; 0:x5=3; 1:x9=2; 1:x8=x; 1:x6=y; 1:x5=0; 1:x10=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t2, 0(fp) //c	ori t2, t2, 1
	sd t2, 0(fp) //e
	sd s1, 0(fp) //f
	ld a0, 0(fp) //g

Final: 1:x5=1; 1:x10=3;

H/W: -

S+sync+data-wsi Power

Initial state: 0:r4=y; 0:r3=1; 0:r2=x; 0:r1=3; 1:r6=2; 1:r5=x; 1:r4=1; 1:r2=y; 1:r1=0; 1:r7=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r3,0(r4) //c	add r3,r3,r4
	std r3,0(r5) //e
	std r6,0(r5) //f
	ld r7,0(r5) //g

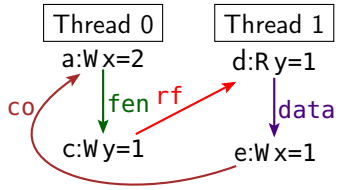
Final: 1:r1=1; 1:r7=3;

H/W: -

Figure 16.37: Litmus test S+fen+data-wsi

write of 2 to x (f), and therefore it could decide to discard the write of 1 to x (e), before performing the read from y (d) that feeds into e's data. After discarding the write of 1 to x (e), it could immediately perform the write of 2 to x (f), before any of the stores of Thread 0 take effect. We have not observed this behaviour on any hardware implementation we have tested.

Notice that S+fen+data appears as a sub-execution in the execution of S+fen+data-wsi. As



S+dmb.sy+data AArch64

Initial state: 0:X3=y; 0:X2=1; 0:X1=x; 0:X0=2; 1:X3=x; 1:X1=y; 1:X0=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X2,[X3] //c	ADD X2,X2,#1
	STR X2,[X3] //e

Final: 1:X0=1; x=2;

H/W: NNNNNN - NNNNNNNNNN

S+fence.rw.rw+data RISC-V

Initial state: 0:x8=y; 0:x7=1; 0:x6=x; 0:x5=2; 1:x8=x; 1:x6=y; 1:x5=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t2, 0(fp) //c	ori t2, t2, 1
	sd t2, 0(fp) //e

Final: 1:x5=1; x=2;

H/W: N

S+sync+data Power

Initial state: 0:r4=y; 0:r3=1; 0:r2=x; 0:r1=2; 1:r5=x; 1:r4=1; 1:r2=y; 1:r1=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r3,0(r4) //c	add r3,r3,r4
	std r3,0(r5) //e

Final: 1:r1=1; x=2;

H/W: N

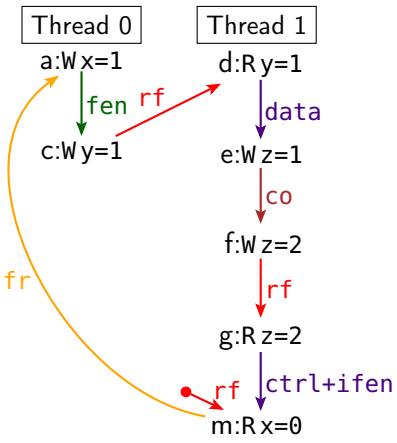
Figure 16.38: Litmus test S+fen+data

S+fen+data is forbidden by any architecture that respects coherence and data dependencies, modelling such an architecture that also allows S+fen+data-wsi in an axiomatic style is challenging. For example, in the Herd axiomatic framework [**?**], the cycle that makes the S+fen+data execution forbidden, will also be present in S+fen+data-wsi, and therefore it is also forbidden.

The Armv8-A, RISC-V and Power architectures all forbid S+fen+data-wsi. The Armv7-A architecture allows the S+fen+data-wsi behaviour, though we have never observed it on Armv7-A

implementations. Originally Armv8-A also allowed S+fen+data-wsi, but it was retroactively changed when Arm published the axiomatic memory model.

It is possible that hardware implementations implement the subsumed write optimisation in ways that are not observable, or ways that are observable but can also be explained by other mechanisms. For example, the behaviour of MP+fen+data-wsi-rfi-ctrlifen, that was discussed in



AArch64

MP+dmb.sy+data-wsi-rfi-ctrlisb

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X7=x; 1:X4=2; 1:X3=z; 1:X1=y; 1:X0=0; 1:X5=0; 1:X6=0; z=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X0,[X2] //c	ADD X2,X2,#1
	STR X2,[X3] //e
	STR X4,[X3] //f
	LDR X5,[X3] //g
	CBNZ X5,LC00
	LC00:
	ISB //h
	LDR X6,[X7] //m

Final: 1:X0=1; 1:X5=2; 1:X6=0; z=2;

H/W: NNNNNN - NNNNNNNNNN

RISC-V

MP+fence.rw.rw+data-wsi-rfi-ctrlfencei

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x12=x; 1:x9=2; 1:x8=z; 1:x6=y; 1:x5=0; 1:x10=0; 1:x11=0; z=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t0, 0(t2) //c	ori t2, t2, 1
	sd t2, 0(fp) //e
	sd s1, 0(fp) //f
	ld a0, 0(fp) //g
	bne a0, zero, LC00
	LC00:
	fence.i
	ld a1, 0(a2) //h

Final: 1:x5=1; 1:x10=2; 1:x11=0; z=2;

H/W: -

Power

MP+sync+data-wsi-rfi-ctrlisync

Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r9=x; 1:r6=2; 1:r5=z; 1:r4=1; 1:r2=y; 1:r1=0; 1:r7=0; 1:r8=0; z=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r1,0(r3) //c	add r3,r3,r4
	std r3,0(r5) //e
	std r6,0(r5) //f
	ld r7,0(r5) //g
	cmpw r7,r7
	bc 12,2,4
	LC00:
	isync //h
	ld r8,0(r9) //m

Final: 1:r1=1; 1:r7=2; 1:r8=0; z=2;

H/W: -

Figure 16.39: Litmus test MP+fen+data-wsi-rfi-ctrlifen

§16.1.9, can also be explained by the write e being subsumed by the write f before the read d is satisfied.

A minor variation of MP+fen+data-wsi-rfi-ctrlifen, MP+fen+data-[rf-fr]-rfi-ctrlifen, is allowed by write forwarding from a non-speculative write, and is not allowed (by architectures with a strong instruction-fence, such as Arm and Power) by write subsumption. The addition of a third thread that reads the first value that Thread 1 writes to z has no effect on the non-speculative write forwarding explanation, but since that value is now observed by another thread (Thread 2), it can no longer be discarded (i.e. write subsumption cannot be used to exhibit the relaxed behaviour). This shows that write subsumption does not subsume write forwarding from a non-speculative write. For the reverse, S+fen+data-wsi (discussed above) has no instance of a load that succeeds a store to the same location, and therefore it cannot be explained by write forwarding from a non-speculative write. Hence, write forwarding from a non-speculative write does not subsume write subsumption.

Modelling write subsumption in mixed-size settings is complicated as in general it is possible that a write is subsumed by a set of writes, each one overwriting different bytes of the subsumed write.

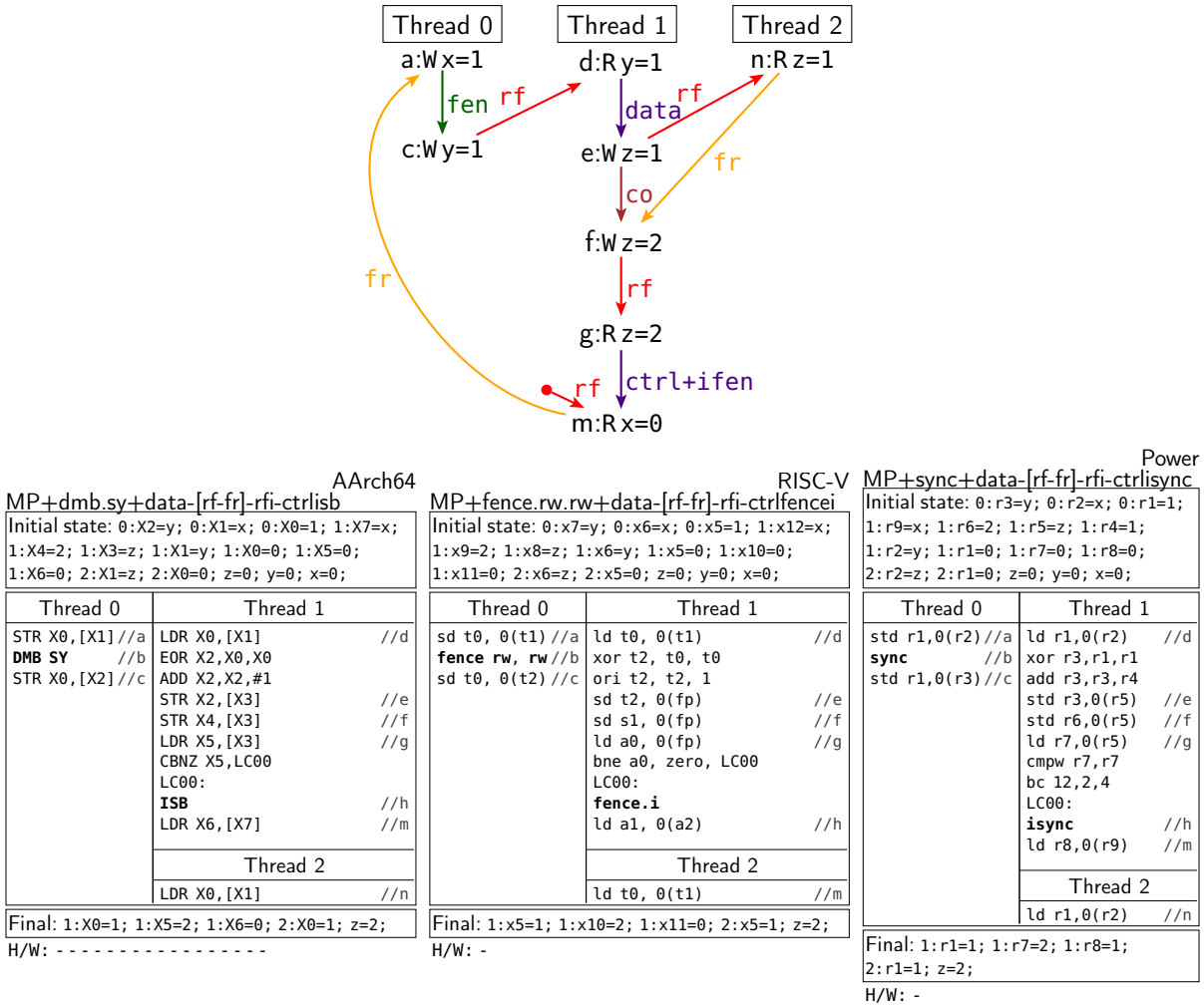


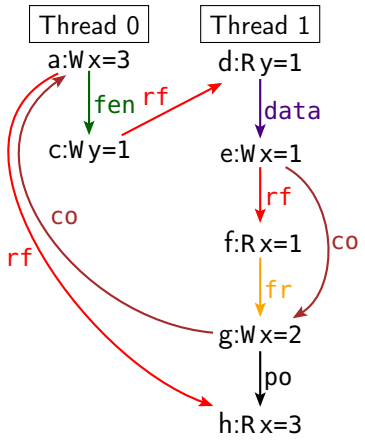
Figure 16.40: Litmus test MP+fen+data-[rf-fr]-rfi-ctrlifen

16.1.13 Symbolic forwarding

This subsection explores the boundaries of what could be architecturally allowed, and it is not clear if any hardware implementation implements such mechanisms.

A hardware implementation may identify a store instruction that is succeeded by a load instruction, where the addresses of both memory accesses are resolved, and to the same location, but the store has a data dependency that is still unresolved. In such a state the hardware implementation can already decide to forward the store to the succeeding load, even though the actual value of the store is not yet known. We call this *symbolic forwarding*.

Consider for example S+fen+data-rfi-fri, which is very similar to S+fen+data-wsi from the discussion about write subsumption, with the addition of a load between the two stores of Thread 1. Recall that such a load would normally prevent the first store from being subsumed by the second store. By deciding that the first store will be forwarded to the intervening load, before the data dependency is resolved, the hardware implementation could subsume the first store by the second store and allow the second store to be propagated to Thread 0. When the data dependency of the first store is resolved, the forwarding is complete, and the first store is never propagated to other threads.



S+dmb.sy+data-rfi-frïAArch64

Initial state: 0:X3=y; 0:X2=1; 0:X1=x; 0:X0=3; 1:X5=2; 1:X3=x; 1:X1=y; 1:X0=0; 1:X4=0; 1:X6=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	EOR X2,X0,X0
STR X2,[X3] //c	ADD X2,X2,#1
	STR X2,[X3] //e
	LDR X4,[X3] //f
	STR X5,[X3] //g
	LDR X6,[X3] //h

Final: 1:X0=1; 1:X4=1; 1:X6=3;

H/W: NNN - - - - NNNNN - NN - N

S+fence.rw.rw+data-rfi-fri

Initial state: 0:x8=y; 0:x7=1; 0:x6=x; 0:x5=3; 1:x10=2; 1:x8=x; 1:x6=y; 1:x5=0; 1:x9=0; 1:x11=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	xor t2, t0, t0
sd t2, 0(fp) //c	ori t2, t2, 1
	sd t2, 0(fp) //e
	ld s1, 0(fp) //f
	sd a0, 0(fp) //g
	ld a1, 0(fp) //h

Final: 1:x5=1; 1:x9=1; 1:x11=3;

H/W: -

RISC-V

S+sync+data-rfi-fri Power

Initial state: 0:r4=y; 0:r3=1; 0:r2=x; 0:r1=3; 1:r7=2; 1:r5=x; 1:r4=1; 1:r2=y; 1:r1=0; 1:r6=0; 1:r8=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	xor r3,r1,r1
std r3,0(r4) //c	add r3,r3,r4
	std r3,0(r5) //e
	ld r6,0(r5) //f
	std r7,0(r5) //g
	ld r8,0(r5) //h

Final: 1:r1=1; 1:r6=1; 1:r8=3;

H/W: -

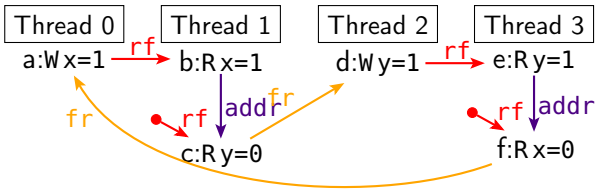
Figure 16.41: Litmus test S+fen+data-rfi-fri

In the non-mixed-size context, symbolic forwarding is only observable in conjunction with write subsumption. The Armv7-A architecture allowed that behaviour, but it is not allowed by the Armv8-A, Power, and RISC-V architectures.

16.1.14 Multi-Copy Atomicity

According to Collier [**?**], who introduced the term *multi-copy atomicity*: "multicopy atomicity requires that all copies of an operand change value at the same instant". Other authors use a more relaxed definition of multi-copy atomicity, as we do here: an architecture is said to be multi-copy atomic if it requires implementations to make each memory write appear as if they become visible to (i.e. can be read from) all hardware threads at the same time, except for the thread from which the write originated from, which might read from the write before the other threads can. The latter definition makes a more useful distinction between practical architectures, as will be explained at the end of this subsection. The Armv8-A Architecture Reference Manual uses Collier's definition of multi-copy atomicity, and calls the more relaxed definition, in which a thread might read from its own writes before other threads can, *other-multi-copy atomicity* [**?**, p. 94].

Although programs do not usually have a direct way of observing when a memory write becomes visible to each thread, a program can, in some cases, observe the relative order in which memory writes become visible. Consider for example IRIW+addrs (Independent Reads of Independent Writes, Fig. 16.42). In this test, Threads 0 and 2 each perform a single write, to x and y respectively. Threads 1 and 3 both read from both of those locations, with address dependencies from the first load to the second load, to keep their execution in-order. Thread 1



IRIW+addrs AArch64

Initial state: 0:X1=x; 0:X0=1;
1:X4=y; 1:X1=x; 1:X0=0; 1:X3=0;
2:X1=y; 2:X0=1; 3:X4=x; 3:X1=y;
3:X0=0; 3:X3=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //b EOR X2,X0,X0 LDR X3,[X4,X2] //c
Thread 2	**Thread 3**
STR X0,[X1] //d	LDR X0,[X1] //e EOR X2,X0,X0 LDR X3,[X4,X2] //f

Final: 1:X0=1; 1:X3=0; 3:X0=1;
3:X3=0;

H/W: NNNNNN-NN---NNN-N

IRIW+addrs RISC-V

Initial state: 0:x6=x; 0:x5=1;
1:x9=y; 1:x6=x; 1:x5=0; 1:x8=0;
2:x6=y; 2:x5=1; 3:x9=x; 3:x6=y;
3:x5=0; 3:x8=0; y=0; x=0;

Thread 0	Thread 1

sd t0, 0(t1) //a	ld t0, 0(t1) //b
	xor t2, t0, t0
	add a0, s1, t2
	ld fp, 0(a0) //c

Thread 2	**Thread 3**

sd t0, 0(t1) //d	ld t0, 0(t1) //e
	xor t2, t0, t0
	add a0, s1, t2
	ld fp, 0(a0) //f

Final: 1:x5=1; 1:x8=0; 3:x5=1;
3:x8=0;

H/W: N

IRIW+addrs Power

Initial state: 0:r2=x; 0:r1=1;
1:r5=y; 1:r2=x; 1:r1=0; 1:r4=0;
2:r2=y; 2:r1=1; 3:r5=x; 3:r2=y;
3:r1=0; 3:r4=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //b xor r3,r1,r1 ldx r4,r3,r5 //c
Thread 2	**Thread 3**
std r1,0(r2) //d	ld r1,0(r2) //e xor r3,r1,r1 ldx r4,r3,r5 //f

Final: 1:r1=1; 1:r4=0; 3:r1=1;
3:r4=0;

H/W: -

Figure 16.42: Litmus test IRIW+addrs

reads x first, and y second, and Thread 3 reads y first, and x second. An execution where Thread 1 reads 1 for x and 0 for y while Thread 3 reads 1 for y and 0 for x, implies that the write of Thread 0 became visible to Thread 1 before the write of Thread 2 did, and the write of Thread 2 became visible to Thread 3 before the write of Thread 0 did. Assume for a contradiction that this execution is multi-copy atomic. By the values that Thread 1 read, the write of Thread 0 became visible (to all threads) before the write of Thread 2 did but, by the values that Thread 3 read, the write of Thread 2 became visible (to all threads) before the write of Thread 0 did. This is of course a contradiction and therefore this execution is not multi-copy atomic.

The Power architecture, which is not multi-copy atomic, allows hardware implementations to make the write of Thread 0 to x visible to Thread 1 before making it visible to Thread 3, and make the write of Thread 2 to y visible to Thread 3 before making it visible to Thread 1. Thread 1 can then read from the write of Thread 0, resolve the address dependency and read the initial value of y, while Thread 3 reads from the write of Thread 2, resolves the address dependency and reads the initial value of x. Here, Thread 1 observes the write of Thread 0 to x before observing the write of Thread 2 to y, while Thread 3 observes the write of Thread 2 to y before observing the write of Thread-0 to x.

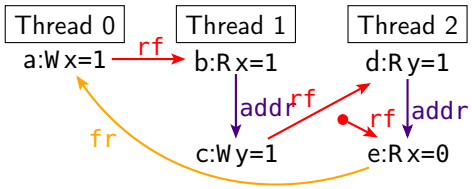
The Armv8-A architecture, as originally specified in the first public Beta release of the Armv8-A reference manual [**?**] from September 2013, was also a non-multi-copy atomic architecture, and as such it allowed IRIW+addrs. This was also the case in the first non-Beta (EAC) release of the reference manual in June 2016 [**?**]. Starting from the version of the reference manual that was released on March 2017 [**?**] the architecture has changed to be multi-copy atomic (other-multi-copy atomic in the manual's terms), and therefore IRIW+addrs is no longer allowed by Armv8-A. As all public implementations of the Armv8-A architecture, up to that time, were multi-copy atomic (i.e., they were all stronger than the architecture required), this change to the architecture did not render those implementations unsound.

This change of the Armv8-A architecture, from non-multi-copy atomic to multi-copy atomic, was informed in part by observations from the above-cited work. Allowing non-multi-copy

atomic behaviour added substantial complexity to the memory model, especially 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. In the Arm context, the potential performance benefits were not thought to justify the complexity of implementation, validation, and reasoning.

The Armv7-A architecture is non-multi-copy atomic and allows IRIW+addrs. The RISC-V architecture is multi-copy atomic and so it forbids IRIW+addrs.

For a long time it was thought that the IRIW shape does not come up in practice, until a code from the Java virtual machine [**?**] was discovered to exhibit it. The WRC (Write-to-Read Causality) [**?**] shape, which is also affected by multi-copy atomicity, might be more common than IRIW. Consider WRC+addrs, this test is very similar to MP, except that in WRC+addrs the



WRC+addrs AArch64

Initial state: 0:X1=x; 0:X0=1; 1:X4=y; 1:X3=1; 1:X1=x; 1:X0=0; 2:X4=x; 2:X1=y; 2:X0=0; 2:X3=0; y=0; x=0;

Thread 0	Thread 1	Thread 2
STR X0,[X1] //a	LDR X0,[X1] //b EOR X2,X0,X0 STR X3,[X4,X2] //c	LDR X0,[X1] //d EOR X2,X0,X0 LDR X3,[X4,X2] //e

Final: 1:X0=1; 2:X0=1; 2:X3=0;

H/W: NNNNNN-NNN-NNNN-N

WRC+addrs RISC-V

Initial state: 0:x6=x; 0:x5=1; 1:x9=y; 1:x8=1; 1:x6=x; 1:x5=0; 2:x9=x; 2:x6=y; 2:x5=0; 2:x8=0; y=0; x=0;

Thread 0	Thread 1	Thread 2

sd t0, 0(t1) //a	ld t0, 0(t1) //b
	xor t2, t0, t0
	add a0, s1, t2
	sd fp, 0(a0) //c
		ld t0, 0(t1) //d
		xor t2, t0, t0
		add a0, s1, t2
		ld fp, 0(a0) //e

Final: 1:x5=1; 2:x5=1; 2:x8=0;

H/W: N

WRC+addrs Power

Initial state: 0:r2=x; 0:r1=1; 1:r5=y; 1:r4=1; 1:r2=x; 1:r1=0; 2:r5=x; 2:r2=y; 2:r1=0; 2:r4=0; y=0; x=0;

Thread 0	Thread 1	Thread 2
std r1,0(r2) //a	ld r1,0(r2) //b xor r3,r1,r1 stdx r4,r3,r5 //c	ld r1,0(r2) //d xor r3,r1,r1 ldx r4,r3,r5 //e

Final: 1:r1=1; 2:r1=1; 2:r4=0;

H/W: Y

Figure 16.43: Litmus test WRC+addrs

first thread is split to two threads, the first one doing the first store, to x, and the second one reading from x, and then doing the store to y, with an address dependency from the read of x, to keep those two accesses locally in order. The third thread then (similarly to the second thread of MP) reads from y and x, with an address dependency between the reads to keep them locally in order. Despite the local order of the accesses in each thread, in a non-multi-copy architecture the read of x in the third thread can return 0 while the read of y in the same thread returns 1. In such an execution, the third thread observes the write to x (by Thread 0) after the write to y (by Thread 1), hence the write to y must become visible to all threads (except Thread 1) before the write to x. This contradicts the address dependency in the second thread, that requires Thread 1 to observe the write to x before performing the write to y, which means the write to y can become visible only after the write to x has already become visible.

WRC+addrs is allowed by the Power and Armv7-A architectures, and forbidden by the Armv8-A and RISC-V architectures.

For non-multi-copy atomic architectures, non-multi-copy atomic behaviour can be exhibited when some threads share caches or store-buffers that are not shared with other threads. For example, a multi-processor machine with 4 cores might have two separate L2 caches, the first one shared by the first two cores and the second one shared by the two other cores. In such configuration, some cache protocols will allow the first core to read from L2 values that were

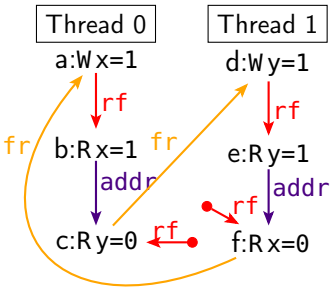
written by the second core, while the other cores can still read older values from their L2 cache. When similar cache configurations are used in multi-copy atomic implementations, the cache protocol has to perform additional synchronisation to guarantee that when a value is written to one of the L2 caches, old values in the other L2 caches are no longer accessible.

A similar example is simultaneous multithreading (SMT), where a single core executes multiple hardware threads at the same time. If the core has an L1 cache that is not shared with any other core, the hardware threads running on a single core share that core's L1 cache, and so those hardware threads can potentially observe the writes of each other while other threads can still observe older values. SMT in a multi-copy atomic implementation can, for example, tag cache lines to prevent sharing of cache lines between hardware threads.

Non-multi-copy atomicity can also be exhibited on system with symmetric caches, where a shared cache is always shared by all cores.

Terminology

Returning to the terminology around multicopy atomicity, note that SB+rfi-addrs is forbidden by



SB+rfi-addrs AArch64

Initial state: 0:X5=y; 0:X1=x; 0:X0=1; 0:X2=0; 0:X4=0; 1:X5=x; 1:X1=y; 1:X0=1; 1:X2=0; 1:X4=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	STR X0,[X1] //d
LDR X2,[X1] //b	LDR X2,[X1] //e
EOR X3,X2,X2	EOR X3,X2,X2
LDR X4,[X5,X3] //c	LDR X4,[X5,X3] //f

Final: 0:X2=1; 0:X4=0; 1:X2=1; 1:X4=0;

H/W: - - - - - - - - - - - - - - - - -

SB+rfi-addrs RISC-V

Initial state: 0:x10=y; 0:x6=x; 0:x5=1; 0:x7=0; 0:x9=0; 1:x10=x; 1:x6=y; 1:x5=1; 1:x7=0; 1:x9=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	sd t0, 0(t1) //d
ld t2, 0(t1) //b	ld t2, 0(t1) //e
xor fp, t2, t2	xor fp, t2, t2
add a1, a0, fp	add a1, a0, fp
ld s1, 0(a1) //c	ld s1, 0(a1) //f

Final: 0:x7=1; 0:x9=0; 1:x7=1; 1:x9=0;

H/W: N

SB+rfi-addrs Power

Initial state: 0:r6=y; 0:r2=x; 0:r1=1; 0:r3=0; 0:r5=0; 1:r6=x; 1:r2=y; 1:r1=1; 1:r3=0; 1:r5=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2) //a	std r1,0(r2) //d
ld r3,0(r2) //b	ld r3,0(r2) //e
xor r4,r3,r3	xor r4,r3,r3
ldx r5,r4,r6 //c	ldx r5,r4,r6 //f

Final: 0:r3=1; 0:r5=0; 1:r3=1; 1:r5=0;

H/W: Y

Figure 16.44: Litmus test SB+rfi-addrs

Collier's definition of multi-copy atomicity, and is allowed by the more relaxed definition. This test is similar to IRIW+addrs, only that Thread 0 and Thread 1 are merged to a single thread, and so are Thread 2 and Thread 3.

Armv8-A, RISC-V, and x86 (TSO) all allow SB+rfi-addrs, and forbid the IRIW+addrs, and therefore are non-multi-copy atomic by Collier's definition, but are multi-copy atomic by the more relaxed definition (other-multi-copy atomic in Armv8-A terms). The Power architecture allows IRIW+addrs (and SB+rfi-addrs), and therefore Power is non-multi-copy atomic by both definitions. The following table indicates which architectures (and SC) are MCA by the different

definitions.

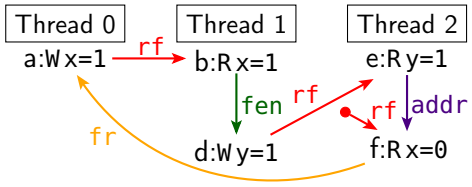
	Collier MCA / Arm OMCA	our MCA
SC	yes	yes
x86 (TSO)	no	yes
Armv8-A, RISC-V	no	yes
Armv7-A	no	no
Power	no	no

If an architecture intends to allow implementations to use store-buffers effectively, which is in fact one of the oldest and most widely used hardware optimisations, and is usually the intention of any non-SC architecture, this architecture can not be multi-copy atomic by Collier's definition, but can still be multi-copy atomic by the more relaxed definition (as Armv8-A, RISC-V and x86 are). Hence, Collier's definition partitions the architectures in a non-useful way, which is why the more relaxed definition is preferred by other authors and here.

Cumulativity of barriers

The presence of an appropriate barrier between two memory writes of a thread (such as in MP+fen+po for example), forces the memory writes to become visible to all threads in-order. In non-multi-copy atomic architectures the barrier can also have an ordering effect on writes from other threads. A memory barrier is said to be A-cumulative if the barrier forces any write from other threads that was observed (i.e. read from, or by A-cumulativity of another barrier) by the thread that performs the barrier, before the barrier was performed, to become visible to all other threads before any write that succeeds the barrier in program-order.

In WRC+fen+addr, for example, Thread 1 observes the write of Thread 0 to x, by reading



WRC+dmb.sy+addr AArch64

Initial state: 0:X1=x; 0:X0=1; 1:X3=y; 1:X2=1; 1:X1=x; 1:X0=0; 2:X4=x; 2:X1=y; 2:X0=0; 2:X3=0; y=0; x=0;

Thread 0	Thread 1	Thread 2
STR X0,[X1] //a	LDR X0,[X1] //b **DMB SY** //c STR X2,[X3] //d	LDR X0,[X1] //e EOR X2,X0,X0 LDR X3,[X4,X2] //f

Final: 1:X0=1; 2:X0=1; 2:X3=0;

H/W: NNNNNN - NNN - NNNN - N

WRC+fence.rw.rw+addr RISC-V

Initial state: 0:x6=x; 0:x5=1; 1:x8=y; 1:x7=1; 1:x6=x; 1:x5=0; 2:x9=x; 2:x6=y; 2:x5=0; 2:x8=0; y=0; x=0;

Thread 0	Thread 1	Thread 2

sd t0, 0(t1) //a	ld t0, 0(t1) //b
	fence rw, rw //c
	sd t2, 0(fp) //d
		ld t0, 0(t1) //e
		xor t2, t0, t0
		add a0, s1, t2
		ld fp, 0(a0) //f

Final: 1:x5=1; 2:x5=1; 2:x8=0;

H/W: N

WRC+sync+addr Power

Initial state: 0:r2=x; 0:r1=1; 1:r4=y; 1:r3=1; 1:r2=x; 1:r1=0; 2:r5=x; 2:r2=y; 2:r1=0; 2:r4=0; y=0; x=0;

Thread 0	Thread 1	Thread 2

std r1,0(r2) //a	ld r1,0(r2) //b
	sync //c
	std r3,0(r4) //d
		ld r1,0(r2) //e
		xor r3,r1,r1
		ldx r4,r3,r5 //f

Final: 1:r1=1; 2:r1=1; 2:r4=0;

H/W: -

Figure 16.45: Litmus test WRC+fen+addr

from it, before it performs the memory barrier, and therefore (if the barrier is A-cumulative) when Thread 2 reads from y, the write of Thread 0 to x must already be visible to Thread 2, and therefore the following read from x cannot return 0.

As the Power sync barrier is A-cumulative, the Power architecture forbids the WRC+fen+addr behaviour. The Armv8-A and RISC-V architectures also forbid this behaviour, but as those are

multi-copy atomic architectures, the local ordering of the barrier is enough to forbid this behaviour for those architectures, in the same way that the address dependency in WRC+addrs did.

A memory barrier is said to be B-cumulative if the barrier forces any write that precedes the barrier in program-order to become visible to all threads before any other write that is observed by some other thread after a write that succeeds the barrier in program-order.

To see the effects of B-cumulativity, consider the ISA2+fen+addr+addr litmus test. This test

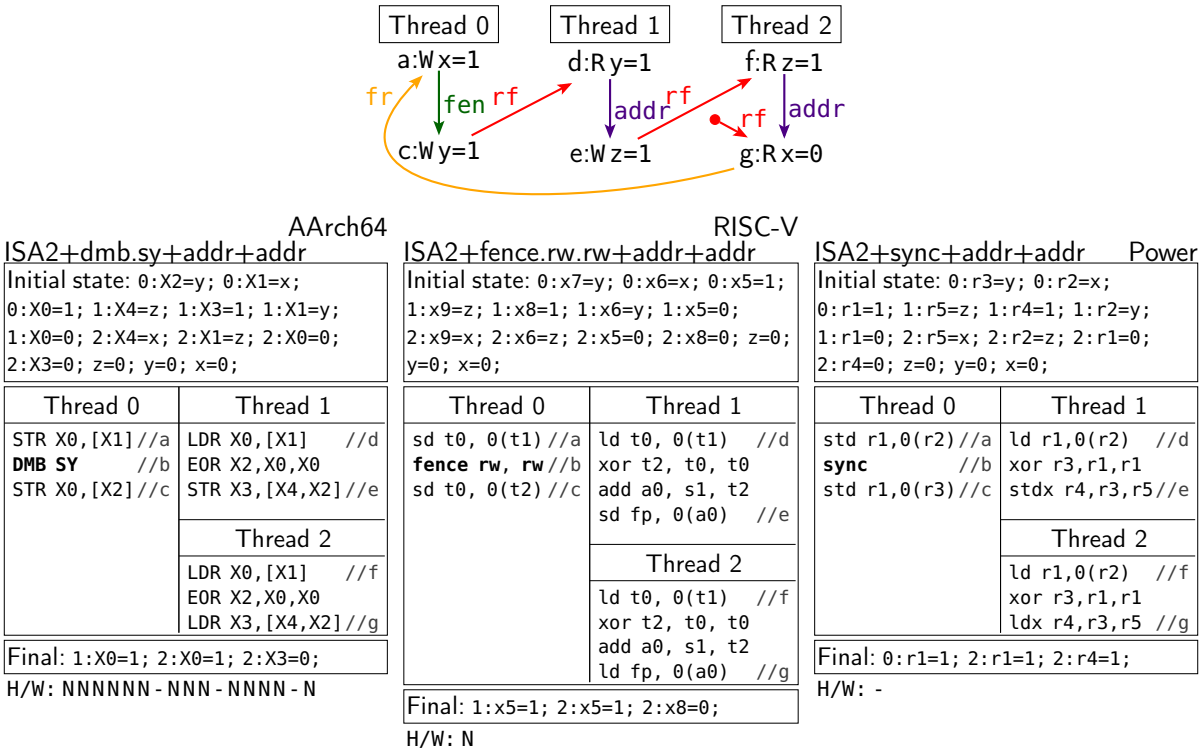


Figure 16.46: Litmus test ISA2+fen+addr+addr

is somewhat similar to MP, with the second thread of MP split over two threads, one that reads from y and then writes to a new location z, and a second thread that reads from z and then reads from x, with address dependencies in those two threads to keep the memory accesses locally in-order, and a barrier in Thread 0. The write to z of Thread 1 can only be performed after the read from y by the same thread, because of the address dependency between the two memory accesses. Hence, the write to z can be performed only after the write of Thread 0 to x became visible to Thread 1. Therefore, if the barrier in Thread 0 is B-cumulative, the write of Thread 0 to x must become visible before the write to z for Thread 2, and therefore if the read of z by Thread 2 returns 1, the following read of x cannot return 0.

As the Power sync barrier is B-cumulative, the Power architecture forbids this behaviour. The Armv8-A and the RISC-V architectures also forbid this behaviour, but again, as those architectures are multi-copy atomic, the local ordering of the barrier is enough to explain that.

16.1.15 Atomic Memory Modification

The Armv8-A, RISC-V, and Power architectures were all originally conceived as RISC architectures. Principally, instructions that perform multiple memory accesses are counter to the RISC principles, and even more so, instructions that do multiple memory accesses atomically (i.e., no other memory access can be observed to be performed between the memory accesses of the atomic instruction). Instead, RISC architectures such as Arm, Power, and RISC-V, used to only offer a pair of special load and store instructions that when performed successfully guarantee

some atomicity between the accesses of the load and store: load-exclusive (`LDXR`) and store-exclusive (`STXR`) in Arm; load-and-reserve (`lwarx`) and store-conditional (`stwcx`) in Power; and load-reserved (`LR`) and store-conditional (`SC`) in RISC-V. Those architectures also provide variants of those instructions, with different access sizes and other attributes. To the best of the author's knowledge, this mechanism was first designed for the S-1 AAP multiprocessor; there the special instructions are called sync-load and sync-store [**?**]. Note that in some older architectures, the similarly named conditional-store is a read-modify-write instruction, different from the Power and RISC-V store-conditional, though it is used to achieve similar synchronisation goals, in different ways. The special load instruction is also sometimes called load-lock, or load-link. In this text, the special load and store instructions are uniformly referred to as *load-exclusive* and *store-exclusive*, respectively, regardless of the architecture.

The AArch64 assembly code in Fig. 16.47 uses load-exclusive and store-exclusive instructions to atomically increment the value in memory pointed by `X1`, by the value held in `X2`. The code starts by reading the value from memory into register `X0`, using a load-exclusive instruction (`LDXR`). The code then adds the value of `X2` to `X0`, and attempts to store the result back to memory, using a store-exclusive instruction (`STXR`). This can either succeed or fail. If it succeeds, the store-exclusive writes 0 to register `X3`, and guarantees that the memory write of the store-exclusive is the immediate successor of the write from which the load-exclusive read from, in the coherence-order, and that no other write will ever be placed between them in the future. If the store-exclusive fails, the store-exclusive writes 1 to register `X3`. The conditional branch that follows the store-exclusive causes this sequence of instructions to repeat until the store-exclusive succeeds.

```
loop:
LDXR X0,[X1]
ADD X0,X0,X2
STXR X3,X0,[X1]
CBNZ X3,loop
```

Figure 16.47:

Note that in general, in some architecture, the success of a store-exclusive does not prohibit writes from the same thread to intervene in the coherence order between the write the load-exclusive read from and the store-exclusive write.

A store-exclusive is paired with the most recent program-order preceding load-exclusive (to the same location or not), if such a load exists, and there are no other store-conditional (successful, unsuccessful, or undetermined, to any location) between them. An unpaired atomic store will always fail in Arm, Power, and RISC-V.

A simple implementation of load-exclusive/store-exclusive can leverage the cache protocol. When the load-exclusive is performed, the cache line holding the relevant memory location is marked as owned by the thread performing the load-exclusive and an additional flag indicates that this line is also reserved. Any operation that normally changes the ownership (e.g. a write by a different thread) also clears the reservation flag. When the store-exclusive is executed, if the reservation flag is set, the store succeeds, and the reservation flag is cleared. In any other case the store-exclusive fails.

Note that this implementation, without additional measures, could easily lead to a livelock. For example, if two threads are executing code similar to the one in Fig. 16.47, the first thread could perform the load-exclusive first, which would take ownership of the cache line, then, before the first thread gets to perform its store-exclusive, the second thread could execute its load-exclusive which would clear the ownership of the first thread (and the reserved flag). This would cause the store-exclusive of the first thread to fail, and then, before the second thread gets

to perform its store-exclusive, the first thread could execute its load-exclusive (for the second time) which would clear the ownership of the second thread (and the reserved flag). This would cause the store-exclusive of the second thread to fail, and so on.

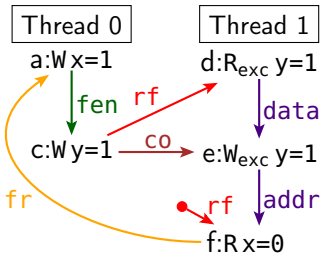
To allow programmers to avoid livelocks, the Arm and RISC-V architectures specify conditions for eventually making forward progress (which hardware implementations must respect): ARM Architecture Reference Manual, B2.9.5 Load-Exclusive and Store-Exclusive instruction usage restrictions [**?**, p. 142]; and, RISC-V Instruction Set Manual, 8.2 Load-Reserved/Store-Conditional Instructions [**?**, p. 49]. The Power architecture does not make such guarantees as an architecture, but implementations make their best effort to eventually make forward progress, and individual implementations can specify their own conditions for forward progress (Power ISA: 1.7.4.2 Forward Progress [**?**, p. 820]).

For implementation simplicity, after a load-exclusive is performed, implementations are not required to track accesses to the exact memory location and size of the load-exclusive, in order to fail the paired store-exclusive if an intervening access is detected. Instead, an implementation may track accesses to a larger block of memory, a *reservation granule* that contains the load-exclusive access. One side effect of this imprecision is that a store-exclusive can fail because of an intervening write that is to a different location from the load/store-exclusive accesses. Semantically this is subsumed by the fact that store-exclusives are allowed to fail arbitrarily. Another side effect of this, is that the architectures do not require a store-exclusive to fail if it is paired with a load-exclusive that access a non-overlapping memory location (as that would require a precise tracking of the load-exclusive access). In the case where the paired load-exclusive and store-exclusive are to a non-overlapping memory locations, it is not clear what kind of atomicity is guaranteed, as the write the load-exclusive reads from might not be ordered with the write of the store-exclusive in the coherence-order. One will normally consider this a programming error.

In §16.1.3 dependencies were defined in terms of data-flow through registers. As such, a dependency stems from a memory load instruction, where the value the load instruction writes to the output register comes from memory, and not from some computation that only depends on registers. The value that is written to the status register of the store-exclusive instruction is similarly not the result of a computation that only depends on the input registers, and as such this register write can be the source of dependencies. In the Power and RISC-V architectures, data-flow through registers that stems from the status register of a store-exclusive preserves observable order, and therefore it induces dependencies. In the Arm architecture this data-flow is not guaranteed to preserve observable order. For example, consider MP+fen+frixx-addr, where the address dependency in Thread 1 stems from the status register of the store-exclusive. As the Power and RISC-V architectures require the address dependency to be respected, the second load of Thread 1 cannot be performed before the store-exclusive, which in turn cannot be performed (successfully) before the load-exclusive. Together with the barrier in Thread 0 this makes the MP+fen+frixx-addr behaviour, on Power and RISC-V, forbidden. On Arm, on the other hand, the address dependency in Thread 1 is not respected as such, and therefore it is valid to speculate the result of the store-exclusive before it is performed, which allows the second load of Thread 1 to be satisfied before any of the stores in Thread 0 have been performed, which makes the behaviour of MP+fen+frixx-addr allowed on Arm.

The reason for the decision to not respect store-exclusive dependencies in Arm is to allow a microarchitectural optimisation in which a pair of load-exclusive and store-exclusive is replaced by an atomic read-modify-write operation in memory. When a pair of exclusives is detected and replaced by a read-modify-write operation, the store-exclusive is guaranteed to succeed before the read-modify-write access is performed, and therefore 0 can be written to the status register of the store-exclusive very early, and succeeding instructions in program-order can read it, effectively breaking any dependency from the status register.

The Power architecture has some additional restrictions on the execution of exclusives that



RISC-V

MP+dmb.sy+frixx-addr AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1; 1:X5=x; 1:X0=y; 1:X1=0; 1:X6=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDXR X1,[X0] //d
DMB SY //b	STXR W2,X1,[X0] //e
STR X0,[X2] //c	EOR X3,X2,X2
	LDR X6,[X5,X3] //f

Final: 1:X1=1; 1:X2=0; 1:X6=0; y=1; x=1;

H/W: - - - - - - - - - - - - - - - - -

MP+fence.rw.rw+frixx-addr

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x9=x; 1:x5=y; 1:x10=0; 1:x6=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	lr.d t1, t0 //d
fence rw, rw //b	sc.d t2, t0, t1 //e
sd t0, 0(t2) //c	xor fp, t2, t2
	add s1, s1, fp
	ld a0, 0(s1) //f

Final: 1:x6=1; 1:x7=0; 1:x10=0; y=1; x=1;

H/W: -

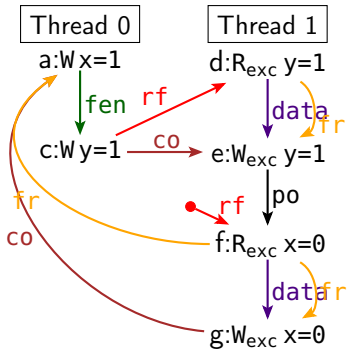
Figure 16.48: Litmus test MP+fen+frixx-addr

are due to the fact that a hardware thread can only track a single load-exclusive at a time. As a result, MP+fen+poxx, where the loads of Thread 1 are load-exclusives, is forbidden on Power. Normally, to exhibit an MP litmus test with a fence between the stores of Thread 0, the second load of Thread 1 has to be speculated before the first load of Thread 1 is satisfied. In the case of MP+fen+poxx on Power, the second load cannot be speculated before the first load is satisfied as both of those loads are load-exclusive and both need to use the reservation facility, of which only one exists. Hence, only after the first load has finished its use of the reservation facility (and therefore has already been satisfied), the second load can be satisfied and make use of the now free reservation facility. Note that this also restricts the execution of store-exclusives: those must execute in program-order with respect to other store-exclusives, but the propagation of the writes from those store-exclusives can still make them appear to be executed out-of-order. This is demonstrated by MP+poxx+addr which is allowed by the Power architecture.

Finally, the Arm architecture forbids write-forwarding from a store-exclusive to a load-acquire, and the Power and RISC-V architectures forbid write-forwarding from a store-exclusive altogether. The reason for this restriction is to enable efficient compilation schemes for C atomics. In particular, this is required to preserve the order induced by C release sequences where the sequence ends with a load-acquire that reads from a read-modify-write (compiled to load-exclusive/store-exclusive loop), and both operations are from the same thread.

16.1.16 Release/Acquire Memory Accesses

Using barriers to enforce order can be expensive, and using dependencies, in places where dependencies do not naturally occur, can make the code unnecessarily complicated. Some architectures, such as Arm and RISC-V (but not Power), provide special load and store instructions with release/acquire semantics that can be used to enforce order instead of barriers and depen-



MP+dmb.sy+poxx AArch64

Initial state: 0:X2=y; 0:X1=x; 0:X0=1;
1:X3=x; 1:X0=y; 1:X1=0; 1:X2=0;
1:X4=0; 1:X5=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDXR X1,[X0] //d
DMB SY //b	STXR W2,X1,[X0] //e
STR X0,[X2] //c	LDXR X4,[X3] //f
	STXR W5,X4,[X3] //g

Final: 1:X1=1; 1:X2=0; 1:X4=0;
1:X5=0; y=1; x=1;

H/W: - - - - - - - - - - - - - - - - -

MP+fence.rw.rw+poxx RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1;
1:x8=x; 1:x5=y; 1:x6=0; 1:x7=0;
1:x9=0; 1:x10=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	lr.d t1, t0 //d
fence rw, rw //b	sc.d t2, t0, t1 //e
sd t0, 0(t2) //c	lr.d s1, fp //f
	sc.d a0, fp, s1 //g

Final: 1:x6=1; 1:x7=0; 1:x9=0;
1:x10=0; y=1; x=1;

H/W: -

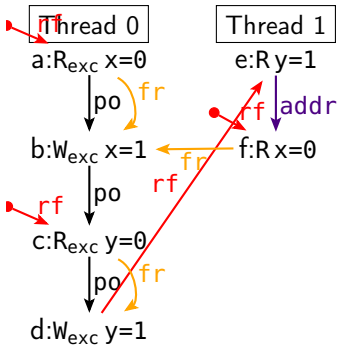
Figure 16.49: Litmus test MP+fen+poxx

dencies. In general, a store-release instruction will not execute observably out-of-order with any program-order preceding memory access instruction, and a load-acquire will not execute observably out-of-order with any program-order succeeding memory access instruction. For example, the litmus test MP+poprl+poaqp that has a store-release for the second store of Thread 0, and a load-acquire for the first load of Thread 1, is forbidden by Arm and RISC-V. The store-release in Thread 0 must execute in-order with the store that precedes it, and the load-acquire in Thread 1 must execute in-order with the load that succeeds it. Hence, all the memory accesses in this litmus test must execute in-order, and therefore, as there are only two threads, a non-SC behaviour is not possible.

Note that because both Armv8-A and RISC-V are multi-copy atomic, using release/acquire to enforce thread-local in-order execution is enough to eliminate non-SC behaviour. As such, WRC+poprl+poaqp, which uses a store-release in Thread 1, and a load-acquire in Thread 2, to enforce thread-local order in those threads, but uses a regular store in Thread 0, is forbidden by the Armv8-A and RISC-V architectures. In those architectures, even though the store in Thread 0 is a regular store, the store-release in Thread 1 can only be performed after the load in the same thread is satisfied, which guarantees that the regular store of Thread 0 has already propagated to all threads, including Thread 2.

In non-multi-copy atomic architectures, the same WRC+poprl+poaqp might be allowed. The author is not aware of any such architecture, but the analogous C litmus test in Fig. 16.53 is an allowed C behaviour, as the C semantics does not assume multi-copy atomicity. This behaviour can be eliminated by changing the store of Thread 0 to store-release, and changing the load of Thread 1 to load-acquire.

In fact, the WRC+poprl+poaqp was also forbidden in the Armv8-A architecture when it was non-MCA, before it transformed to MCA. The reason for this is that the intended semantics of



MP+poxx+addr AArch64

Initial state: 0:X4=y; 0:X1=1; 0:X0=x; 0:X2=0; 0:X3=0; 0:X5=0; 0:X6=0; 1:X4=x; 1:X1=y; 1:X0=0; 1:X3=0; y=0; x=0;

Thread 0		Thread 1	
LDXR X2,[X0]	//a	LDR X0,[X1]	//e
STXR W3,X1,[X0]	//b	EOR X2,X0,X0	
LDXR X5,[X4]	//c	LDR X3,[X4,X2]	//f
STXR W6,X1,[X4]	//d		

Final: 0:X2=0; 0:X3=0; 0:X5=0; 0:X6=0; 1:X0=1; 1:X3=0; y=1; x=1;

H/W: - - - - - - - - - - - - - - - - -

MP+poxx+addr RISC-V

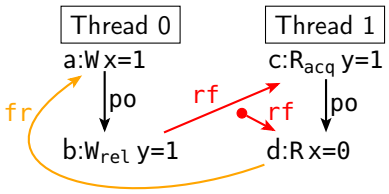
Initial state: 0:x9=y; 0:x6=1; 0:x5=x; 0:x7=0; 0:x8=0; 0:x10=0; 0:x11=0; 1:x9=x; 1:x6=y; 1:x5=0; 1:x8=0; y=0; x=0;

Thread 0		Thread 1	
lr.d t2, t0	//a	ld t0, 0(t1)	//e
sc.d fp, t0, t1	//b	xor t2, t0, t0	
lr.d a0, s1	//c	add a0, s1, t2	
sc.d a1, s1, t1	//d	ld fp, 0(a0)	//f

Final: 0:x7=0; 0:x8=0; 0:x10=0; 0:x11=0; 1:x5=1; 1:x8=0; y=1; x=1;

H/W: -

Figure 16.50: Litmus test MP+poxx+addr



MP+popl+poap AArch64

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

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

Final: 1:X0=1; 1:X2=0;

H/W: NNNNNN-NNNNNNNNNN

MP+poprl+poaqp RISC-V

Initial state: 0:x7=y; 0:x6=x; 0:x5=1; 1:x8=x; 1:x6=y; 1:x5=0; 1:x7=0; y=0; x=0;

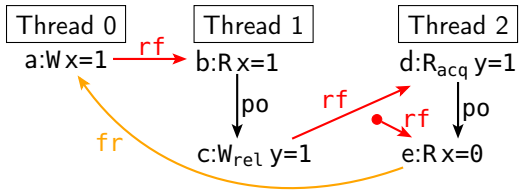
Thread 0		Thread 1	
sd t0, 0(t1)	//a	ld.aq t0, 0(t1)	//c
sd.rl t0, 0(t2)	//b	ld t2, 0(fp)	//d

Final: 1:x5=1; 1:x7=0;

H/W: -

Figure 16.51: Litmus test MP+poprl+poaqp

store-release and load-acquire in Armv8-A is *Release Consistency with special access sequentially consistent* (RCsc) [**?**]. In the RCsc semantics, memory accesses to shared locations are partitioned to competing accesses which are intended to force-order, such as the writing and reading of the flag in message-passing, and non-competing accesses that are protected by the former



WRC+popl+poap AArch64

Initial state: 0:X1=x; 0:X0=1; 1:X3=y; 1:X2=1; 1:X1=x; 1:X0=0; 2:X3=x; 2:X1=y; 2:X0=0; 2:X2=0; y=0; x=0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //b STLR X2,[X3] //c
	Thread 2
	LDAR X0,[X1] //d LDR X2,[X3] //e

Final: 1:X0=1; 2:X0=1; 2:X2=0;

H/W: - - - - - - - - - - - - - - - - -

WRC+poprl+poaqp RISC-V

Initial state: 0:x6=x; 0:x5=1; 1:x8=y; 1:x7=1; 1:x6=x; 1:x5=0; 2:x8=x; 2:x6=y; 2:x5=0; 2:x7=0; y=0; x=0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //b sd.rl t2, 0(fp) //c
	Thread 2
	ld.aq t0, 0(t1) //d ld t2, 0(fp) //e

Final: 1:x5=1; 2:x5=1; 2:x7=0;

H/W: -

Figure 16.52: Litmus test WRC+poprl+poaqp

Thread 0

```
atomic_store_explicit(x, 1,
  memory_order_relaxed);
```

Thread 1

```
int r0 = atomic_load_explicit(x, // Read 1
           memory_order_relaxed);
atomic_store_explicit(y, 1,
  memory_order_release);
```

Thread 2

```
int r1 = atomic_load_explicit(y, // Read 1
           memory_order_acquire);
int r2 = atomic_load_explicit(x, // Read 0
           memory_order_relaxed);
```

Figure 16.53: The WRC litmus test in C

accesses, such as the writing and reading of the data in message-passing. With RCsc semantics, a properly-labelled program where competing store accesses are store-release, and competing load accesses are load-acquire, can only exhibit SC behaviour. This makes the Armv8-A (confusingly named) load-acquire and store-release a suitable mapping from the SC-atomic load and store of C. Similarly, the RISC-V load-acquire-RCsc and store-release-RCsc implement the RCsc semantics. The RISC-V architecture has an additional pair of load and store instructions, load-acquire-RCpc and store-release-RCpc, which implement the slightly weaker *Release Consistency with special access processor consistent* (RCpc) semantics [**?**]. The Armv8-A architecture provides a load-acquirePC, which can be paired with store-release to achieve RCpc semantics.

16.2 Mixed-Size Memory Accesses

Previous sections considered only memory accesses that were of the same size and suitably aligned. In such settings a load instruction always reads a value that was written by a single store instruction (or from the initial state). In practice, however, load and store instructions can

access different numbers of bytes. For example, the AArch64 instruction set of Armv8-A includes the instructions `LDRB`, for loading a single byte from memory, `LDRH`, for loading a half-word (two bytes) from memory, and `LDR` for loading either a word (four bytes) or a double word (eight bytes) from memory, together with `STRB`, `STRH` and `STR` for storing the corresponding sized values to memory. The location accessed by those instructions is not required to be aligned, though misaligned accesses are not guaranteed to be (single-copy) atomic (see §16.2.3). AArch64, along with the other architectures that appear in this thesis and most other modern architectures, does not support sub-byte accesses.

16.2.1 Reading from Multiple Writes

The most fundamental phenomenon of the mixed-size setting is that writes and reads may be of different sizes, with reads potentially reading from fragments of writes and from multiple writes. For example, in the sequential case, we might have a sequence of two overlapping writes followed by a read that reads from both of them, as in the MIX1 litmus test.

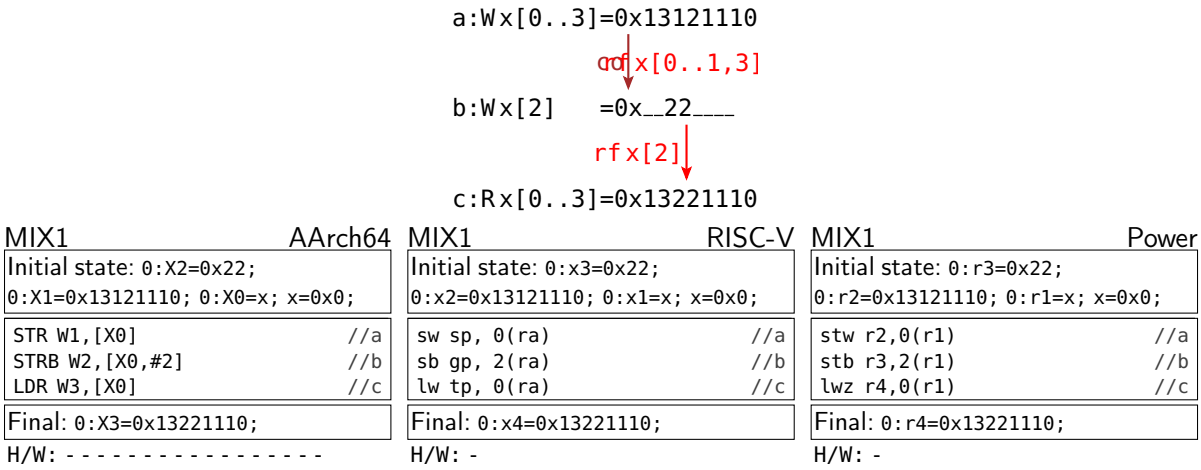


Figure 16.54: Litmus test MIX1

Here, `l[i..j]=n` indicates that the value `n` is being written to or read from the sequence of bytes at memory locations `l+i`, `l+i+1`, ..., `l+j`, where `l` is a shared memory location symbol (e.g. `x`, `y`, `z`), `i` and `j` are integers, and `n` is a memory value comprising of `j+1-i` bytes. The set of byte locations that an instruction (or memory event) accesses are called the memory footprint of the instruction (or memory event). When talking about an instruction (or memory event) that access multiple bytes, `l[i..j]`, it is said that the instruction (or memory event) access `j+1-i` bytes at `l+i`. The shorthand `l[i]=n` stands for `l[i..i]=n`, i.e., the value `n` is being written to or read from the one byte at `l+i`. Recall that in this thesis, memory accesses in non-mixed-size litmus tests are all 8 bytes, hence a memory access `l=n` in a non-mixed-size litmus test is a shorthand for `l[0..7]=n`.

As a read event in the mixed-size setting might read from multiple writes, a read event in the litmus diagrams can have multiple read-from edges, one from each write it reads from. In addition, the read-from edges are labelled with the subset of bytes the read event reads from, as a read event does not necessarily read all the bytes of the write it reads from. This can be seen in the diagram of MIX1 where the rf edge from a to c is labelled with $x[0..1, 3]$ to indicate that in this execution c reads only from the bytes $x + 0$, $x + 1$, and $x + 3$, out of the four bytes that a writes. For the remaining byte, $x + 2$, c reads from the value that b writes, as indicated by the label x[2] of the rf edge from b to c.

16.2.2 Mixed-size Coherence

Lifting the definition of coherence to mixed-size memory accesses requires some care. In the non-mixed-size setting we defined coherence over memory accesses (**??**). That definition of coherence partitions the memory accesses in an execution by their location, and induces a total order over the memory accesses in each partition. In a non-mixed-size execution, all the memory accesses in a single partition have exactly the same footprint.

In the mixed-size context there are two plausible coherence notions: per-byte and per-access. The two notions are defined as follows.

Those two definitions of coherence are not equivalent. To see the difference between them, consider MP+sis. This test can be viewed as a version of MP where Thread 0 writes some data

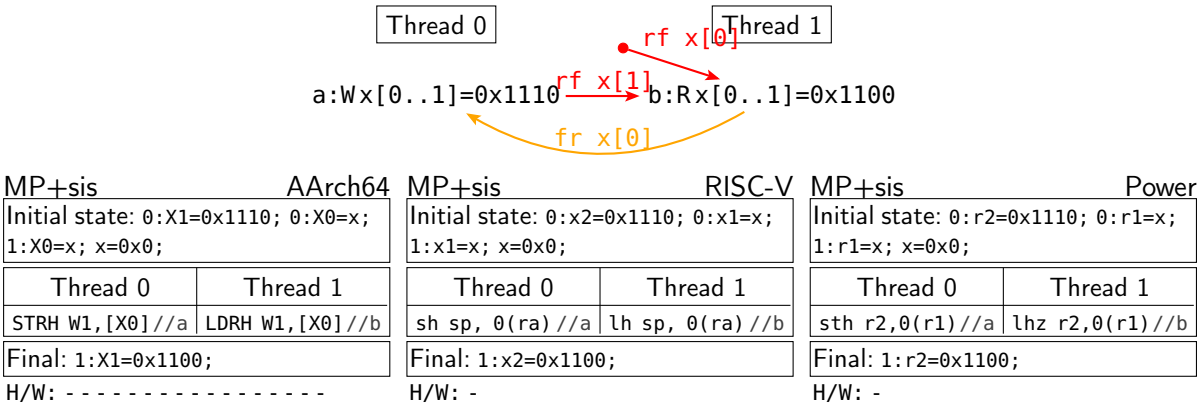


Figure 16.55: Litmus test MP+sis

to $x[0]$ and sets the flag at $x[1]$ in one big store to $x[0..1]$, and Thread 1 reads the flag and the data in one big load from $x[0..1]$. The si part in the litmus test name stands for *same-instruction*, indicating that the two memory accesses (two one-byte accesses) it relates are from the same instruction (the suffix s, as usual, indicates that the si applies to both threads).

As a and b have overlapping footprints, per-access coherence requires that either a is ordered before b, which implies that b cannot return 0x00 for byte $x + 1$, or b is ordered before a, which entails that b cannot return 0x11 for byte x, and therefore MP+sis is forbidden by per-access coherence. Per-byte coherence, by itself (ignoring other architecture restrictions), does not forbid MP+sis. The per-byte total orders $b \leq a$ for location x, and $a \leq b$ for location $x + 1$, allow the read of Thread 1 to return the value 0x1100.

Observe that per-access coherence entails per-byte coherence, as the partial order over memory accesses, when restricted to memory accesses that include a particular one-byte location, forms a total order over those memory accesses. The additional order that per-access coherence provide over per-byte coherence is called *single-copy atomicity* and it is discussed in the next subsection.

Per-access coherence is a better match to microarchitecture, where writes propagate, and win or lose coherence races, as atomic units. It also appears to be simpler algorithmically. The PLDI11 operational memory model of the Power architecture constructs the per-access coherence order explicitly (). The POP operational memory model () constructs a partial order over memory accesses and barrier events, that when restricted to memory accesses is a per-access coherence order. In Herd, the axiomatic models framework, it is more convenient to separate single-copy atomicity from coherence, and therefore per-byte coherence is used.

In our litmus diagrams the co edges depict per-access coherence, as it is less cluttered, but the fr edges are computed, as described below, from the per-byte coherence induced from the per-access coherence order of the co edges.

To make litmus tests more readable we usually choose memory values of the form $0xc_{k-1}k-1\ldots c_1 1 c_0 0$, where c_i implies the intended place of this value in the per-byte coher-

$$\mathsf{fr} = \{(r, s, w) | \exists w', s'.(w', s', r) \in \mathsf{rf} \wedge s = \{b \in s' | (w', w) \in \mathsf{co}_b\} \wedge s \neq \emptyset\}$$

Figure 16.56: Formal definition of fr.

ence order of the byte with offset i. For example, the value that is written by W x[0..1]=0x2110 is composed of two bytes, 0x21 and 0x10. The byte with value 0x10 is written to the byte location $x[0]$, as implied by the "0", and it is intended to be the first write to this byte location in per-byte coherence order, as implied by the "1". The byte with value 0x21 is written to the byte location $x[1]$, as implied by the "1", and it is intended to be the second write to this byte location in per-byte coherence order, as implied by the "2".

Note that MIX1 and MP+sis assume little-endianness, as do all the litmus tests that appear in this thesis. Arm, Power and RISC-V all support little- and big-endian modes. When running litmus tests on a machine for this thesis, the machine's default endianness is used. Except for Power7 which is big-endian, the default mode of all the machines that were used for running litmus tests is little-endian. Since the endianness of the Power7 machine is different from the one assumed in the included litmus tests, the tests that were run on that machine are slightly different. For example, in the big-endian version of MIX1 that was tested on the Power7 machine, the store of Thread 0 writes the value 0x10111213 instead of 0x13121110, and the load of Thread 1 is expected to read the value 0x10112213 instead of 0x13221110.

As the from-reads relation is derived from the read-from and coherence relations, in the mixed-size setting it is labelled with an appropriate footprint. The mixed-size from-reads relation is computed by composing the converse of read-from with per-byte coherence. The formal definition of fr is given in Fig. 16.56.

Note that in the MIX1 diagram there is no fr edge from c to b despite there being an rf edge from a to c and a co edge from a to b, which in the non-mixed-size setting would entail an fr edge from c to b. This is because the label of the rf edge from a to c is x[0..1,3], and the memory footprint of b is x[2], which is disjoint.

16.2.3 Single-copy Atomicity

The per-access definition of coherence from the previous subsection is stronger (i.e. allows less behaviours) than the per-byte definition of coherence, as it requires all the overlapping bytes of two accesses to have consistent order. This consistency between bytes of overlapping footprints is called single-copy atomicity [**?**], and an architecture may guarantee it for the set of bytes accessed by an instruction in some cases, and not in other cases.

In the Arm, Power, and RISC-V architectures, the size of a single-copy atomic memory access is always some power of 2, and the location is always aligned to that size. It follows, that if two single-copy atomic memory accesses are ordered in per-access coherence, $a \leq b$, there exists a single-copy atomic memory access c in the execution, such that $a \leq c \leq b$, and the footprint of c subsumes the footprints of a and b.

Single-copy atomicity is not the opposite of multi-copy atomicity. Multi-copy atomicity guarantees that when a write is observable to any other thread, it is also observable to all other threads (see §16.1.14), whereas single-copy atomicity guarantees that when a byte of a multi-byte write is observable to a thread, all other bytes of the write are also observable to that thread.

An execution is *single-copy atomic consistent* if it is per-access coherent over single-copy memory access events. Alternatively, an execution is single-copy atomic consistent if it is per-byte coherent, and in addition, for every two byte locations x and y, and for every two single-copy memory access events a and b that both access both byte locations x and y, a and b are ordered in the same direction in the per-byte orders for x and for y.

The Arm, Power and RISC-V architectures require that a properly aligned regular memory access instruction is single-copy atomic, where properly aligned means that the location the instruction access (lowest byte location) is divisible by the number of bytes being accessed. The Armv8-A [**?**, p. 93] and RISC-V [**?**, p. 82] architectures treat a misaligned regular memory access as an unordered set of single-byte single-copy-atomic accesses. The Power architecture used to be similar, but now provides stronger requirements [**?**, p. 811] (first match applies):

- A misaligned access of 16 bytes that is aligned to 8 bytes is performed as two disjoint single-copy atomic accesses of 8 bytes.
- A misaligned access of at least 8 bytes that is aligned to 4 bytes is performed as multiple disjoint single-copy atomic accesses of 4 bytes.
- A misaligned access of at least 4 bytes that is aligned to 2 bytes is performed as multiple disjoint single-copy atomic accesses of 2 bytes.
- Otherwise, a misaligned access is performed as multiple disjoint single-copy atomic accesses of one byte.

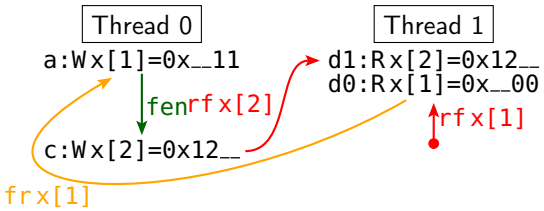
The additional requirements in Power allow for more efficient copying of arrays, without breaking single-copy atomicity of the array's elements, when the array is aligned to the size of the elements in the array.

Memory access events in litmus execution graphs represent single-copy atomic memory accesses. Hence, each properly aligned load or store instruction will be associated with a single memory access event in an execution, reflecting the architecture specifications for such instruction. Each misaligned memory access instruction will be associated with multiple memory access events.

Consider MP+sis (Fig. 16.55) again. A hypothetical microarchitecture might exhibit MP+sis by decomposing the store of Thread 0 to two one-byte writes (for whatever reason), and then performing the load of Thread 1 after the write to $x[1]$ and before the write to $x[0]$. Or alternatively, a microarchitecture might decompose the load of Thread 1 to two one-byte reads, and then perform the store of Thread 0 before the read from $x[1]$ and after the read from $x[0]$.

Recall that the symbolic memory locations in litmus tests are 8-byte aligned (§15.4), and as such, the store and load of MP+sis are both properly aligned, and therefore both are single-copy atomic in the Arm, Power, and RISC-V architectures. Therefore, MP+sis violates single-copy atomicity and therefore the test is forbidden by Arm, Power and RISC-V.

Contrast MP+sis with MP+fen+si1. Here, the writing of the data and the flag in Thread 0



MP+dmb.sy+si1 AArch64

Initial state: 0:X2=0x12; 0:X1=0x11; 0:X0=x; 1:X1=0x1; 1:X0=x; x=0x0;

Thread 0	Thread 1
STRB W1,[X0,#1] //a	LDRH W2,[X0,X1] //d0,d1
DMB SY //b	
STRB W2,[X0,#2] //c	

Final: 1:X2=0x1200;

H/W: - - - - - - - - - - - - - - - - -

MP+fence.rw.rw+si1 RISC-V

Initial state: 0:x2=0x12; 0:x1=0x11; 0:x4=x; 1:x4=x; x=0x0;

Thread 0	Thread 1
sb ra, 1(tp) //a	lh sp, 1(tp) //d0,d1
fence rw, rw //b	
sb sp, 2(tp) //c	

Final: 1:x2=0x1200;

H/W: -

MP+sync+si1 Power

Initial state: 0:r2=0x12; 0:r1=0x11; 0:r4=x; 1:r4=x; x=0x0;

Thread 0	Thread 1
stb r1,1(r4) //a	lhz r2,1(r4) //d0,d1
sync //b	
stb r2,2(r4) //c	

Final: 1:r2=0x1200;

H/W: -

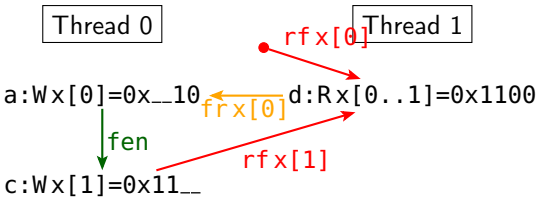
Figure 16.57: Litmus test MP+fen+si1

are done in two distinct instructions, that are separated by a fence, and the reading of the flag

and the data is done by a single instruction with a misaligned memory access that architecturally gives rise to two single-copy atomic one-byte reads. The 1 in si1 in the litmus test name indicates that the associated memory access is offset by 1 byte from an address that is cache-size aligned. In the litmus diagram, the two one-byte reads, $d0$ and $d1$, from the single load instruction of Thread 1, appear adjacent to each other, with no edges between them.

When a microarchitecture executes the load of Thread 1 it is allowed to perform it as two one-byte reads, one from $x[1]$ ($d0$ in the litmus diagram) and one from $x[2]$ ($d1$ in the litmus diagram), in arbitrary order. In particular, the microarchitecture may do the read from $x[1]$ first, before any of the writes of Thread 0 has been performed, and it may do the read from $x[2]$ after the writes of Thread 0 has been performed and made visible to Thread 1. Hence, the load of Thread 1 may return the value 0x1200, where the byte value 0x12 was read from the write c of Thread 0, and the byte value 0x00 was read from the initial value of $x[1]$.

Similar behaviour is not allowed by the Arm, Power, and RISC-V architectures for MP+fen+si,



MP+dmb.sy+si AArch64

Initial state: 0:X2=0x11; 0:X1=0x10; 0:X0=x; 1:X0=x; x=0x0;

Thread 0	Thread 1
STRB W1,[X0] //a	LDRH W2,[X0]//d
DMB SY //b	
STRB W2,[X0,#1]//c	

Final: 1:X2=0x1100;

H/W: - - - - - - - - - - - - - - - - -

MP+fence.rw.rw+si RISC-V

Initial state: 0:x2=0x11; 0:x1=0x10; 0:x4=x; 1:x4=x; x=0x0;

Thread 0	Thread 1
sb ra, 0(tp) //a	lh sp, 0(tp) //d
fence rw, rw //b	
sb sp, 1(tp) //c	

Final: 1:x2=0x1100;

H/W: -

MP+sync+si Power

Initial state: 0:r2=0x11; 0:r1=0x10; 0:r4=x; 1:r4=x; x=0x0;

Thread 0	Thread 1
stb r1,0(r4) //a	lhz r2,0(r4)//d
sync //b	
stb r2,1(r4) //c	

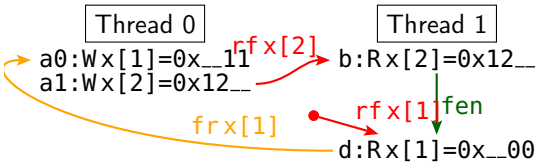
Final: 1:r2=0x1100;

H/W: -

Figure 16.58: Litmus test MP+fen+si

as here the load of Thread 1 is properly aligned and therefore must not be observed as two reads. This, together with the fence in Thread 0, ensures that if the load reads the value 0x11 for the byte location $x[1]$, it cannot read the value 0x00 for the byte location $x[0]$.

For store instructions, MP+si1+fen and MP+si+fen demonstrate the same point, misaligned



MP+si1+dmb.sy AArch64

Initial state: 0:X2=0x1; 0:X1=0x1211; 0:X0=x; 1:X0=x; x=0x0;

Thread 0	Thread 1
STRH W1,[X0,X2]//a0,a1	LDRB W1,[X0,#2]//b
	DMB SY //c
	LDRB W2,[X0,#1]//d

Final: 1:X1=0x12; 1:X2=0x0;

H/W: - - - - - - - - - - - - - - - - -

MP+si1+fence.rw.rw RISC-V

Initial state: 0:x6=x; 0:x5=0x1211; 1:x6=x; x=0x0;

Thread 0	Thread 1
sh t0, 1(t1) //a0,a1	lb t0, 2(t1) //b
	fence rw, rw//c
	lb t2, 1(t1) //d

Final: 1:x5=0x12; 1:x7=0x0;

H/W: -

MP+si1+sync Power

Initial state: 0:r2=0x1211; 0:r1=x; 1:r1=x; x=0x0;

Thread 0	Thread 1
sth r2,1(r1)//a0,a1	lbz r2,2(r1)//b
	sync //c
	lbz r3,1(r1)//d

Final: 1:r2=0x12; 1:r3=0x0;

H/W: -

Figure 16.59: Litmus test MP+si1+fen

and properly aligned respectively.

In addition to the misaligned litmus tests above, similar variants with offsets of 3, 7, 15, 31, 63, and 127 were also tested on real hardware. Microarchitecturally, one would expect at

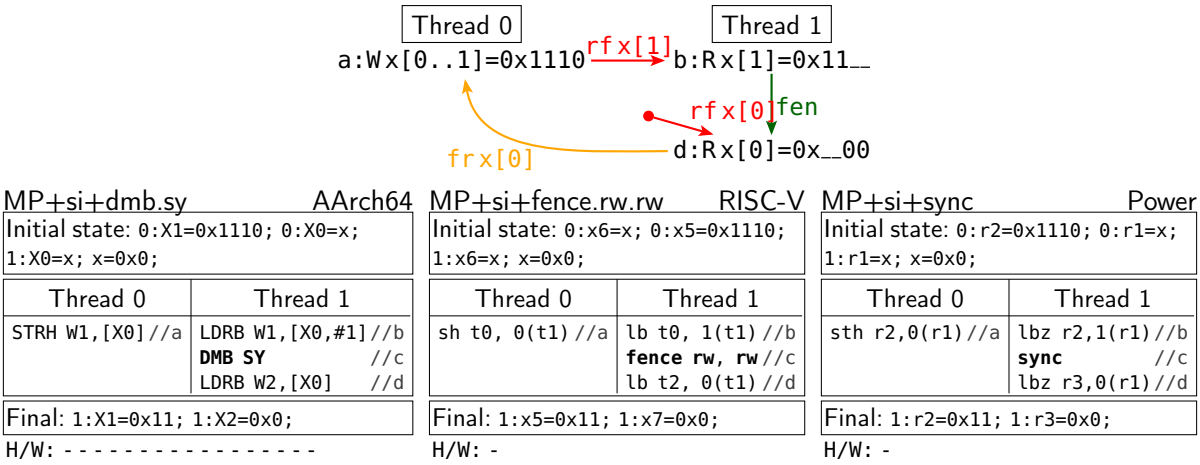


Figure 16.60: Litmus test MP+si+fen

least memory accesses whose footprint spans a cache-line boundary to be split (otherwise one is in the realm of hardware transactional memory implementations, to provide atomic access to multiple cache lines while avoiding a deadlock), but splitting at finer granularities is also observed. Such splitting could be the result of store-buffer width, and other microarchitecture mechanisms, such as write-forwarding. The architectures explicitly do not guarantee single-copy atomicity for misaligned accesses within cache lines, or indeed commit to any particular cache-line sizes, so programmers should not rely on that.

The split of misaligned memory access instructions into multiple single-copy atomic accesses means that some of those accesses might execute out-of-order with another instruction, while other accesses execute in-order with the same instruction. Consider MP+fen+pos-si1, in which

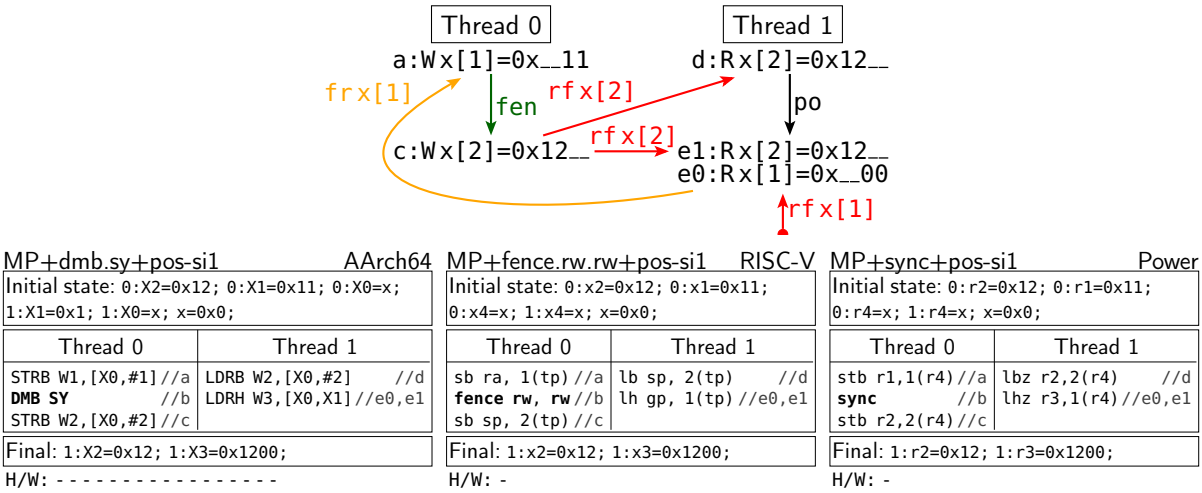


Figure 16.61: Litmus test MP+fen+pos-si1

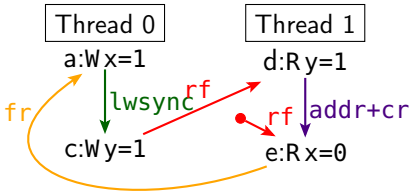
the second load of Thread 1 is misaligned and has a memory footprint that overlaps with the first load. Because the second load is misaligned, the Arm, Power, and RISC-V architectures allow the second load to be performed as two one-byte memory reads, as illustrated in Fig. 16.61, and because read $e1$ does not overlap with read d, $e1$ can be performed out-of-order with d, making this litmus test allowed by these architectures.

16.2.4 Atomicity of register accesses

In the Arm, Power, and RISC-V architectures, general purpose register (GPR) writes always write to all the bits of the register. When an instruction manipulates data that is smaller than the GPR it is written to, the data is either zero or sign extended to fit the register size, depending on the architecture and the instruction. Hence, the values of all the bits that a GPR read reads come from a single GPR write, the most recent preceding write, in program-order, to that GPR.

For special-purpose registers this is not always the case. Some instructions can access (read or write) particular bits of special-purpose registers. It is then up to the architecture to define in which circumstances a write/read pair to such a special-purpose register induces dependency that restricts out-of-order execution. The Arm, Power, and RISC-V architectures manuals are not explicit about this, perhaps because it was not considered, or perhaps intending to allow the most relaxed behaviour, allowing hardware implementors the most freedom. The relaxed behaviour is also easier to model mathematically, and is the behaviour implemented in the architecture models that will be discussed in later chapters.

In the Power architecture, CR (*Condition Register*) is a special-purpose 32-bit register, partitioned to eight 4-bit fields (CR0,...,CR7), that reflect the result of certain operations, and can be tested with conditional branch instructions. All four bits of a single CR field can be accessed using the `mtocrf` (write to a field) and `mfocrf` (read from a field) instructions, and specific bits of the CR can be accessed using the `crnand` instruction which reads two specified bits of CR, and writes the negated AND of their values to a third specified bit. Consider MP+lwsync+addr-cr,



MP+lwsync+addr-cr Power

Initial state: `0:r4=y; 0:r2=x; 1:r5=x; 1:r2=y;`

Thread 0	Thread 1
`li r1,1`	`lwz r1,0(r2) //d`
`stw r1,0(r2) //a`	`//move to cr field 3`
`lwsync` `//b`	`mtocrf 16,r1`
`li r3,1`	`//move from cr field 4`
`stw r3,0(r4) //c`	`mfocrf r2,8`
	`xor r3,r2,r2`
	`lwzx r4,r3,r5 //e`

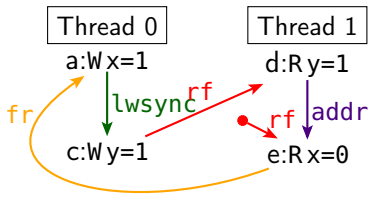
Final: `1:r1=1; 1:r4=0;`

H/W: -

Figure 16.62: Litmus test MP+lwsync+addr-cr

where the value returned from the first load of Thread 1 is written to (some bits of) CR, which is followed by a read of (some bits of) CR, which feeds the address computation of the second load of Thread 1. Note that here one has to look at the assembly, not just the diagram, as the diagram just shows the memory accesses, with just an addr+cr edge, not the register accesses. If the fact that the write and read of CR involve different fields were to be ignored, there would appear to be a data-flow from the first load to the second load, which would induce an address dependency between them, which would restrict their out-of-order execution (as in MP+lwsync+addr, Fig. 16.63). However, as the write to CR writes just the four bits of the CR3 field, and the read form CR reads only from the four bits of the CR4 field, there is no actual data-flow between the accesses of CR. The architecture design has a free choice here whether to regard that as a respected dependency or not. Experimental results from running the litmus test on Power machines show the out-of-order behaviour is observed. This has been discussed with the Power designers, and this is intentional. Therefore, the PLDI11 model of the Power architecture does not restrict the out-of-order execution of those two loads.

MP+lwsync+addr-cr5 extends MP+lwsync+addr-cr by adding, between the accesses to CR3 and CR4, a write to bit 0 of CR5, of the negation of bit 0 of CR3, and a write to bit 0 of CR4 of the negation of bit 1 of CR5. Again, if the fact that the write and read of CR5 involve different



MP+lwsync+addr Power

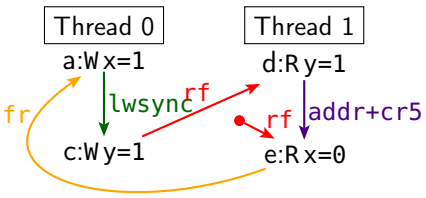
Initial state: 0:r3=y; 0:r2=x; 0:r1=1; 1:r5=x; 1:r2=y; 1:r1=0; 1:r4=0; y=0; x=0;

Thread 0	Thread 1
std r1,0(r2)//a	ld r1,0(r2) //d
lwsync //b	xor r3,r1,r1
std r1,0(r3)//c	ldx r4,r3,r5//e

Final: 1:r1=1; 1:r4=0;

H/W: N

Figure 16.63: Litmus test MP+lwsync+addr



MP+lwsync+addr-cr5 Power

Initial state: 0:r4=y; 0:r2=x; 1:r5=x; 1:r2=y;

Thread 0	Thread 1
li r1,1	lwz r1,0(r2) //d
stw r1,0(r2)//a	//move to cr field 3
lwsync //b	mtocrf 16,r1
li r3,1	//set bit 0 of field 5 to NOT of
stw r3,0(r4)//c	//bit 0 of field 3
	crnand 20,12,12
	//set bit 0 of field 4 to NOT of
	//bit 1 of field 5
	crnand 16,21,21
	//move from cr field 4
	mfocrf r2,8
	xor r3,r2,r2
	lwzx r4,r3,r5 //e

Final: 1:r1=1; 1:r4=0;

H/W: -

Figure 16.64: Litmus test MP+lwsync+addr-cr5

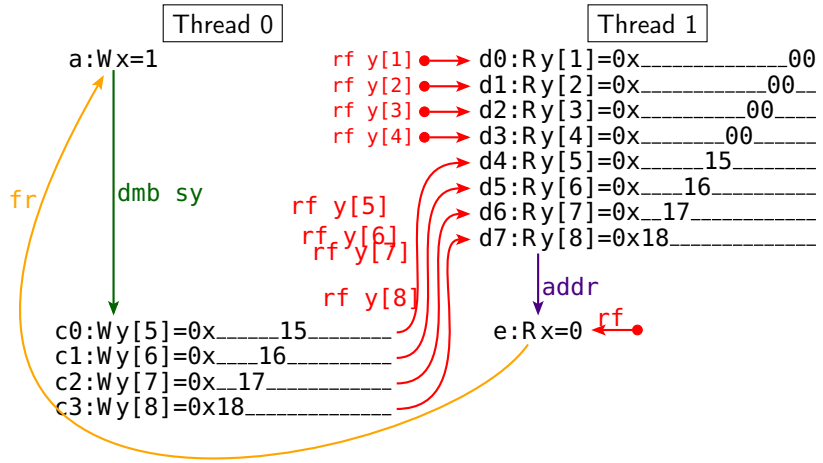
bits is ignored for a moment, there appears to be a data-flow from the first load to the second load, which induces an address dependency between them, which would normally restrict their out-of-order execution. However, as the write to CR5 writes to bit 0 of the field, and the read form CR5 reads only bit 1 of the field, there is no data-flow between the accesses of CR5, and therefore the Power architecture does not restrict the out-of-order execution of the two loads. This out-of-order behaviour was not observed in the experimental results for the litmus test on Power machines.

In Arm the situation is more complicated. The Armv8 architecture defines an abstraction of the process state, *PSTATE*, composed of fields each of which can be accessed as special-purpose registers, though most of those are not accessible by application-level code (EL0), and therefore are out of scope for this thesis. The NZCV field of the PSTATE is accessible by application-level code as the 64-bit NZCV special-purpose register. The NZCV register holds 4 flags: the negative condition flag in bit index 31 (N), the zero condition flag in bit index 30 (Z), the carry condition flag in bit index 29 (C), and the overflow condition flag in bit index 28 (V). The other bits of the NZCV register are unused.

In versions of Armv8 older than Armv8.4-A, the pseudocode that defines the behaviour of instructions in the Arm Architecture Reference Manual [**?**] always writes all four flags together. For example, the pseudocode of the `ANDS` instruction writes to the N and Z flags values computed from the operands of the instruction, and 0 to the C and V flags. Hence, it appears that accessing NZCV is similar to accessing GPRs in that a read of NZCV always returns the value that was written by the most recent write (in program-order) to NZCV, and therefore such a sequence will always induce a dependency (as GPRs do). Armv8.4-A introduced instructions that can write to individual flags of NZCV, without changing the values of other flags. For example, the

RMIF instruction can be used to set, clear, or leave untouched any combination of the NZCV flags. Using the RMIF instruction to write just the N flag, and the SBC instruction to read just the C flag, an MP-like litmus test can be constructed, MP+dmb.sy+addrNC, in which the address dependency between the loads of Thread 1 passes through disjoint bits of the NZCV register.

Another aspect of register atomicity is whether a register read that accesses a strict subset of the bits that the most recent write to the same register wrote, can appear to be performed before the register write is completed, as the read only requires the overlapping bits to be available. For example, consider MP+dmb.sy+si1-addr, in which the second load of Thread 1 is misaligned.



MP+dmb.sy+si1-addr AArch64

Initial state: 0:X4=0x18171615; 0:X3=0x1; 0:X2=0x5; 0:X1=y; 0:X0=x;
1:X5=0x0; 1:X3=0x0; 1:X2=0x1; 1:X1=x; 1:X0=y; x=0x0; y=0x0;

Thread 0		Thread 1
STR X3,[X0]	//a	LDR X3,[X0,X2] //d0,d1,d2,d3,d4,d5,d6,d7
DMB SY	//b	EOR W4,W3,W3
STR W4,[X1,X2] //c0,c1,c2,c3		LDR X5,[X1,W4,SXTW] //e

Final: 1:X3=0x1817161500000000; 1:X5=0x1;

H/W: - - - - - - - - - - - - - - - -

Figure 16.65: Litmus test MP+dmb.sy+si1-addr

This litmus test is very similar to MP+dmb.sy+addr (Fig. 16.11) except that the first load of Thread 1 is misaligned, and the address dependency feeds only from the bottom half of the register, the value of which is written by the preceding load after reading it from the initial memory value. A hypothetical microarchitecture could exhibit this behaviour, by allowing the first load of Thread 1 to read from $y[1-4]$ the initial value, and write it to the bottom 4 bytes of the output register, before reading from $y[5-8]$, and before writing to the top 4 bytes of the output register. The address dependency would then be resolved, as it only reads from the bottom 4 bytes of the register, and the second load of Thread 1 would be performed, reading 0, before any of the writes of Thread 0 has been performed.

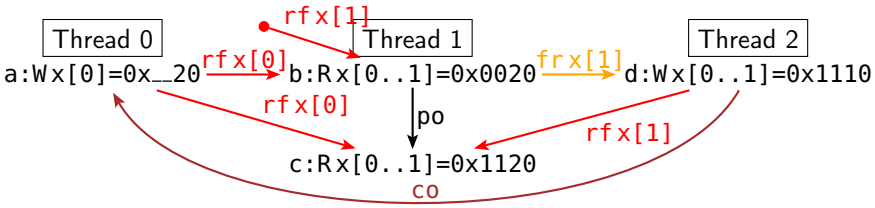
The Arm, Power, and RISC-V architectures all forbid this behaviour, because even though the first load is allowed to be performed as 8 separate reads, the value that the load writes to the output register must appear to become available to program-order succeeding instructions atomically.

16.2.5 Mixed-size Multi-copy Atomicity

In the mixed-size setting, non-multi-copy atomicity allows stores with overlapping memory footprints to be observed by a hardware thread in order that is reversed to their coherence order. This is surprising, and weaker than the Power reference manual suggests [**?**, p. 813]: *Atomic*

stores to a given location are coherent if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order; and also: *a processor or mechanism can never load a "newer" value first and then, later, load an "older" value*.

For example, consider WRR+2W+sis, in which the store a of Thread 0, and the store d of



WRR+2W+sis Power

Initial state: 0:r2=0x20; 0:r1=x; 1:r1=x; 2:r2=0x1110; 2:r1=x; x=0x0;

Thread 0	Thread 1
stb r2,0(r1) //a	lhz r2,0(r1) //b lhz r3,0(r1) //c
	Thread 2
	sth r2,0(r1) //d

Final: 1:r2=0x20; 1:r3=0x1120; x=0x1120;

H/W: -

WRR+2W+sis AArch64

Initial state: 0:X0=0x20; 0:X1=x; 1:X1=x; 2:X0=0x1110; 2:X1=x; x=0x0;

Thread 0	Thread 1
STRB W0,[X1] //a	LDRH W0,[X1] //b LDRH W2,[X1] //c
	Thread 2
	STRH W0,[X1] //d

Final: 1:X0=0x20; 1:X2=0x1120; x=0x1120;

H/W: - - - - - - - - - - - - - - - -

WRR+2W+sis RISC-V

Initial state: 0:x2=0x20; 0:x1=x; 1:x1=x; 2:x2=0x1110; 2:x1=x; x=0x0;

Thread 0	Thread 1
sb sp, 0(ra) //a	lh sp, 0(ra) //b lh gp, 0(ra) //c
	Thread 2
	sh sp, 0(ra) //d

Final: 1:x2=0x20; 1:x3=0x1120; x=0x1120;

H/W: -

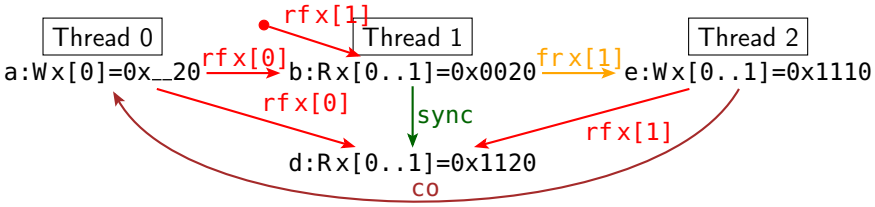
Figure 16.66: Litmus test WRR+2W+sis

Thread 2 have overlapping footprints. The Power architecture, which is non-multi-copy atomic, allows the execution of this litmus test where d is (per-access) coherence-before a, and the first load of Thread 1 reads $x[0]$ from a, before d becomes observable to Thread 1, as evident by reading from the initial memory value of $x[1]$. This behaviour is consistent with the excerpts from the Power reference manual above, with respect to the per-byte coherence interpertation, but if the per-access interpertation of coherence is considered, it is not clear that those excerpts still hold. Note that all the memory access instructions in this litmus test are properly aligned and therefore they are each associated with exactly one single-copy atomic memory access.

Microarchitecturally, this behaviour can occur in several ways. A simple one is when Threads 0 and 1 share some level of cache that is not shared with Thread 2. In that configuration, b can update the cache level that is shared between Threads 0 and 1, before the coherence race between a and c is determined. Thread 1 can then read the value 0x0020 for $x[0..1]$, from the shared cache. Finally, c wins the coherence race with a (i.e. c is sequenced before a), the shared cache is updated, and Thread 1 reads the bytes at $x[0..1]$ again (for the second load), this time returning the value 0x1120.

Although not immediately obvious, this non-multi-copy atomic behaviour has significant im-

plications on the ability of programmers of Power systems to restore sequential consistency using fences. Consider WRR+2W+sis+VAR1, which is similar to WRR+2W+sis, with the addition of



WRR+2W+sis+VAR1 Power

Initial state: 0:r2=0x20; 0:r1=x; 1:r1=x; 2:r2=0x1110; 2:r1=x; x=0x0;

Thread 0	Thread 1	Thread 2

stb r2,0(r1) //a	lhz r2,0(r1) //b
	sync //c
	lhz r3,0(r1) //d
		sth r2,0(r1) //e

Final: 1:r2=0x20; 1:r3=0x1120; x=0x1120;

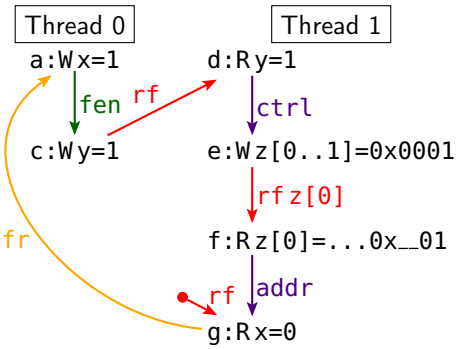
H/W: -

Figure 16.67: Litmus test WRR+2W+sis+VAR1

a barrier between the loads in Thread 1, which one might hope can only exhibit sequentially consistent behaviour, as all the instructions are separated by a fence. This litmus test is allowed by the Power architecture and observed on Power machines. This behaviour is not sequentially consistent: there is no total order over the instructions in which each read reads each byte from the most recent write to that byte (in the total order). Moreover, this behaviour cannot be eliminated by inserting more fences, as Threads 0 and 2 each have only one memory access, and the two memory accesses in Thread 1 are already separated by a fence. This contradicts a standard result for relaxed memory models, which is also a property that architectures have been thought to intend and to guarantee. The only exception that the author of this thesis was previously aware of is the Itanium architecture [**?**].

16.2.6 Mixed-size write-forwarding

The non-mixed-size litmus test PPOCA (Fig. 16.22) demonstrates simple write-forwarding, where a single speculative write (e) satisfies a read (f) by forwarding. In the mixed-size setting the footprints of the speculative write and the read that it satisfies do not have to match perfectly, for example in the litmus test MP+fen+ctrl-si-rfi-addr, the speculative write that is being forwarded writes to two bytes and the read to which it is being forwarded reads from just one of them. In the MP+fen+ctrl-rfi-si-addr the speculative write that is being forwarded writes one byte and the read to which it is being forwarded reads two bytes, one by the forwarding, and the other byte from the initial value in memory. Finally, in MP+fen+ctrl-rfi-si-addr+VAR1 two speculative one-byte writes to adjacent locations are forwarded to a two-byte read. All these variations of PPOCA are allowed by Arm, Power, and RISC-V.



AArch64

MP+dmb.sy+ctrl-si-rfi-addr

Initial state: 0:X2=y; 0:X1=x;
0:X0=0x1; 1:X7=x; 1:X3=z;
1:X2=0x1; 1:X1=y; 1:X0=0x0;
1:X4=0x0; 1:X6=0x0; z=0x0; y=0x0;
x=0x0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	CBNZ X0,LC00
STR X0,[X2] //c	LC00:
	STRH W2,[X3] //e
	LDRB W4,[X3] //f
	EOR X5,X4,X4
	LDR X6,[X7,X5] //g

Final: 1:X0=0x1; 1:X4=0x1;
1:X6=0x0;

H/W: - - - - - - - - - - - - - - - -

RISC-V

MP+fence.rw.rw+ctrl-si-rfi-addr

Initial state: 0:x7=y; 0:x6=x;
0:x5=0x1; 1:x12=x; 1:x8=z; 1:x7=0x1;
1:x6=y; 1:x5=0x0; 1:x9=0x0;
1:x11=0x0; z=0x0; y=0x0; x=0x0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	bne t0, zero, LC00
sd t0, 0(t2) //c	LC00:
	sh t2, 0(fp) //e
	lb s1, 0(fp) //f
	xor a0, s1, s1
	add a3, a2, a0
	ld a1, 0(a3) //g

Final: 1:x5=0x1; 1:x9=0x1; 1:x11=0x0;

H/W: -

Power

MP+sync+ctrl-si-rfi-addr

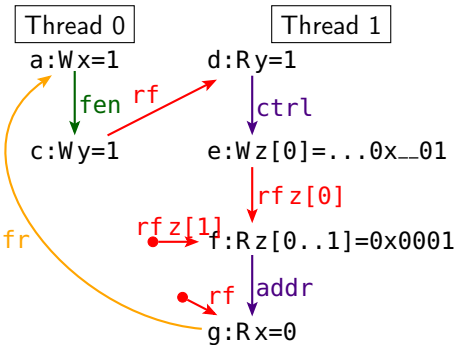
Initial state: 0:r3=y; 0:r2=x;
0:r1=0x1; 1:r8=x; 1:r4=z;
1:r3=0x1; 1:r2=y; 1:r1=0x0;
1:r5=0x0; 1:r7=0x0; z=0x0; y=0x0;
x=0x0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	cmpw r1,r1
std r1,0(r3) //c	bc 12,2,4
	LC00:
	sth r3,0(r4) //e
	lbz r5,0(r4) //f
	xor r6,r5,r5
	ldx r7,r6,r8 //g

Final: 1:r1=0x1; 1:r5=0x1;
1:r7=0x0;

H/W: -

Figure 16.68: Litmus test MP+fen+ctrl-si-rfi-addr



AArch64

MP+dmb.sy+ctrl-rfi-si-addr

Initial state: 0:X2=y; 0:X1=x;
0:X0=0x1; 1:X7=x; 1:X3=z;
1:X2=0x1; 1:X1=y; 1:X0=0x0;
1:X4=0x0; 1:X6=0x0; z=0x0; y=0x0;
x=0x0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	CBNZ X0,LC00
STR X0,[X2] //c	LC00:
	STRB W2,[X3] //e
	LDRH W4,[X3] //f
	EOR X5,X4,X4
	LDR X6,[X7,X5] //g

Final: 1:X0=0x1; 1:X4=0x1;
1:X6=0x0;

H/W: - - - - - - - - - - - - - - - -

RISC-V

MP+fence.rw.rw+ctrl-rfi-si-addr

Initial state: 0:x7=y; 0:x6=x;
0:x5=0x1; 1:x12=x; 1:x8=z; 1:x7=0x1;
1:x6=y; 1:x5=0x0; 1:x9=0x0;
1:x11=0x0; z=0x0; y=0x0; x=0x0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	bne t0, zero, LC00
sd t0, 0(t2) //c	LC00:
	sb t2, 0(fp) //e
	lh s1, 0(fp) //f
	xor a0, s1, s1
	add a3, a2, a0
	ld a1, 0(a3) //g

Final: 1:x5=0x1; 1:x9=0x1; 1:x11=0x0;

H/W: -

Power

MP+sync+ctrl-rfi-si-addr

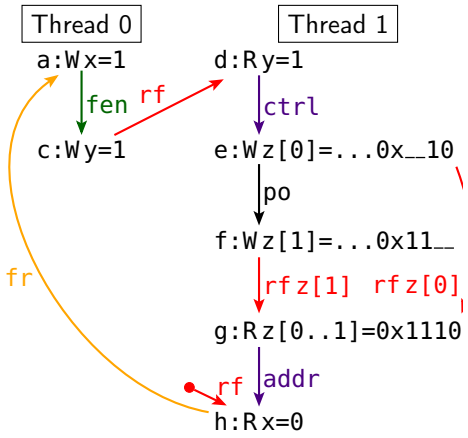
Initial state: 0:r3=y; 0:r2=x;
0:r1=0x1; 1:r8=x; 1:r4=z;
1:r3=0x1; 1:r2=y; 1:r1=0x0;
1:r5=0x0; 1:r7=0x0; z=0x0; y=0x0;
x=0x0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	cmpw r1,r1
std r1,0(r3) //c	bc 12,2,4
	LC00:
	stb r3,0(r4) //e
	lhz r5,0(r4) //f
	xor r6,r5,r5
	ldx r7,r6,r8 //g

Final: 1:r1=0x1; 1:r5=0x1;
1:r7=0x0;

H/W: -

Figure 16.69: Litmus test MP+fen+ctrl-rfi-si-addr



AArch64
MP+dmb.sy+ctrl-rfi-si-addr+VAR1

Initial state: 0:X2=y; 0:X1=x;
0:X0=0x1; 1:X7=x; 1:X3=z;
1:X8=0x11; 1:X2=0x10; 1:X1=y;
1:X0=0x0; 1:X4=0x0; 1:X6=0x0;
z=0x0; y=0x0; x=0x0;

Thread 0	Thread 1
STR X0,[X1] //a	LDR X0,[X1] //d
DMB SY //b	CBNZ X0,LC00
STR X0,[X2] //c	LC00:
	STRB W2,[X3] //e
	STRB W8,[X3,#1] //f
	LDRH W4,[X3] //g
	EOR X5,X4,X4
	LDR X6,[X7,X5] //h

Final: 1:X0=0x1; 1:X4=0x1110;
1:X6=0x0;

H/W: - - - - - - - - - - - - - - - -

RISC-V
MP+fence.rw.rw+ctrl-rfi-si-addr+VAR1

Initial state: 0:x7=y; 0:x6=x;
0:x5=0x1; 1:x12=x; 1:x8=z;
1:x14=0x11; 1:x7=0x10; 1:x6=y;
1:x5=0x0; 1:x9=0x0; 1:x11=0x0;
z=0x0; y=0x0; x=0x0;

Thread 0	Thread 1
sd t0, 0(t1) //a	ld t0, 0(t1) //d
fence rw, rw //b	bne t0, zero, LC00
sd t0, 0(t2) //c	LC00:
	sb t2, 0(fp) //e
	sb a4, 1(fp) //f
	lh s1, 0(fp) //g
	xor a0, s1, s1
	add a3, a2, a0
	ld a1, 0(a3) //h

Final: 1:x5=0x1; 1:x9=0x1110;
1:x11=0x0;

H/W: -

Power
MP+sync+ctrl-rfi-si-addr+VAR1

Initial state: 0:r3=y; 0:r2=x;
0:r1=0x1; 1:r8=x; 1:r4=z;
1:r9=0x11; 1:r3=0x10; 1:r2=y;
1:r1=0x0; 1:r5=0x0; 1:r7=0x0;
z=0x0; y=0x0; x=0x0;

Thread 0	Thread 1
std r1,0(r2) //a	ld r1,0(r2) //d
sync //b	cmpw r1,r1
std r1,0(r3) //c	bc 12,2,4
	LC00:
	stb r3,0(r4) //e
	stb r9,1(r4) //f
	lhz r5,0(r4) //g
	xor r6,r5,r5
	ldx r7,r6,r8 //h

Final: 1:r1=0x1; 1:r5=0x1110;
1:r7=0x0;

H/W: -

Figure 16.70: Litmus test MP+fen+ctrl-rfi-si-addr+VAR1

Part III

Systems concurrency

Part IV

Reflections, related work, and history

Bibliography

[1] C++ standards committee papers. https://www.open-std.org/JTC1/SC22/WG21/docs/papers/. Accessed 2023-09-01.

[2] *The SPARC Architecture Manual, V. 8*. SPARC International, Inc., 1992. Revision SAV080SI9308. http://www.sparc.org/standards/V8.pdf.

[3] *The SPARC Architecture Manual, V. 8*. SPARC International, Inc., 1992. Revision SAV080SI9308. https://sparc.org/technical-documents/#V8, https://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz. Accessed 2023-09-17.

[4] *The SPARC Architecture Manual, V. 9*. SPARC International, Inc./PTR Prentice Hall, 1992. Revision SAV09R1459912. https://sparc.org/technical-documents/#V9, https://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz. Accessed 2023-09-17.

[5] Linux Kernel mailing list, thread "spin_unlock optimization(i386)", 119 messages, Nov. 20–Dec. 7th, 1999, https://lkml.org/lkml/1999/11/20/76. Accessed 2023/09/10., 1999.

[6] cpp-threads mailing list archives. https://www.decadent.org.uk/pipermail/cpp-threads/, 2004–2013. Accessed 2023-09-01.

[7] *Power ISA^{TM} Version 2.03*. IBM, September 2006. 850 pages.

[8] *AMD64 Architecture Programmer's Manual (3 vols)*. Advanced Micro Devices, September 2007. rev. 3.14.

[9] Intel 64 architecture memory ordering white paper, 2007. Intel Corporation. SKU 318147-001.

[10] *Power ISA^{TM} Version 2.05*. IBM, October 2007.

[11] *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols)*. Intel Corporation, March 2010. rev. 34.

[12] ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile). https://developer.arm.com/documentation/ddi0487/, 2017. ARM DDI 0487B.a (ID033117). Issue B.a. 6354 pages. Accessed 2023-08-30.

[13] RISC-V Memory Model Task Group mailing list archives. https://lists.riscv.org/g/tech-memory-model-archive/, 2017. https://lists.riscv.org/g/tech-memory-model-archive/messages. Accessed 2023-09-01.

[14] ARM Architecture Reference Manual (for A-profile architecture). https://developer.arm.com/documentation/ddi0487/, November 2024. ARM DDI 0487. Issue L.a. 14568 pages. Accessed 2025-01-08.

[15] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. A load-buffer semantics for total store ordering. *Log. Methods Comput. Sci.*, 14(1), 2018.

[16] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application to the powerpc architecture. *IEEE Trans. Parallel Distributed Syst.*, 14(5):502–515, 2003.

[17] Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volumes 1-5. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf, June 2023. Accessed 2023-08-30. 3336 pages.

[18] ahadali5000, Alasdair Armstrong, Alexander Richardson, Aril Computer Corp. (for contributions by Scott Johnson), Ben Marshall, Bicheng Yang, Bilal Sakhawat, Brian Campbell, Chris Casinghino, Christopher Pulte, Martin Berger Codasip (for contributions by Tim Hutt, Ben Fletcher), dylux, eroom1966, Google LLC (for contributions by its employees), Hesham Almatary, Jan Henrik Weinstock, Jessica Clarke, Jon French, Martin Berger, Michael Sammler, Microsoft (for contributions by Robert Norton-Wright, Nathaniel Wesley Filardo), Muhammad Bilal Sakhawat, Nathaniel Wesley Filardo, Paul A. Clarke, Peter Rugg, Peter Sewell, Philipp Tomsich, Prashanth Mundkur, Rafael Sene, Rishiyur S. Nikhil (Bluespec, Inc.), Robert Norton-Wright, Shaked Flur, Thibaut Pérami, Thomas Bauereiss, VRULL GmbH (for contributions by its employees), William McSpaddden, and Xinlai Wan. Sail RISC-V instruction-set architecture (ISA) model, 2014–2024.

[19] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In Lawrence Snyder, editor, *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '93, Velen, Germany, June 30 - July 2, 1993*, pages 251–260. ACM, 1993.

[20] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

[21] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[22] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proc. TACAS*, 2011.

[23] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, l'Université Paris 7 – Denis Diderot, 2010. http://www0.cs.ucl.ac.uk/staff/J.Alglave/these.pdf.

[24] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 577–591. ACM, 2015.

[25] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016.

[26] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2):8:1–8:54, 2021.

[27] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2):8:1–8:54, 2021.

[28] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[29] Jade Alglave and Luc Maranget. herd "legacy" repository. https://github.com/herd/legacy/. Accessed 2023-10-23. Github contributors: Luc Maranget, John Wickerson, Kate Deplaix, Mark Batty, Shaked Flur, Susmit Sarkar, jacquev6, Gabriel Kerneis, nafe.

[30] Jade Alglave and Luc Maranget. The herdtools7 tool suite. diy.inria.fr, https://github.com/herd/herdtools7/. Accessed 2023-08-30.

[31] Jade Alglave and Luc Maranget. x86 tso mixed axiomatic model, 2020. https://github.com/herd/herdtools7/blob/master/herd/libdir/x86tso-mixed.cat. Accessed 2023-09-20.

[32] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 405–418. ACM, 2018.

[33] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. Linux kernel memory consistency model (linux kernel documentation), 2018–2023. https://kernel.googlesource.com/pub/scm/linux/kernel/git/torvalds/linux/+/refs/heads/master/tools/memory-model/README, https://kernel.googlesource.com/pub/scm/linux/kernel/git/torvalds/linux/+/refs/heads/master/tools/memory-model/Documentation/README. Accessed 2024-04-01.

[34] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Proc. TACAS: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 6605*, TACAS'11/ETAPS'11, pages 41–44. Springer-Verlag, 2011.

[35] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2):170–205, April 2012.

[36] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[37] ARM. ARM Barrier Litmus Tests and Cookbook, October 2008. PRD03-GENC-007826 2.0.

[38] Alasdair Armstrong. The isla-axiomatic tool. https://isla-axiomatic.cl.cam.ac.uk/. Accessed 2020-10-10.

[39] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte,

Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[40] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Proc. 33rd International Conference on Computer-Aided Verification*, volume 12759 of *Lecture Notes in Computer Science*, pages 303–316. Springer, July 2021.

[41] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models (extended version). *Formal Methods in System Design*, May 2023.

[42] Charles Babbage. On the mathematical powers of the calculating engine: [26. dec. 1837]. In *The Origins of Digital Computers: Selected Papers*, pages 19–54. Springer, 1837.

[43] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[44] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and opencl. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 634–648. ACM, 2016.

[45] Mark Batty, Scott Owens, Jean Pichon-Pharabod, Susmit Sarkar, and Peter Sewell. Cppmem: C/C++ memory model exploration tool, 2012–2019. [web interface].

[46] Mark John Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, 2014. 2015 SIGPLAN John C. Reynolds Doctoral Dissertation award and 2015 CPHC/BCS Distinguished Dissertation Competition winner.

[47] Thomas Bauereiss, Brian Campbell, Alasdair Armstrong, Alastair Reid, Kathryn E. Gray, Anthony Fox, Peter Sewell, and Arm Limited. Sail Armv9.4-A, Armv9.3-A and ARMv8.5-A instruction-set architecture (ISA) models, 2019, 2022, 2024.

[48] P. Becker, editor. *Programming Languages — C++. Final Committee Draft.* 2010. ISO/IEC JTC1 SC22 WG21 N3092.

[49] Adam Biltcliffe, Michael Dales, Sam Jansen, Thomas Ridge, and Peter Sewell. Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In *Proceedings of the 14th IEEE International Conference on Network Protocols (Santa Barbara)*, pages 117–126, November 2006. See also the SWIFT MAC Protocol: HOL Specification at `http://www.cl.cam.ac.uk/~pes20/optical/spec.pdf`.

[50] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the Sockets API. *J. ACM*, 66(1):1:1–1:77, December 2018.

[51] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of the ACM Conference on Computer Communications (Philadelphia),* published as Vol. 35, No. 4 of *Computer Communication Review*, pages 265–276, August 2005.

[52] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston)*, pages 55–66, January 2006.

[53] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, pages 113–124, New York, NY, USA, 2011. ACM.

[54] Hans Boehm, Martin Buchholz, David Holmes, Andrew Haley, and Erik Osterlund. Actual IRIW use case? (thread on jmm-dev mailing list). https://mail.openjdk.org/pipermail/jmm-dev/2020-September/000440.html, September 2020. Accessed 2023-09-07. See also https://github.com/openjdk/jdk/pull/387.

[55] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 68–78. ACM, 2008.

[56] William Collier. Personal communication, October 2024.

[57] William W. Collier. *Reasoning about parallel architectures*. Prentice Hall, 1992.

[58] William W. Collier. Archtest, 1994. https://www.mpdiag.com/. Accessed 2025-01-01.

[59] William W. Collier. Testing memory models. In *Ninth International Workshop on Microprocessor Test and Verification, MTV 2008, Austin, Texas, USA, 8-10 December 2008*, pages 14–17. IEEE Computer Society, 2008.

[60] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. Intel, November 2006. 253668-022US.

[61] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. Intel, July 2008. 253668-027US.

[62] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, June 2023. Accessed 2023-08-30. 5066 pages.

[63] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[64] Will Deacon. The armv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat, https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7#diff-0461c726950c4454a08bd97fbfd49252, 2016. Accessed 2023-10-05.

[65] Will Deacon, Jade Alglave, Nikos Nikoleris, and Artem Khyzha. The armv8 application level memory model, December 2024. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat and https://developer.arm.com/herd7. Accessed 2025-01-08.

[66] D. Dice. Java memory model concerns on Intel and AMD systems. http://blogs.sun.com/dave/entry/java_memory_model_concerns_on, January 2008.

[67] Paul Durbaba. Mechanising proofs of equivalence of operational and axiomatic formalisations of the x86-TSO memory model. Part III Project Dissertation, Department of Computer Science and Technology, University of Cambridge, May 2021.

[68] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[69] Shaked Flur. litmus-latex: Machinery for including litmus tests in latex documents, 2019–2023. https://github.com/litmus-tests/litmus-latex. Accessed 2023-10-22.

[70] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA)*, pages 608–621, January 2016.

[71] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Paris)*, pages 429–442, January 2017.

[72] Open POWER Foundation. Power ISA Version 3.1B. https://files.openpower.foundation/s/dAYSdGzTfW4j2r2, September 2021. Accessed 2023-08-30. 1562 pages.

[73] Open POWER Foundation. Power Instruction Set Architecture Specification Document, Power ISA Version 3.1C, May 2024. https://files.openpower.foundation/s/9izgC5Rogi5Ywmm, https://openpowerfoundation.org/specifications/isa/. Revision 99a1cac of 2023-06-22. Accessed 2025-01-08. 1495 pages.

[74] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213. https://riscv.org/technical/specifications/, December 2019. Contributors: Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Ken Dockser, Roger Espasa, Shaked Flur, Stefan Freudenberger, Marc Gauthier, Andy Glew, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau, Josh Scheid, Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert Watson, Derek Williams, Andrew Wright, Reinoud Zandijk, Sizhuo Zhang. 238 pages. Accessed 2023-08-30.

[75] FSF. Gcc 13.2 manual, 6.47.2 Extended Asm – Assembler Instructions with C Expression Operands, 2023. https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/Extended-Asm.html. Accessed 2023-09-17.

[76] Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 47–60, New York, NY, USA, 2020. Association for Computing Machinery.

[77] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki)*, pages 635–646, December 2015.

[78] Jim Handy. *The cache memory book*. Academic Press Professional, Inc., USA, 1993.

[79] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[80] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[81] Intel. Pentium pro processor specification update, January 1999. 242689-035. http://www.cpu-zone.com/Pentium/Pentium%20processor%20specification.pdf. Accessed 2023-09-17.

[82] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, fifth edition, December 2017.

[83] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, fourth edition, June 2018. Final draft at https://web.archive.org/web/20181230041359/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf.

[84] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30:7–15, March 2010.

[85] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 649–662. ACM, 2016.

[86] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

[87] Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. Cerberus-BMC: a principled reference semantics and exploration tool for concurrent and sequential C. In *Proc. 31st International Conference on Computer-Aided Verification*, July 2019.

[88] Stella Lau, Kayvan Memarian, Victor B. F. Gomes, Kyndylan Nienhuis, Justus Matthiesen, James Lingard, and Peter Sewell. Cerberus-BMC tool for exploring the behaviour of small concurrent C test programs with respect to an arbitrary axiomatic concurrency model, 2019. [web interface].

[89] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[90] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, 2009.

[91] Arm Ltd. Memory model tool: Herd7 simulator, 2025. https://developer.arm.com/herd7. Accessed 2025-01-08.

[92] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for power multiprocessors. In *Proc. CAV, 24th International Conference on Computer Aided Verification, LNCS 7358*, pages 495–512, 2012.

[93] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, October 2012. Draft.

[94] Kayvan Memarian. The Cerberus C semantics. Technical Report UCAM-CL-TR-981, University of Cambridge, Computer Laboratory, May 2023. PhD thesis.

[95] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO/IEC JTC1/SC22/WG14 N2311.

[96] Dominic Mulligan, Thomas Bauereiss, Kathryn E. Gray, Scott Owens, Peter Sewell, Thomas Tuerk, Basile Clement, Brian Campbell, Christopher Pulte, David Sheets, Fabian Immler, Frederic Loulergue, Francesco Zappa Nardelli, Gabriel Kerneis, James Lingard, Jean Pichon-Pharabod, Justus Matthiesen, Kayvan Memarian, Kyndylan Nienhuis, Lars Hupel, Mark Batty, Michael Greenberg, Michael Norrish, Ohad Kammar, Peter Boehm, Robert Norton, Sami Mäkelä, Shaked Flur, Stephen Kell, Thibaut Pérami, Thomas Bauereiss, Thomas Williams, Victor Gomes, and emersion. Lem, a tool for lightweight executable mathematics, 2010–2023.

[97] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, September 2014. ACM.

[98] John von Neumann. First draft of a report on the edvac. Technical report, 1945.

[99] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[100] Frank P. O'Connell and Steven W. White. POWER3: The next generation of PowerPC processors. *IBM Journal of Research and Development*, 44(6):873–884, 2000.

[101] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, pages 391–407, 2009.

[102] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In ***ECOOP 2010**: Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer, 2010.

[103] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, University of Cambridge, Computer Laboratory, March 2009. 52pp.

[104] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface ARM Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.

[105] Christopher Pulte. *The Semantics of Multicopy Atomic ARMv8 and RISC-V*. PhD thesis, University of Cambridge, 2018. https://www.repository.cam.ac.uk/handle/1810/292229.

[106] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2018.

[107] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. Promising-arm/risc-v: a simpler and faster operational concurrency model. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1–15. ACM, 2019.

[108] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022.

[109] Alastair Reid. Trustworthy specifications of arm® v8-a and v8-m system level architecture. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 161–168. IEEE, 2016.

[110] Tom Ridge. A rely-guarantee proof system for x86-tso. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'10, pages 55–70, Berlin, Heidelberg, 2010. Springer-Verlag.

[111] Tom Ridge, Michael Norrish, and Peter Sewell. A rigorous approach to networking: TCP, from implementation to protocol to service. In *Proceedings of the 15th International Symposium on Formal Methods (Turku, Finland), LNCS 5014*, pages 294–309, May 2008.

[112] RISC-V. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20240411. https://riscv.org/technical/specifications/, December 2021. Contributors: Krste Asanović and Peter Ashenden and Rimas Avižienis and Jacob Bachmeyer and Allen J. Baum and Jonathan Behrens and Paolo Bonzini and Ruslan Bukin and Christopher Celio and Chuanhua Chang and David Chisnall and Anthony Coulter and Palmer Dabbelt and Monte Dalrymple and Paul Donahue and Greg Favor and Dennis Ferguson and Marc Gauthier and Andy Glew and Gary Guo and Mike Frysinger and John Hauser and David Horner and Olof Johansson and David Kruckemyer and Yunsup Lee and Daniel Lustig and Andrew Lutomirski and Prashanth Mundkur and Jonathan Neuschäfer and Rishiyur Nikhil and Stefan O'Rear and Albert Ou and John Ousterhout and David Patterson and Dmitri Pavlov and Kade Phillips and Josh Scheid and Colin Schmidt and Michael Taylor and Wesley Terpstra and Matt Thomas and Tommy Thorn and Ray VanDeWalker and Megan Wachs and Steve Wallach and Andrew Waterman and Claire Wolf and Reinoud Zandijk. https://riscv.org/specifications/ratified/. Accessed 2025-01-08. 172 pages.

[113] RISC-V. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20240411, November 2024. Contributors: Derek Atkins and Arvind and Krste Asanović and Rimas Avižienis and Jacob Bachmeyer and Christopher F. Batten and Allen J. Baum and Abel Bernabeu and Alex Bradbury and Scott Beamer and Hans Boehm and Preston Briggs and Christopher Celio and Chuanhua Chang and David Chisnall and Paul Clayton and Palmer Dabbelt and L Peter Deutsch and Ken Dockser and Paul Donahue and Aaron Durbin and Roger Espasa and Greg Favor and Andy Glew and Shaked Flur and Stefan Freudenberger and Marc Gauthier and Andy Glew and Jan Gray and Gianluca Guida and Michael Hamburg and John Hauser and John Ingalls and David Horner and Bruce Hoult and Bill Huffman and Alexandre Joannou and Olof Johansson and Ben Keller

and David Kruckemyer and Tariq Kurd and Yunsup Lee and Paul Loewenstein and Daniel Lustig and Yatin Manerkar and Luc Maranget and Ben Marshall and Margaret Martonosi and Phil McCoy and Nathan Menhorn and Christoph Müllner and Joseph Myers and Vijayanand Nagarajan and Rishiyur Nikhil and Jonas Oberhauser and Stefan O'Rear and Markku-Juhani O. Saarinen and Albert Ou and John Ousterhout and Daniel Page and David Patterson and Christopher Pulte and Jose Renau and Josh Scheid and Colin Schmidt and Peter Sewell and Susmit Sarkar and Ved Shanbhogue and Brent Spinney and Brendan Sweeney and Michael Taylor and Wesley Terpstra and Matt Thomas and Tommy Thorn and Philipp Tomsich and Caroline Trippel and Ray VanDeWalker and Muralidaran Vijayaraghavan and Megan Wachs and Paul Wamsley and Andrew Waterman and Robert Watson and David Weaver and Derek Williams and Claire Wolf and Andrew Wright and Reinoud Zandijk and Sizhuo Zhang. https://riscv.org/specifications/ratified/. Accessed 2025-01-08. 670 pages.

[114] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL 2009*, January 2009.

[115] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*, 2012.

[116] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011.

[117] Susmit Sarkar, Peter Sewell, Luc Maranget, Shaked Flur, Christopher Pulte, Jon French, Ben Simner, Scott Owens, Pankaj Pawan, Francesco Zappa Nardelli, Sela Mador-Haim, Dominic Mulligan, Ohad Kammar, Jean Pichon-Pharabod, Gabriel Kerneis, Alasdair Armstrong, Thomas Bauereiss, and Jeehoon Kang. RMEM: Executable operational concurrency model exploration tool for ARMv8, RISC-V, Power, and x86, 2010–2023. [web interface].

[118] Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of Theoretical Aspects of Computer Software (Sendai), LNCS 2215*, pages 535–559, October 2001.

[119] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, 2013.

[120] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[121] Dennis E. Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[122] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A separation logic for fictional sequential consistency. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 736–761, 2015.

[123] Ben Simner, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, , and Peter Sewell. Relaxed exception semantics for Arm-A (extended version), 2024.

[124] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in Armv8-A. In *Proceedings of the 31st European Symposium on Programming*, volume 13240 of *Lecture Notes in Computer Science*, pages 143–173. Springer, April 2022.

[125] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. ARMv8-A system semantics: instruction fetch in relaxed architectures. In *Proceedings of the 29th European Symposium on Programming*, April 2020.

[126] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–42. Kluwer, 1991.

[127] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–41. Springer, 1992. Book editors: Michel Dubois and Shreekant Thakkar. Another version of this chapter appeared as Xerox TR CSL-91-11.

[128] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4-5):505–522, 2005.

[129] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, USA, 1992.

[130] Intel staff. Personal communication, November 2009.

[131] W. J. Starke, J. Stuecheli, D. M. Daly, J. S. Dodson, F. Auernhammer, P. M. Sagmeister, G. L. Guthrie, C. F. Marino, M. Siegel, and B. Blaner. The cache and memory subsystems of the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):3:1–3:13, 2015. Part of IBM Journal of Research and Development Issue 1 Jan.-Feb. 2015 *IBM POWER8 Technology*, https://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=7029148&punumber=5288520.

[132] Joel M. Tendler, J. Steve Dodson, J. S. Fields Jr., Hung Le, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[133] A. M. Turing. Proposals for Development in the Mathematics Division of an Automatic Computing Engine (ACE). Report E.882, The National Physical Laboratory. https://www.npl.co.uk/getattachment/about-us/History/Famous-faces/Alan-Turing/turing-proposal-Alan-LR.pdf?lang=en-GB. Accessed 2024-04-01.

[134] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[135] J. von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[136] Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proceedings of the 11th European Symposium on Programming (Grenoble), LNCS 2305*, pages 278–294, April 2002.

[137] M. V. Wilkes and W. Renwick. The EDSAC (electronic delay storage automatic calculator). *Math. Comp.*, 4:61–65, 1950. https://www.ams.org/journals/mcom/1950-04-030/S0025-5718-1950-0037589-7/. Accessed 2024-04-01.

[138] F. C. Williams and T. Kilburn. Electronic digital computers. *Nature*, 162:487, 1948. https://www.nature.com/articles/162487a0. Accessed 2024-04-01.

[139] M. Woodger. Automatic Computing Engine of the National Physical Laboratory. *Nature*, 167:270–271, 1951.