The Semantics of Multicopy Atomic ARMv8 and RISC-V

Christopher Pulte

This dissertation is submitted for the degree of Doctor of Philosophy

Department of Computer Science
University of Cambridge

Clare Hall, September 2018

für Oma Hildegard

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text.

It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text.

This dissertation contains fewer than *65,000* words including tables and footnotes, excluding appendices, bibliography, equations, and diagrams, and has fewer than 150 figures. The application for extending the word limit to 65,000 words has been approved by the Degree Committee.

Word count: 64,854.

Abstract

Previous work has established precise operational concurrency models for Power and ARMv8, in an abstract micro-architectural style based on detailed discussion with IBM and ARM staff and extensive hardware testing. To account for the precise architectural behaviour these models are complex. This thesis aims to provide a better understanding for the relaxed memory concurrency models of the architectures ARMv8, RISC-V, and (to a lesser degree) Power.

Power and early versions of ARMv8 have non-multicopy-atomic (non-MCA) concurrency models. This thesis provides abstraction results for these, including a more abstract non-MCA ARMv8 storage subsystem model, and characterisations of the behaviour of mixed-size Power and non-MCA ARMv8 programs when using barriers or release/acquire instructions for all memory accesses, with respect to notions of Sequential Consistency for mixed-size programs.

During the course of this PhD project, and partly due to our extended collaboration with ARM, ARM have shifted to a much simplified multicopy-atomic concurrency architecture that also includes a formal axiomatic concurrency model. We develop a correspondingly simplified operational model based on the previous non-MCA models, and, as the main result of this thesis, prove equivalence between the simplified operational and the reference axiomatic model.

We have also been actively involved in the RISC-V Memory Model Task Group. RISC-V has adopted a multicopy atomic model closely following that of ARMv8, but which incorporates some changes motivated by issues raised in our operational modelling of ARMv8. We develop an adapted RISC-V operational concurrency model that is now part of the official architecture documentation.

Finally, in order to give a simpler explanation of the MCA ARMv8 and RISC-V concurrency models for programmers, we develop an equivalent operational concurrency model in a different style. The Promising-ARM/RISC-V model, based on the C11 Promising model, gives up the micro-architectural intuition the other operational models offer in favour of providing a more abstract model. We prove it equivalent to the MCA ARMv8 and RISC-V axiomatic models in Coq.

Acknowledgements

I am very grateful to my supervisor, Peter Sewell, for all his support. Peter made time for me whenever I needed help, and provided patient support and valuable guidance, whether on the PhD work or anything else.

I would also like to thank my examiners, Derek Dreyer and Neel Krishnaswami, whose helpful feedback improved this thesis, and Andrew Pitts and the late Mike Gordon for their guidance in earlier stages of the PhD.

I thank my co-authors, especially Shaked Flur, Jean Pichon-Pharabod, Chung-Kil Hur, Jeehoon Kang, Sung-Hwan Lee, Will Deacon, Susmit Sarkar, and Luc Maranget for exciting and productive collaborations. I also thank Shaked for his tikz library that produces the litmus diagrams in the thesis. I would like to thank Alan Stern for interesting discussions. I enjoyed working in the Programming, Logic, and Semantics Group at the Computer Laboratory, which provided a stimulating research environment.

I am grateful for the funding provided by the Computer Laboratory's and Qualcomm's Premium Studentship, the EPSRC's and Arm Ltd.'s Industrial CASE Studentship (grant no. EP/L505389/1), and the EPSRC Programme Grant "REMS: Rigorous Engineering for Mainstream Systems" (grant no. EP/K008528/1). I would also like to thank the department administration and especially Lise Gough for their help.

I am very grateful to my parents and my family for their lifelong support, and to my girlfriend Nan jan for always being there and cheering me up.

Contents

Chapter 1

Introduction

Much of the literature on concurrency assumes Sequential Consistency (SC), which describes the concurrency behaviour as the possible sequentialised interleavings of the threads' commands. As a result, Sequentially Consistent concurrency is well-understood, and there exist different formalisations, reasoning techniques, and model checkers for it. In practice, however, widely used processor architectures such as ARMv8 and IBM Power, and programming languages such as C/C++ and Java, have so-called relaxed-memory concurrency models. Instead of the simple programming model offered by Sequential Consistency, relaxed memory models give much weaker guarantees about the concurrency behaviour: at the expense of a more complicated concurrency semantics, relaxed-memory concurrency models allow the effects of certain processor and compiler optimisations to become visible to the programmer, in order to allow for better performance and power efficiency, and easier implementability.

In the case of the Power and ARMv8 processor architectures, the allowed concurrency behaviour includes the behaviours resulting from typical processor optimisations: the hardware threads execute instructions out-of-order and speculatively, and instructions observably do not execute as atomic units; processors use store queues and caches to speed up memory accesses, resulting in the delayed propagation of writes to other threads. At the same time, however, the hardware must provide certain guarantees to the programmer, such as coherence, and ordering from memory barriers. The requirement of providing these guarantees while allowing for enough freedom for hardware implementers for optimisations leads to subtle concurrency semantics for Power and ARMv8.

Example relaxed behaviours To illustrate some of the concurrency behaviours the Power and ARM concurrency models are concerned with, consider the following examples. (A reader familiar with examples of relaxed memory behaviour may wish to skip this.)

Thread 0:	Thread 1:
store [x] 54	X0 := load [y]
store [y] 1	X2 := load [x]

Thread 0:	Thread 1:
MOV X0,#54	LDR X0,[X3]
STR X0,[X1]	LDR X2,[X1]
MOV X2,#1	
STR X2,[X3]	

Figure 1.1: MP test

Figure 1.1 shows the message passing (MP) litmus test with two threads, on the left in pseu-

docode, on the right in ARMv8 assembly: in this example, Thread 0 writes some data to x, here the value 54, and subsequently writes 1 to a flag y to signal to other threads that the data is available; Thread 1 reads y and subsequently x. The right-hand side implements this in ARMv8 assembly, assuming both threads' registers `X1` hold the memory address of x and `X3` that of y. Under a Sequentially Consistent concurrency model one can expect that if Thread 1 reads 1 for y, it must subsequently see the up-to-date value 54 for x. In Power and ARMv8 this is not the case: it is possible for Thread 1 to read $y = 1$ and subsequently read the (old) initial value $x = 0$. One reason this behaviour is allowed in Power and ARMv8 is that the two stores of Thread 0 are allowed to propagate their writes out of order: the store to y before the store to x, so that Thread 1 may read $y = 1$ and $x = 0$ before Thread 0's write to x propagates to Thread 1.

In order to allow focussing just on the concurrency aspects and ignoring the details of the assembly language when discussing example executions, we now introduce *execution graphs*.

Execution graphs specify executions of a given concurrent program in terms of events and relations over these events. Each execution graph represents a single concrete allowed or forbidden execution of the program. It contains the following data:

- The set of events of the particular control-flow unfolding of the execution. This will typically have a single write event for each executed store instruction instance and a single read event for each executed load instruction instance. (In the case of programs with misaligned memory accesses there might be multiple writes or reads for a single instruction instance.)
- The program-order relation (po) is a total order on the events from the same thread that corresponds to the control-flow unfolding of the execution.
- The reads-from relation (rf), relating a write w with a read r if r reads from w in this execution.
- The coherence relation (co), relating a write w with another write w' to the same address if w is sequenced before w' in memory.
- Moreover, the derived *from-reads* relation fr [7, 14] relates a read event to any write event that is later in the coherence order co than the write from which it read (defined as $\mathsf{rf}^{-1}; \mathsf{co}$, where semicolon ";" denotes sequential composition of relations).
- Dependency relations. These record certain dataflow or control flow dependencies between events, and typically relate a read event to another event whose execution depends on the return value of the read's load. (More on these later.)

Execution diagrams may also contain additional events or relations, such as for the barrier instructions of the execution, etc. We will later introduce those when they become relevant.

Axiomatic memory models specify an architecture's or programming language's concurrency semantics in terms of such execution graphs: given an input program

they specify the allowed behaviours by first enumerating the set of all possible such execution graphs, called *candidate executions*, and then ruling out graphs violating the axioms of the concurrency model. The axioms typically forbid cycles of certain shapes in the relations of the candidate execution.

For now, we focus just on the concurrency behaviours, no particular (operational or axiomatic) model, but we will later return to such axiomatic models in the context of the multicopy-atomic ARMv8 architecture in Section 6.4.1.

Returning to the example behaviours, Figure 1.2 shows the execution graph of the previously discussed relaxed behaviour in the MP test. Here, the events are a, b, c, and d; the writes a and b originate from Thread 0's stores, the reads c and d from Thread 1's loads. The po edges relate the events of both threads, a before b and c before d. (Later, the po edges will sometimes be omitted.) In this particular execution, c reads-from b, and d reads-from the initial write for x ($x = 0$). Since the initial write for x is coherence-ordered-before a, the read d is from-reads-related to a.

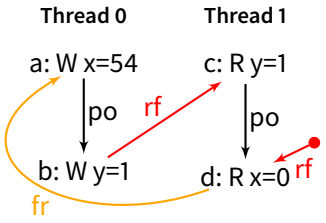


Figure 1.2: MP execution

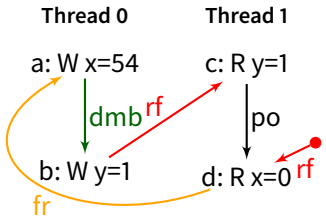


Figure 1.3: MP+dmb

The next test, in Figure 1.3, places a strong memory barrier `dmb sy` between the two writes on Thread 0 (`dmb sy`, sometimes abbreviated `dmb`, is a strong memory barrier in ARMv8, `sync` is the corresponding barrier in Power). In the diagram, the edge `dmb` indicates the barrier between a and b. The memory barrier prohibits propagating b before a, but the behaviour of the test remains allowed: instead of propagating the writes out of order Thread 1 can execute the reads out of order — d reading from the initial write to x before a and b propagate to Thread 1 and c reads from b. Placing a `dmb sy` barrier also between c and d would prevent the behaviour. The following paragraph gives an informal overview of the barriers available in ARMv8 that this thesis considers.

Barriers in ARMv8. The barrier `dmb sy` is a strong/full memory barrier. All loads and stores program-order-succeeding the barrier wait for all loads and stores program-order-preceding the barrier. ARMv8 also provides two similar but weaker barriers: `dmb st` is a store-barrier, ordering stores program-order-before and program-order-after the barrier; `dmb ld` is a load barrier, ordering loads program-order-before the barrier before loads and stores program-order-after it.

ARMv8 also has load acquire and store release instructions, so-called half barriers. A load acquire is a load with extra ordering: all loads and stores program-order-

succeeding it must wait for it. The store-release is the dual: it waits for all program-order-preceding loads and stores. Moreover, load acquire instructions wait for all program-order-preceding store release instructions. Since ARMv8.3 there is a weaker variant of load acquire (`LDAPR`), which does not wait for preceding store releases.

Apart from these, the thesis considers one other barrier, `ISB` (Instruction Synchronisation Barrier). This barrier is intended for use in the context of self-modifying code (which is not a topic of this thesis), but also provides ordering for "normal" memory accesses. An ISB orders any load l program-order-before the ISB before any load after it, if the return value of l affects the control flow preceding the ISB or the address of any memory access preceding the ISB. The precise semantics of each of these instructions is a topic of this thesis and will be explained in detail later.

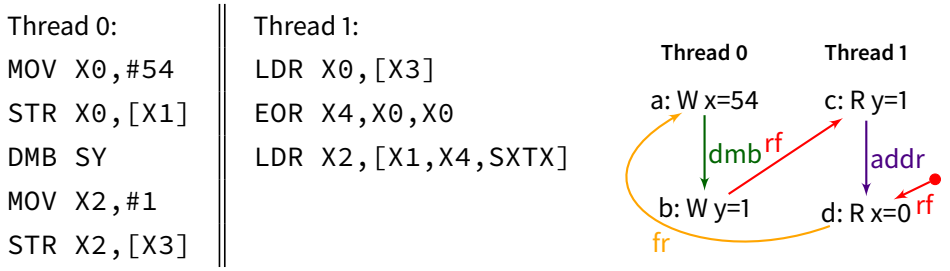


Figure 1.4: MP+dmb+addr

Figure 1.4 changes the program to make d *address-dependent* on c — the return value of c is used in computing the address of d: in the program, on the left of the figure, the second load uses the value of register `X4` to compute the memory address as an offset from the base address x; the value of `X4` is data-dependent on the value returned by the first load, since `EOR` (exclusive OR) uses `X0` in computing the value of `X4` (here this value of `X4` is always 0 due to the exclusive OR). The address dependency is indicated in the execution graph by the addr relation. Other possible dependencies are data dependencies (data), and control dependencies (ctrl). Control dependencies are dependencies from a read event to program-order-later events that follow a conditional branch or computed jump whose branch target depends on the value returned from the read. The dependency on Thread 1 enforces sequential execution of c and d, and, in conjunction with the strong barrier, forbids the behaviour. If in example Figure 1.4 the address dependency is replaced by a control dependency, however, the execution is allowed in Power and ARMv8 (diagram omitted). Both architectures allow the effects of *speculative execution* to be observable: they allow executing d before the read c determines the control flow — provided the speculation later turns out to have been correct — and thus allow the execution of the adapted example.

There are many variants of this basic message passing (MP) test shape, and many other such

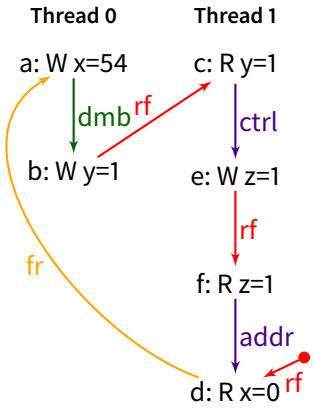


Figure 1.5: PPOCA

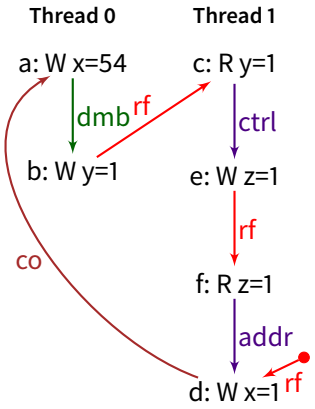


Figure 1.6: S+dmb+ctrl-rfi-co

tests that expose particular choices in an architecture's (or other language's) concurrency model. The following gives two more pairs of examples of such tests, concerning the ordering of events within the threads, to illustrate some more subtle case distinctions Power and ARM's concurrency models make. Figure 1.5 shows the *PPOCA* litmus test [110], which changes the previous test to replace Thread 1's address dependency by a sequence of dependencies: e is a write to a separate location z that control-depends on c, and d address-depends on the read f at location z, which is assumed to read from e. The control dependency of e on c prohibits propagating e into memory before satisfying c. One might think that the conditional branch depending on c will therefore hold back d from executing before c, due to the address dependency on f. However, in both Power and ARMv8, Thread 1 can speculatively explore the conditional branch of e, f, and d, determine the address and data of e, and make e available for *thread-local forwarding*: in micro-architecture, each core may keep a buffer of currently-speculative writes, for which it is not yet determined whether they may be committed into the storage subsystem and start becoming observable to other threads; the core may then allow loads to be satisfied speculatively by such writes with matching footprint in this buffer. Hence, f can be satisfied by forwarding and resolve d's address dependency, and d can read from the initial write for x before c is satisfied (again, provided the speculation later turns out to have been correct). The similar execution of Figure 1.6, where in place of the read d is a write of x that becomes coherence ordered before a, is forbidden, because in Power and ARM d is only allowed to propagate when all program-order-earlier control-flow is determined, and so only after c is satisfied.

Figure 1.7 shows the similar *RSW* test [110] where the reads c and d are separated by two other reads e and f of z, reading from a write g on another thread. The read of e cannot be satisfied until c is, due to the incoming address dependency. In both Power and ARMv8, however, the second read of z is allowed to be satisfied before e, thereby resolving d's address dependency and allowing d to satisfy early, thereby allowing the execution. If, however, as in the *RDW* test [110] of Figure 1.8 e and f read from different writes, e from the initial write and f from g, then the execution is forbidden, since then the concurrency models of Power and ARMv8 require e and

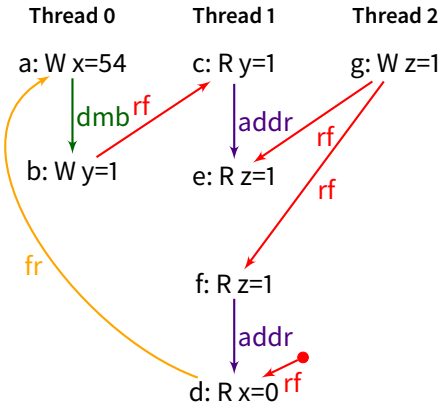


Figure 1.7: RSW

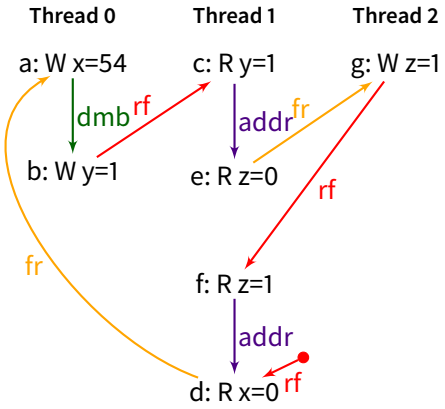


Figure 1.8: RDW

f to execute in program order.

While the Power and ARM concurrency models allow executing same-address reads out of order under certain circumstances (as in the RSW example), they are only allowed to execute in a way that preserves *coherence*: once a thread has "seen" a particular write, it must not subsequently read from a write coherence-older than this. For instance, the following example, called CoRR [110], is forbidden in Power and ARM. In this example Thread 0 writes 54 to x. On Thread 1

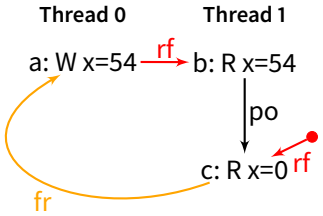


Figure 1.9: CoRR

the read b reads from this write, but the program-order-subsequent read c reads 0 from the initial value. The hardware may execute the two loads out of order, satisfying c before b, if b is being held back by register dependencies, for instance. When b is subsequently satisfied, by a coherence newer write (here a), the hardware will detect a coherence violation, and *restart* the load instruction of c, or reset it to an earlier state, to force it to be satisfied again and resolve the coherence violation.

In addition to ordering and coherence guarantees, Power and ARM also have instructions that provide a certain *atomicity guarantee*: load exclusive and store exclusive instructions (in ARMv8; their Power analogues are called load reserve and store conditional). Exclusive instructions work in pairs: a store exclusive is *paired* with a load exclusive if the load exclusive program-order-precedes it, and there is no other exclusive instruction between them. Typically the paired instructions are to the same memory address. The purpose of a load/store exclusive pair is that between executing the load exclusive and the store exclusive, the thread can acquire "exclusive access" to their location: a store exclusive can succeed or fail; the architectures guarantee, that a store

exclusive s paired with a preceding same-address load exclusive l can only succeed if its write is the immediate coherence successor of the write l read from. (Other writes by the thread of l and s are also allowed between.) This guarantee is used in implementing lock data structures. A store exclusive that is not paired must fail, and a store exclusive can always fail "for no reason".

To illustrate this, consider the ARMv8 program in Figure 1.10 and assume that on both threads `X0` holds some address x. The first instruction of Thread 0 is a load exclusive, reading x into register `W1`. Thread 0 then writes 1 to register `W2` and executes a store exclusive of 1 to x (`W2` holds the value to be written and `X0` the location, x). The store exclusive is paired with the load exclusive.

Now assume the load exclusive reads from the initial write $x = 0$. If after that the store exclusive executes, it can succeed and write 1 to x, since no other write to x has gone into memory after the initial write. If, however Thread 1 executed its write of 2 to x before that, the store exclusive would fail: it would not be allowed to succeed, because Thread 1's write would have overwritten the write read by the paired load exclusive. The store exclusive has an additional register argument, `W29`, the *success* or *status register* of the store exclusive. To this register the store exclusive writes a bit indicating whether it succeeded or failed. In the example, if there were other stores to x by Thread 0 between the load and the store exclusive, the store exclusive would still be allowed to succeed, provided there are no coherence-intervening writes *by other threads*.

Thread 0:	Thread 1:
`LDXR W1,[X0]`	`MOV W1,#2`
`MOV W2,#1`	`STR W1,[X0]`
`STXR W29,W2,[X0]`	

Figure 1.10

This thesis These examples show only few of the behaviours the Power and ARMv8 concurrency models are concerned with, but illustrate some of the subtleties of the semantics that originate from the architectures' goal to define a relaxed memory model that allows for hardware implementation freedom but also provides coherence, ordering guarantees due to barriers and dependencies, etc.

The goal of this thesis is to improve the understanding of the relaxed-memory concurrency models of ARMv8 and (to a lesser degree) Power. The work is based on the models of Sarkar et al. [110, 111], Gray et al. [60], and Flur et al. [51, 52] that specify the non-multicopy-atomic (non-MCA) concurrency behaviour of Power [110, 111, 60] and early versions of ARMv8 [51, 52]. (Non-multicopy-atomicity here means that writes may propagate to different threads at different points in time, as will be explained later.) These models are exhaustively executable operational concurrency models written in the specification language Lem [96], developed based on extensive testing of existing processor implementations and discussion with IBM and ARM architects. This thesis develops simplifications and proves abstraction results for the architecture models.

The aforementioned operational Power and ARMv8 models describe not only the architecturally

allowed concurrency behaviour but also the sequential aspects of the semantics for large parts of the user-mode instruction set architectures. For the latter, the models have instruction definitions written in the *Sail* specification language [60], that were initially integrated into these models using a definitional interpreter. The instruction semantics being defined in terms of an interpreter means studying the semantics of sequential and concurrent Power and ARMv8 programs requires reasoning about the behaviour of the interpreter. Moreover, the interpreter introduces run-time overhead when using the model as an executable simulator. Chapter 2 describes an alternative implementation of the instruction semantics using a shallow embedding of Sail, in order to simplify the architectural models and express the sequential aspects of the instruction semantics more directly and efficiently. This shallow embedding work is partly based on earlier work on a shallow embedding of Sail into OCaml by Kathy Gray.

The non-multicopy-atomicity of early versions of ARMv8 led to a particularly complicated concurrency model for ARMv8. The Flowing and POP models [51, 52] specify the subtle concurrency semantics in an operational model (written in Lem) that comprises a thread subsystem and storage subsystem, in which the thread subsystem manages the hardware threads' out-of-order and speculative execution, and the storage subsystem abstracts from multiple hierarchies of caches in the memory subsystem and manages the propagation of writes between threads and replying to read requests. The Flowing and POP storage subsystems handle the propagation of events — which is central in non-multicopy-atomic models — using explicit propagation transitions. As a result, in order to understand the architecturally allowed concurrency behaviour one has to think about the interaction of these with the rest of the thread and storage subsystem behaviour. Chapter 4 introduces the NOP storage subsystem model for the non-MCA ARMv8 architecture, simplifying POP by abstracting from the explicit event propagation (for memory accesses of the same size and alignment), formalised in Lem. NOP is proved to allow all behaviour allowed by POP and is experimentally equivalent on a large suite of litmus tests. The definition of POP has changed since the development of NOP. It should be possible to adapt NOP accordingly, and also to support load/store exclusive and mixed-size instructions, that it currently does not. Chapter 4 discusses the details of these caveats.

Unlike most previous work on concurrency models, Flur et al. [52] consider the concurrency behaviour of Power and ARMv8 where — as occurring in practical examples — memory accesses are allowed to be of different sizes ("mixed-size" memory accesses/programs). A standard result and a typical expectation for concurrency models is that inserting barriers between all pairs of memory accesses restores the simpler programming model of Sequential Consistency (at the expense of performance). An example, due to Shaked Flur [52], shows, this does not hold for Power and non-MCA ARMv8 when considering mixed-size programs. Chapter 5 proposes a notion of Mixed-Size Sequential Consistency that is weaker than the standard definition of Lamport [76] and proves that non-MCA ARMv8 and Power programs with barriers between all memory accesses do provide Mixed-Size Sequential Consistency, in addition to a guarantee about the atomicity of loads and stores. Moreover, non-MCA ARMv8 programs where all memory loads are Load Acquires and all stores are Store Releases do restore the original strong notion of Sequential Consistency.

During the course of this project, and partly due to the issues raised in our operational modelling

of the non-MCA ARMv8 architecture, principally the work by Shaked Flur, ARM have revised ARMv8 and shifted to a multicopy atomic concurrency (MCA) model — a model in which writes propagate to all threads except the writer thread at the same time — a big conceptual simplification. (In the following, the text will refer to the non-MCA ARMv8 architecture and the MCA ARMv8 architecture to distinguish.) Chapter 6 presents a much simplified operational model, called the *Flat* (ARMv8) model, for MCA ARMv8 based on Flowing [51, 52], enabled by this architectural change. The model was developed principally by the author and Shaked Flur, in joint work with Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell.

The revised ARMv8 architecture also has, for the first time for ARM, a formal concurrency specification as part of the official architecture documentation, specified as an axiomatic concurrency model: it is defined as a predicate over *candidate executions* of a given program, specifying the allowed behaviour of the program by ruling out illegal candidate executions, typically such in which certain model-specific relations contain a cycle. This model will henceforth be called ARMv8-axiomatic. The model can be used in the herd [17] framework.[1]

This axiomatic model is much more concise than the Flat operational model and expresses the concurrency semantics more abstractly. Since the axiomatic model is expressed as a predicate on candidate executions, it directly states global properties of the complete executions allowed by the model. Historically the axiomatic computation of allowed outcomes by herd has been considerably faster than the exhaustive memoised search in the previous operational models, which is especially combinatorially challenging for non-MCA models. The simplified operational model for the revised architecture, however, improves on this and makes the performance difference less drastic.

The operational model, on the other hand, is more complex. Some of this complexity is due to the operational model's goal of abstracting from micro-architectural implementations in order to gain confidence in the correctness of the model, and some due to handling more features of the architecture: while the axiomatic model includes an ISA model for only few memory and arithmetic instructions the operational model integrates the more extensive Sail ISA model of around 220 user-mode ARMv8 instructions, and handles mixed-size memory accesses. Whereas the axiomatic model directly gives global properties of full executions, the operational model computes the model-allowed behaviour incrementally. In addition to exhaustive enumeration, the operational model supports interactively exploring the behaviour of sequential and concurrent tests in a web user-interface and running tests pseudo-randomly for the purpose of debugging larger concurrent programs, in the *rmem* tool (formerly *ppcmem*) [110, 111, 60, 51, 52, 104].

As the main result presented in this text, Chapter 7 gives an equivalence (hand) proof between the Flat operational model for the revised architecture and the official ARMv8-axiomatic model, for finite executions of ARMv8 programs in the fragment covered by both models — roughly user-mode non-mixed-size ARMv8 programs, without ARMv8.3's weaker form of Load Acquire.

Chapter 8 describes work on a concurrency model for the RISC-V processor architecture: during the course of this project, the RISC-V community has started work on defining and formally specifying the memory model for RISC-V. We have been directly involved in the Memory Model

[1] The next two paragraphs follow Pulte et al. [104].

Task Group chaired by Daniel Lustig. RISC-V has decided on a memory model that closely follows that of ARMv8, but deviates from it to address issues raised by our operational modelling of ARMv8. The architecture documentation includes, alongside two presentations of an axiomatic specification, an adapted version of the operational model for MCA ARMv8. As of late September 2018, the proposed memory model has been ratified, together with the axiomatic and operational models as non-normative explanatory material. This is joint work with Shaked Flur, Peter Sewell, Luc Maranget, Susmit Sarkar, and the Memory Model Task Group.

Finally, Chapter 9 describes an alternative, more abstract operational concurrency model for MCA ARMv8 and RISC-V. In recent work on the concurrency semantics of the C/C++ programming language, Kang et al. [71] have developed a concurrency model called the Promising Semantics. This model executes instructions mostly in program order, except for stores: the model has a memory which retains the history of all writes propagated and explains the out-of-order execution of loads by allowing them to read from older writes in the message history. Write out-of-order execution is explained using a notion of *promises* — loosely, writes executed "early" — and the execution of loads and stores is constrained by *views*, capturing certain ordering requirements within the threads. This text describes joint work with Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur on an operational model for the revised ARMv8 architecture and RISC-V in the style of the Promising Semantics. This model gives up the closer relation to micro-architectural implementations offered by the previous models in order to achieve a simpler semantics from a programmer's view, by expressing it in terms of an operational model that emphasises the thread-local execution of instructions in program order. Moreover, this model can be made executable and achieves significantly better performance than the Flat operational and ARMv8-axiomatic models. The model is formalised separately in Lem and in Coq, and proved equivalent with the ARMv8 and RISC-V axiomatic models in Coq.

In short, the contributions of this thesis are:

- a more abstract specification of the instruction semantics, by a shallow embedding of Sail (Chapter 2);
- a more abstract storage subsystem model for the non-multicopy-atomic ARMv8 architecture, NOP, and a (hand) proof of soundness of NOP with respect to POP (Chapter 3);
- a proposal of a definition for Mixed-Size Sequential Consistency and
- a (hand) proof that "fully-barriered" Power and non-MCA ARMv8 programs have Mixed-Size Sequentially Consistent behaviour, as well as a guarantee about the atomicity of loads and stores (Chapter 5);
- a (hand) proof that non-multicopy-atomic ARMv8 programs that use Release/Acquire instructions for all memory accesses have Sequentially Consistent behaviour (Chapter 5);
- a simplified operational concurrency model for the revised multicopy atomic ARMv8 architecture, based on Flowing (Chapter 6);
- a (hand) proof of equivalence of this operational model with the official ARMv8-axiomatic model for finite executions and the features covered by both models (Chapter 7);
- a discussion of our work in the RISC-V Memory Model Task Group, and an operational concurrency model for RISC-V based on that for MCA ARMv8 (Chapter 8); and

- a simpler alternative operational concurrency model for multicopy atomic ARMv8 and RISC-V based on the C/C++ Promising semantics, with proof of equivalence with the ARMv8 and RISC-V axiomatic models, mechanised in Coq (Chapter 9).

Collaboration and publications This thesis presents work in collaboration with many people. The following list details the authorship, and the papers in which the results were published.

Chapter 2 The first part of this chapter provides background on Sail and an overview of existing Sail models, not the author's work. The Sail shallow embedding is the author's work, partly based on Kathy Gray's early work on a shallow embedding of Sail into OCaml. The generation of instruction tests is the author's work, but relies on Kathy Gray's Sail implementation and interface. The shallow embedding has been used in Pulte et al. [104] and the test generation described in Gray et al. [60] for Power, and in Flur et al. [51] for ARMv8.

Chapter 3 This chapter only provides background on the non-multicopy-atomic concurrency models of Power and early ARMv8.

Chapter 4 The NOP model and soundness proof are the author's work.

Chapter 5 The observation that fully-barriered mixed-size Power and ARMv8 programs do not have SC behaviour is due to Shaked Flur, the characterisations of mixed-size Power and non-MCA ARMv8 programs with barriers or release/acquire instructions and the proofs are the author's work. The results were published in Flur et al. [52].

Chapter 6 The simplified Flat operational model and ARMv8-axiomatic model were co-developed. The Flat operational model was developed principally by the author and Shaked Flur, in joint work with Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. It is directly based on the previous work on non-MCA ARMv8 concurrency [51, 52], in turn based on that for Power [110, 111, 60]; the ARMv8-axiomatic model is principally due to Will Deacon. The contents of the chapter was published in Pulte et al. [104]

Chapter 7 The proof of equivalence between the ARMv8-axiomatic model and the Flat operational model is the author's work, incorporating corrections by Susmit Sarkar. The proof was published in the supplementary material of Pulte et al. [104].

Chapter 8 The work on RISC-V concurrency is joint work with Shaked Flur, Peter Sewell, Luc Maranget, Susmit Sarkar, and the Memory Model Task Group, chaired by Daniel Lustig. The adapted Flat RISC-V operational model is principally due to Shaked Flur and the author. The text description of the Flat RISC-V model was published as Appendix B.3 of the RISC-V ISA manual [121].

Chapter 9 The work on the Promising-ARM/RISC-V model, based on the C11 Promising model [71], is joint work with Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur, with the author as first author. The contents of this chapter is mostly taken from a joint paper, currently under submission [103]. The Coq formalisation is due to Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. (The chapter's combined ARMv8 and RISC-V axiomatic model is merely a simplification and combination of the already existing ARMv8 and RISC-V axiomatic concurrency models.)

The definitions of the concurrency models given in this thesis are the prose versions of formal

definitions written in the Lem specification language. (The Promising-ARM/RISC-V description follows the Coq development more closely.) The formal models, operational specifications of the concurrency behaviour, integrated with the Sail ISA models into the executable rmem tool, are available at https://github.com/rems-project/rmem. Just as the papers mentioned above, these models and the tool are the result of the collaboration of many people, as detailed on the same page, and are based on the ppcmem/rmem tools and models from prior work on the Power and non-MCA ARM concurrency models of Sarkar et al. [110, 111], Gray et al. [60], and Flur et al. [51, 52].

Throughout the thesis, the text will illustrate certain aspects of the concurrency behaviour with example litmus tests. The reader is encouraged to try these in the rmem web interface for the Power model, the ARMv8 Flowing, POP, and Flat models, the RISC-V Flat model, and the Promising ARM and RISC-V models, all of which are available at https://www.cl.cam.ac.uk/~pes20/rmem.

Chapter 2

Instruction semantics

The operational Power and ARMv8 architecture models [51, 52, 60, 110, 31] this text is concerned with comprise combinations of concurrency models and models for the instruction set architecture (ISA). The concurrency models are defined in higher-order logic in the *Lem* specification language [96] that generates OCaml code for execution and theorem prover definitions in the HOL4 and Isabelle theorem provers. The ISA models are defined in *Sail*, a special purpose instruction description language [60]. The *rmem* tool integrates the concurrency and ISA models into an executable tool that allows for the interactive, pseudorandom, and exhaustive exploration of sequential and concurrent test programs in the form of either litmus tests or Linux ELF binaries, the latter via the ELF front-end of Kell et al. [72]. Prior to this work, Sail's dynamic semantics was implemented as an interpreter defined in Lem. The following sections introduce Sail and the Sail interpreter, and give an overview of the Sail ISA models integrated into rmem. This thesis introduces a new shallow embedding of Sail into Lem, and the appendix describes a method for the semi-automatic generation of instruction tests to validate the instruction semantics. The shallow embedding has been used in Pulte et al. [104] and the test generation described in Gray et al. [60] for POWER, and in Flur et al. [51] for ARMv8.

2.1 Background: Introduction and interpreter

Sail is a domain-specific language for formally describing the sequential behaviour of instructions, in a similar style as the pseudocode descriptions found in the architecture manuals of Power and ARM [2, 22]. To this end, Sail is a first-order imperative language that supports the programming language features used in the pseudocode descriptions, including user-defined algebraic types, union types, record types, register type definitions, and recursive (potentially mutually recursive) function definitions, pattern matching, local and global state and exceptions, and computation on undefined bits. Moreover, since much of the architecture specification is concerned with the manipulations of bit vectors and calculations on them, Sail has a lightweight dependent type system with type inference that supports bit-vector length and integer range typing. Additionally, Sail has a simple effect system that tracks information about a Sail definitions effects, such as whether the definition is pure, reads or writes registers, reads or writes memory, or produces memory barrier requests. Figure 2.1 shows an example of instruction decode and execute behaviour specified in Sail.

The figure shows the definition of the instruction class AddSubCarry from a ARMv8 Sail specification, hand-written by Shaked Flur in correspondence with the ARMv8 Architecture Reference Manual. The upper part defines the instruction decode function, taking as an argument a bit vector and producing a value of type option<(ast<'R,'D>)>, where 'R and 'D are constrained to be certain

```
function forall Nat 'R, 'R IN {32,64}, Nat 'D, 'D IN {8,16,32,64}. option<(ast<'R,'D>)> effect pure
    decodeAddSubtractWithCarry ([sf]:[op]:[S]:0b11010000:Rm:0b000000:Rn:Rd) = {
  (reg_index) d := UInt_reg(Rd);
  (reg_index) n := UInt_reg(Rn);
  (reg_index) m := UInt_reg(Rm);
  ([:'R:]) datasize := if sf == 1 then 64 else 32;
  (boolean) sub_op := (op == 1);
  (boolean) setflags := (S == 1);
  Some (AddSubCarry(d,n,m,datasize,sub_op,setflags));
}

function clause execute(AddSubCarry(d,n,m,datasize,sub_op,setflags)) = {
  (bit['R]) operand1 := rX(n);
  (bit['R]) operand2 := rX(m);
  if sub_op then operand2 := NOT(operand2);
  let (result,nzcv) = AddWithCarry(operand1,operand2,PSTATE_C) in
  if setflags then wPSTATE_NZCV( ) := nzcv;
  wX(d) := result;
}
```

Figure 2.1: Example Sail pseudocode for AddSubCarry in ARMv8

powers of 2. The annotation effect pure enforces that the decode function has no side effects. The function header pattern matches on the argument bit vector, where ":" is vector concatenation.

The second part of the figure shows the execute behaviour for the same instruction. Here the := operator is assignment, rX(n) and wX(n), respectively, are functions for reading and writing the *n*th general purpose register, PSTATE_C is a register read of the special purpose register of the same name, and wPSTATE_NZCV( ) is a write to the NZCV special purpose register. (For presentation the syntax for Sail let expressions is adapted to drop curly braces around the expression in which the name is bound.)

In the sequential case the whole-system semantics can be expressed directly in Sail, by an implementation of the instruction semantics operating on a standard register and memory state. In the concurrent case the instruction behaviour and the register and memory semantics become more complicated and the Sail implementation needs to be combined with a memory model that separately handles the concurrency behaviour. Accurately modelling the concurrency behaviour imposes some requirements on the Sail implementation.

Firstly, the concurrency behaviour of Power and ARM makes visible that instructions do not execute in program order or as atomic steps. Instead, there is observable out-of-order execution, and program-order-later instructions that are executed early can observe how far program-order-earlier instructions have progressed. For instance, propagating a write into memory does not in

general require all program-order-earlier instructions to be done, but it is only possible when the addresses of all program-order-earlier memory accesses are known (to avoid coherence violations). To support modelling these concurrency aspects in the behaviour of instructions correctly, the Sail implementation has to allow stepping through instructions at a smaller granularity than full instructions, and communicate information relevant to the concurrency model as soon as it becomes available.

Secondly, the concurrency model requires information about the *footprint* of an instruction: information about the register and memory accesses and conditional branching the instruction will do. This is necessary to allow the concurrency model to determine which instruction actions are allowed to happen out of program order and which have to be kept in order. For instance, one instruction's read of a certain register may be done only when the last program-order-preceding instruction instances that write the relevant parts of this register have done these register writes, so in this case it is necessary to know the register write footprint of the program-order-earlier instruction instances.

In order to meet the first requirement, the interaction between the concurrency model and Sail interpreter was originally captured by the following definition of the outcome type: (This and the following text omits some values concerned with certain reads and writes of tagged memory that do not occur in Power, ARMv8, and RISC-V, and certain failure values.)

```
type outcome =
  | Read_mem of read_kind * address * size * (mem_value → outome)
  | Excl_res of bool → outcome
  | Write_ea of write_kind * address * size * outcome
  | Write_memv of memory_value * (bool → outcome)
  | Barrier of barrier_kind * outcome
  | Read_reg of reg_name * (reg_value → outcome)
  | Write_reg of reg_name * reg_value * outcome
  | Internal of outcome
  | Footprint of outcome
  | Done
```

Each step in the execution of the instruction's pseudocode generates a new outcome. An outcome is a request that is handed to the memory model by the instruction semantics, to inform it about progress in the instruction and in some cases to request information necessary to continue the instruction execution, such as in the case of a register or memory read. The different possible outcome values have the following meaning:

Read_mem(*read_kind*,*address*,*size*,*read_cont*)

This is a request to read memory, of *size* bytes at *address* (a bit vector of length 64). The argument *read_kind* identifies whether the read is from a (plain) load, or from a load with special memory semantics; the possible values are Read_plain, Read_acquire, Read_exclusive, or Read_exclusive_acquire. (In the case of RISC-V acquire loads come in a weaker RCpc and stronger RCsc variant.) The argument *read_cont* is a pseudocode continuation describing

the remaining instruction semantics behaviour: given a memory value (a byte list of the correct length) returned for this read request by the memory model it produces a new pseudocode state.

Excl_res(*res_cont*)

This is used only by ARMv8's store exclusive instructions and is a request to the memory model to determine whether the store exclusive should succeed or fail; *res_cont* is a pseudocode continuation that, given a boolean value indicating success or failure, returns the next pseudocode state.

Write_ea(*write_kind*,*address*,*size*,*next_state*)

This announces a memory store's address *address* and size *size*. Similar to *read_kind* in the previous case, *write_kind* identifies the write as coming from a plain store, or a store with special semantics, where the possible values are: Write_plain; Write_conditional for Power's store conditional instruction; Write_release, Write_exclusive, and Write_exclusive_release for ARMv8 and release and exclusive stores; Write_release, Write_conditional, and Write_conditional_release for RISC-V. (Similarly to the case of loads, RISC-V release stores come in the weak RCpc and strong RCsc variant.) The argument *next_state* is the next pseudocode execution state.

Write_memv(*mem_value*,*write_cont*)

After having announced the address and size of the memory access a store generates a request to write to memory with the value *mem_value*. The argument *write_cont* describes the pseudocode state reached when given a boolean argument for success or failure or the store exclusive. (ARMv8 store exclusives determine success early, using the Excl_res request, in Power and RISC-V the success is determined at the point of the Write_memv request.)

Barrier(*barrier_kind*,*next_state*)

This is a barrier request. Different barriers generate different values for *barrier_kind*:

- for Power Barrier_Sync for `sync`, Barrier_Lwsync for `lwsync`, Barrier_Eieio for `eieio`, and Barrier_Isync for `isync`;
- for ARM Barrier_dmb for `dmb sy`, Barrier_dmb_st for `dmb st`, Barrier_dmb_ld for `dmb ld`, Barrier_dsb Barrier_dsb_st, Barrier_dsb_ld for `dsb`, and Barrier_isb for `isb`; and
- for RISC-V Barrier_i for `fence.i`, Barrier_tso for `fence.tso`, and Barrier_*before*_*after* for every combination of *before*, *after* ∈ {r, w, rw}. The RISC-V `fence` instruction is parametric in the ordering it creates with respect to program order earlier (*before*) and program-order-later (*after*) memory accesses: reads r, writes w, or both rw. The argument *next_state* is the pseudocode execution state following the request.

Read_reg(*reg_name*,*read_cont*)

This is a request to read register *reg_name*, and *read_cont* is a continuation (a pure function) that returns the pseudocode state reached when given the register value for *reg_name* in the form of a vector of (possibly undefined) bits.

Write_reg(*reg_name*,*reg_value*,*next_state*)

This is a write of *reg_value* to *reg_name*, and *next_state* is the next pseudocode execu-

tion state.

Internal(*next_state*)

This outcome indicates an internal computation step in the execution of the pseudocode, such as arithmetic or calling auxiliary functions.

Footprint(*next_state*)

This is a request for dynamically recalculating the instruction instance's dependency footprint, used only in Power. Here *next_state* is the pseudocode state reached after the request. The analysis of instruction footprints and its recalculation will be explained in the following.

Done

This marks the end of the pseudocode execution of the instruction instance.

The outcome type supports the interaction between memory model and the instruction semantics, and executing instructions at the right granularity to explain the concurrency behaviour observable on Power and ARM to meet the first requirement from above.

For the second requirement, the Sail implementation needs to be able to provide a footprint analysis for an instruction instance. The information required by the concurrency model for any instruction instance is the following:

- the instruction kind: memory load and its load kind, memory store and its store kind, branch instruction, etc.;
- the register input footprint: which registers the instruction instance will read from, including the specific bit ranges;
- the register output footprint: which registers, and which bit ranges thereof, the instruction instance will write;
- the subset of the register input footprint that affects the memory address of any memory accesses the instruction may do; and
- the possible values of the program-counter (PC, named `CIA` and `NIA` in Power, `_PC` in ARMv8, and `PC` and `nextPC` in RISC-V) after the execution of this instruction instance; this is necessary for the correct handling of branch instructions that might set the PC to the value loaded from a register or an absolute address.

Prior to this work, Sail's dynamic semantics was implemented as a small step interpreter written in Lem — as a function that, given an environment and a Sail expression, evaluates this expression to a value of the outcome type given above. In addition to the normal mode of execution the interpreter provides a mode that allows deriving the footprint analysis for an instruction instance from the Sail definition of the instruction. This mode runs the instruction definition exhaustively and collects the set of the read and write requests to registers or memory this instruction generates, and tracks the dataflow within the instruction. The set of requests generated by the instruction is necessary to derive the instruction kind, the register footprint of the instruction, and the next branch targets; the dataflow analysis provides the information necessary to determine which register reads affect the instruction's memory accesses. When the instruction semantics produces a request to read from a register or memory location (or another kind of input) the instruction semantics requires a return value to feed into the pseudocode continuation (see outcome type

```
function clause execute (Lswx (RT, RA, RB)) = {
  ...
  ([|128|]) n_top := XER[57 .. 63];
  ...
  n_r := n_top quot 4;
  n_mod := n_top mod 4;
  n_r := if n_mod == 0 then n_r else n_r + 1;
  foreach (n from n_r to 1 by 1 in dec) {
    r := ([|32|]) (r + 1) mod 32;
    ...
    GPR[r] := temp
  }
}
```

Figure 2.2: A code fragment of the execute behaviour of Power's `lswx` instruction. The number of registers written to depends on the value n_r, derived from the value from the special purpose register XER

above). When analysing the footprint the interpreter feeds a distinguished *Unknown* value of the correct type into the continuation if the value is not already determined from the model state. When the control flow within the instruction definition depends on an unknown value the interpreter explores all possible paths, ensuring the footprint information returned for the possible paths is the same.

For most instructions in the subsets of the Power, ARMv8, and RISC-V architectures handled by the current Sail models, the footprint of an instruction instance is static: given a concrete instruction instance (that fixes the values for the instruction fields), the instruction instance's footprint is independent of which concrete value will be fed into the continuations generated by the outcome requests of the instruction. The Power architecture, however, has the instruction `lswx` ("Load String Word Indexed") where which registers will be accessed by the instruction depends on the value returned from one of `lswx`'s register reads (see Figure 2.2; the case of `stswx` is similar). However, if the architecture intends to allow the maximally liberal concurrency behaviour for this instruction, the concurrency model needs the precise register output footprint in order to decide at which point during the execution of an `lswx` a program-order-succeeding instruction instance's register read is allowed, whether it has to wait for the `lswx` instance to write to it.

For this reason, Sail has the Footprint outcome for the case of instruction instances with dynamic footprint: for `lswx` the instruction analysis first provides an over-approximation of the potential register output footprint of `lswx`, and the pseudocode description of `lswx` is adapted by hand to include a call to the function recalculate_footprint once the register read of XER is available, which generates the Footprint outcome; the concurrency model, when given the Footprint outcome, calls

the Sail interpreter to re-analyse the instruction footprint in the new context (where the XER value is available) to obtain the precise register output footprint information.

In the future, if the concurrency models and ISA models are extended to cover more aspects of the instruction behaviour, such as supervisor level code, it is likely that other instructions will have a more dynamic footprint, too.

2.2 Sail models

To date, there exist seven different ISA specifications written in Sail: Power, two ARMv8 models, x86, MIPS and CHERI-MIPS, and RISC-V, of varying sizes and feature coverage. Since this thesis is concerned with the concurrency models of ARMv8, Power, and RISC-V, for the purposes of this text only their ISA models are important. Recently, Sail has been revised to make changes to the type system and the Sail implementation. Some of the models below are for the original Sail (Sail 1 for emphasis), some for the updated Sail 2. An additional feature of Sail 2, mostly implemented by Alasdair Armstrong, is a C back-end for Sail, translating Sail definitions to C programs for the sake of emulation.

Power The Power Sail 1 ISA model described by Gray et al. [60] covers only user-mode instructions. Of those user-mode instructions it covers the instructions in the Branch Facility and Fixed-Point Facility of the Power ISA User Instruction Set Architecture of 154 instructions and four memory barrier instructions (this is counting different variants of instructions, such as `add`, `add.`, `addo`, and `addo` as one instruction); this excludes system trap instructions, vector and floating-point instructions, and non-user-level instructions [60]. The Power Sail ISA model is produced by an automatic extraction process: an XML version of the Framemaker sources used to produce the PDF version of the Power reference manual is parsed and the pseudocode information extracted and translated to Sail, with a number of patches applied [60]. The authors of the Power Sail model are Kathy Gray, Gabriel Kerneis, Susmit Sarkar, Peter Sewell, and the author. The Power ISA model has around 4600 lines of Sail code (around 6000 when including the hand-written functions computing the footprint information) and is validated using the single-instruction tests described in appendix Chapter A.

ARMv8 There are two Sail ARMv8 models. The first one for Sail 1, of Flur et al. [51], covers all non-floating-point, non-vector instructions from the application-level ISA. These include 224 instructions (counting subsections of Chapter C6 A64 Base Instruction Descriptions), except 21 instructions (and aspects of four other instructions) that handle exceptions, debug, and system-mode features, and excluding the load-non-temporal-pair instruction [51]. In contrast to the Power Sail model, the ARMv8 Sail model was hand-written by Shaked Flur in direct correspondence to the ARM Architecture Reference Manual [51]. The specification has around 5400 lines of Sail code (around 6000 including footprint information). This ARMv8 model is validated by experimental testing using the single-instruction tests of appendix Chapter A.

Whereas the former (hand-written) ARMv8 model only covers application-level aspects of the ISA, a more recent second ARMv8 ISA model for Sail 2 covers all of the 64-bit instructions of the

ARMv8.3 architecture, in around 23000 lines of code [24]. This model is automatically produced by translating from ARM's publicly released specification written in ARM's internal instruction description language ASL [105, 106, 107] into Sail. A separate translation from ARM's non-public internal specification additionally provides full specifications for all system registers, leading to an overall specification size of 30000 lines of code, after removing vector instructions (to reduce the code size). The ASL-to-Sail translation was implemented principally by Alasdair Armstrong and Alastair Reid; Thomas Bauereiss, Brian Campbell, and Ian Stark have worked on translating this from Sail to theorem prover definitions. The specification is experimentally validated against ARM's non-public Architecture Validation Suite, and by booting Linux in an executable model generated from the Sail model. In contrast to the hand-written ARMv8 specification the second is not yet integrated with the concurrency model. Both translations have focused on the 64-bit architecture and currently ignore ARMv8's 32-bit instructions.

MIPS and CHERI-MIPS The primary purpose of the MIPS model is to specify the CHERI extension to MIPS [123], a research architecture that adds hardware capabilities to MIPS. (Capabilities are a security mechanism that can be used for purposes of isolation and memory protection [24].) The MIPS Sail 2 model covers most of the MIPS64 architecture, in big endian mode only, and excluding some extensions such as floating point. The CHERI/MIPS model additionally covers the full CHERI extension. The MIPS and CHERI-MIPS models were written by Robert Norton-Wright and have around 2000 and 4000 lines of code, respectively. The models are predated by a CHERI-MIPS model in the L3 specification language L3 [53] (also [54, 55]). The CHERI model has been validated primarily using the CHERI test suite and by booting the MIPS version of FreeBSD. Earlier versions of the model were integrated with the ARM concurrency models in rmem, by Robert Norton-Wright and Shaked Flur.

RISC-V The RISC-V Sail 2 model covers "the 64-bit (RV64) version of the ISA: the rv64imac dialect (integer, multiply-divide, atomic, and compressed instructions), with user, machine, and supervisor modes, and the Sv39 address translation mode" [24]. The model has around 5000 lines of code and is integrated with the rmem RISC-V concurrency model. In the cases of Power and ARM the Sail models were closely related to a pseudocode description found in the vendor architecture manuals. RISC-V currently has no such pseudocode descriptions in the manual, and so the Sail model was hand-written, by Prashanth Mundkur, Robert Norton, and Shaked Flur, based only on the textual description in the architecture documentation. The model is validated primarily by comparing the Sail-generated OCaml executable model against the RISC-V Spike reference simulator, and by booting Linux and seL4.

2.3 Shallow embedding

The Sail interpreter provides a Sail implementation meeting the requirements described above, and, integrated into rmem, experimentally describes the correct sequential and concurrent instruction behaviour. However, the definition in terms of an interpreter has downsides for some applications: it complicates reasoning about the instruction behaviour and causes a performance

overhead at runtime. These two issues, primarily the issue in making rmem and the integrated ISA models useful for reasoning and formal proof, motivate simplifying the instruction semantics with a shallow embedding of Sail into Lem. This is partly based on Kathy Gray's early work on a shallow embedding of Sail into OCaml.

The shallow embedding maps Sail definitions to equivalent Lem definitions. Sail, however, has a number of features that cannot directly be represented in Lem. Most importantly, the translation from Sail to Lem has to cope with mapping Sail programs with imperative features such as local mutable variables and global register and memory state into pure function definitions in Lem. To handle this, the translation works in two phases: first transforming the Sail definitions into definitions in a subset of Sail that only uses the features that can be expressed directly in Lem; then pretty-printing the Sail definitions into their Lem counterparts. The Sail-to-Sail transformations necessary for this are as follows.

Remove vector-concatenation patterns Sail allows for various kinds of pattern-matching, including vector-concatenation pattern-matching, that Lem does not support. Since the Lem shallow embedding represents Sail bit vectors as lists of bits (untyped in the list length), this transformation replaces bit-vector concatenation patterns with list patterns that can be directly implemented in Lem. The steps for this are as follows, illustrated using the pattern

```
decodeAddSubtractWithCarry ([sf]:[op]:[S]:0b11010000:Rm:(0b000000:Rn:Rd))) = ...
```

from the previous definition of decodeAddSubtractWithCarry as an example (with some nesting added for presentation).

The first step is to introduce new names for all vector-concatenation patterns and their argument patterns that do not already have a name. Here the new names are v, v1, w, w1.

```
decodeAddSubtractWithCarry
  (([sf]:[op]:[S]:(0b11010000 as v1):Rm: (((0b000000 as w1):Rn:Rd) as w)) as v) = ...
```

Then the expression is rewritten to introduce let-bindings for the named slices of the given vector, dropping names that are not referred to in the rest of the program, and removing the explicit naming syntax ("as…") except the top-most one, v here.

```
decodeAddSubtractWithCarry
  (([sf]:[op]:[S]:0b11010000:Rm:(0b000000:Rn:Rd)) as v) =
  let Rm = v[11..15] in
  let w = v[16..31] in
  let Rn = w[6..10] in
  let Rd = w[11..15] in ...
```

In the general case, and unlike in this example, the original pattern might have nested vector-concatenation patterns. At this point these will be removed by recursively applying the vector-concatenation removal transformation, and the resulting pattern can be flattened and translated to a list pattern:

```
decodeAddSubtractWithCarry
```

```
  ([sf;op;S;1;1;0;1;0;0;0;0;_;_;_;_;_;0;0;0;0;0;0;_;_;_;_;_;_;_;_;_;_;_] as v) =
  let Rm = v[11..15] in
  let w = v[16..31] in
  let Rn = w[6..10] in
  let Rd = w[11..15] in ...
```

Handle local effects Sail has mutable local variables, to support the style of specifications found in the Power and ARM reference manuals. There are two obvious possible replacements for local mutable variables when targeting Lem: handle local state in a monad or transform mutable variable updates to let-bindings. The Sail-to-Lem translation chooses the latter, for two reasons: firstly, a monadic handling of local variables would require some form of polymorphic state that would be difficult to achieve in the non-dependently typed Lem language; secondly, treating local variables "more functionally" seems preferable for reasoning about the instruction semantics in theorem provers.

This means expressions of the form x := v; e will be transformed into expressions without local state updates of the form let x = v in e'. Expressions that affect the control-flow, such as if-expressions, case-expressions, and for-loops, need special care: these expressions have to be rewritten to return — in addition to any other values they might return — the values of any local variables updated inside the expression, without violating the scoping. Consider for instance the example expression below.

```
foreach (i from 0 to 31 by 1 in inc) {
  (bit[32]) foo := e;
  ...
  m := e';
  n := e'';
};
cmd
```

Here foo is local to the body of the for-loop but m and n are the local mutable variables declared in the surrounding code that are be mutated in the loop body. This will be translated to an expression of the shape below.

```
let [m;n] =
  forInc (0,1,31) [m;n] (fun i [m;n] →
    let (bit[32]) foo = e in
    ...
    let m = e' in
    let n = e'' in
    [m;n]
  ) in
cmd
```

Here foreach is a (higher-order) function that takes as arguments the current loop index — here initially 0 — the loop increment 1 and upper loop bound 31, a list of the values of variables mutated in the loop (their value before executing the loop body) — here *m* and *n* — and the body of the loop. The body of the for loop is represented as a function that takes as input the current loop index and the list of local variables mutated inside the loop and returns an updated variable list. The foreach function then recursively unrolls the loop according to the loop bounds and returns the list of values of the updated variables to the surrounding expression. Since foo is local to the loop body this must not be included in the list returned to the surrounding expression.

After this Sail-to-Sail transformation, any use of "local effects" in the form of local variable updates and reads has been replaced by purely functional let-bindings. The output code only uses "global effects", such as memory and register state accesses and memory barriers, that will be removed by the next transformation.

Handle global effects The goal of this Sail-to-Sail rewrite pass is to transform code with global effects into code that can be mapped to purely functional code that handles the effects with a monad. Sail does not make a distinction between imperative commands and pure expressions: effectful terms can legally occur in most places where general expressions are allowed. In order to bring the definitions into a form where they can be embedded into monadic code, this transformation separates effectful expressions from pure ones. For each effectful term the rewrite pass introduces a monadic let-binding that will eventually be mapped to the monad bind operator.

The implementation for this rewrite pass is based on an algorithm for transforming programs into A-normal form by Might [95], which extends an algorithm presented in Flanagan et al. [50] to handle side-effecting expressions. For the purpose of the Sail-to-Lem translation this algorithm can be adapted to Sail's grammar, and to introduce let-bindings only for effectful terms instead of all complex expressions: whereas the original algorithm transforms all expressions into a series of let-bindings of atomic expressions, here it is only necessary to let-bind any effectful subexpressions.

This transformation is mostly based on the effect types that Sail's type system annotates the internal AST representation of Sail expressions with. In addition to introducing monadic let-bindings, this rewrite pass also introduces the monadic return operation into pure subexpressions if by the expression's type an effectful term is expected, such as in the case where one side of an if-expression is effectful and the other pure.

```
let n = if b then rX(10) else 0 in e
```

is transformed into

```
letM n = if b then rX(10) else return 0 in e
```

where letM is pseudo-syntax for the monadic let-binding that is used only internally in the Sail implementation.

The three transformations in conjunction translate the Sail code into a subset of Sail that can be mapped directly to Lem. For instance, the above Sail code for the execute definition of the `AddSubCarry` instruction class is translated to the following Sail code:

```
function clause execute(AddSubCarry(d,n,m,datasize,sub_op,setflags)) = {
  letM (bit['R]) operand1 = rX(n) in
  letM (bit['R]) operand2 = rX(m) in
  let operand2 = if sub_op then NOT(operand2) else operand2 in
  letM w__0 = PSTATE_C in
  let (result,nzcv) = AddWithCarry(operand1,operand2,w__0) in
  letM ( ) = if setflags then wPSTATE_NZCV( ) := nzcv else return ( ) in
  wX(d) := result
}
```

This is then mapped into the following Lem code (dropping some type casts and typing arguments for simplicity), where >>= is the monad's bind operator and >> is defined by m >> n = m >>= fun ( ) → n:

```
let execute_AddSubCarry (d, n, m, datasize, sub_op, setflags) =
 rX n >>= fun operand1 →
 rX m >>= fun operand2 →
 let operand2 = if bitU_to_bool sub_op then NOT(operand2) else operand2 in
 read_reg_bitfield NZCV "C" >>= fun w__0 →
 let (result, nzcv) = AddWithCarry (operand1,operand2,w__0) in
 (if setflags then wPSTATE_NZCV (( ),nzcv) else return ( )) >>
 wX (d,result)
```

What remains to be done is define the monad type, and the return and bind operators, in a way that supports the interaction between instruction semantics and concurrency model captured by the outcome type. As it turns out, with one modification the outcome type from before can be made the monad type: in order to embed pure computations with arbitrary return type into the monad type, outcome has to be made polymorphic in an argument added to the Done constructor, where Done then becomes the unit/return of the monad.

```
type outcome 'a =
  | Read_mem of (read_kind * address * size) * (mem_value → outcome 'a)
  | Excl_res of (bool → outcome 'a)
  | Write_ea of (write_kind * address * size) * outcome 'a
  | Write_memv of memory_value * (bool → outcome 'a)
  | Barrier of barrier_kind * outcome 'a
  | Read_reg of reg_name * outcome 'a
  | Write_reg of reg_name * reg_value * outcome 'a
  | Internal of outcome 'a
  | Footprint of outcome 'a
  | Done of 'a
```

A value of type outcome 'a is an effectful computation that will eventually produce a result of type 'a; every such computation is either a finished computation Done a or a request that requires

interaction with the outside context, here the concurrency model. Every such request (Read_mem, Write_memv, etc.) has a description of the request and a continuation that, given an answer to the request, describes the remaining computation leading to a result of type 'a. The Internal outcome is an outcome generated by the Sail interpreter only, not the shallow embedding, purely for user-interface purposes (rmem uses it to print the Sail interpreter state when annotated with additional information).

The outcome type modified in this way can be seen as an instance of the free monad, as described by Swierstra [118] (Swierstra in turn attributes free monads to Awodey [29]). The free monad allows capturing effectful computations as compositions of such requests but leaving the meaning of the effects open to *interpretation*: an effectful term in the monad is a nested sequence of requests whose implementation can be delegated to an effect interpreter; the interpretation could then be done e.g. using a state monad in the sequential setting, or using a more complex memory model for concurrent programs.

With a more powerful type system than Lem offers the outcome type could be encoded as an instance of the following type (in pseudo-Lem):

```
type free 'f 'a =
  | Pure of 'a
  | Impure of ('f (Free 'f 'a))
```

With Pure taking the role of Done of the outcome type, and where 'f can be an arbitrary functor describing the effects of the language, in this case concretely instantiated in the following way to obtain the equivalent of the outcome type:

```
type effect 'a =
  | Read_mem of (read_kind * address * size) * (mem_value → 'a)
  | Excl_res of (bool → 'a)
  | Write_ea of (write_kind * address * size) * 'a
  | Write_memv of memory_value * (bool → 'a)
  | Barrier of barrier_kind * 'a
  | Read_reg of reg_name * 'a
  | Write_reg of reg_name * reg_value * 'a
  | Internal of 'a
  | Footprint of 'a
```

These definitions, however, require higher-kinded polymorphism (the ability to abstract over type constructors) that Lem does not support.

Without the abstraction given by the free type defined above the definition of the monad bind operator needs a case for each of the outcomes; the bind operator composes two computations by threading the answer value through the outcome types and nesting the requests, and the monad's return is the Done constructor.

```
val return : forall 'a. 'a → outcome 'a
let return a = Done a
```

```
val bind : forall 'a 'b. outcome 'a → ('a → outcome 'b) → outcome 'b
let rec bind m f = match m with
  | Done a → f a
  | Read_mem descr k → Read_mem descr (fun v → bind (k v) f)
  | Read_reg descr k → Read_reg descr (fun v → bind (k v) f)
  | Write_memv descr k → Write_memv descr (fun v → bind (k v) f)
  | Excl_res k → Excl_res (fun v → bind (k v) f)
  | Write_ea descr o → Write_ea descr (fun v → bind o f)
  | Barrier descr o → Barrier descr (fun v → bind o f)
  | Footprint o → Footprint (fun v → bind o f)
  | Write_reg descr o → Write_reg descr (fun v → bind o f)
  | Internal descr o → Internal descr (fun v → bind o f)
end
```

To illustrate the above definitions, consider the case of applying bind to a request m of type outcome bool and a function f of type bool → outcome unit. For example,

$$\begin{aligned} m &= \text{Done true} \\ f &= \text{fun } c \to \text{if c then Write\_memv descr (Done ()) else Done ()}. \end{aligned}$$

In the case of a finished computation Done v in the first argument, such as in the case of m, the monad bind simply "unpacks" the value v from the Done constructor and applies f to it (writing the infix $\gg\!=$ operator for bind):

$$\begin{aligned} m \gg\!= f &= (\text{Done true}) \gg\!= (\text{fun } c \to \text{if c then Write\_memv descr (Done ()) else Done ()}) \\ &= (\text{fun } c \to \text{if c then Write\_memv descr (Done ()) else Done ()})\,\text{true} \\ &= \text{Write\_memv descr (Done ())} \end{aligned}$$

If the left-hand side of bind is another request, the two requests are nested, for example for $m = \text{Excl\_res}\,(\text{fun } c \to \text{Done } c)$ and f as before.

$$\begin{aligned} m \gg\!= f &= (\text{Excl\_res}\,(\text{fun } c \to \text{Done } c)) \gg\!= \\ &\qquad (\text{fun } c \to \text{if c then Write\_memv descr (Done ()) else Done ()}) \\ &= \text{Excl\_res}\,(\text{fun } c \to (\text{Done } c) \gg\!= \\ &\qquad (\text{fun } c \to \text{if c then Write\_memv descr (Done ()) else Done ()})) \\ &= \text{Excl\_res}\,(\text{fun } c \to (\text{fun } c \to \text{if c then Write\_memv descr (Done ()) else Done ()})\,c) \\ &= \text{Excl\_res}\,(\text{fun } c \to \text{if c then Write\_memv descr (Done ()) else Done ()}) \end{aligned}$$

The result is a computation with two requests: first a request to determine the success of a store exclusive instruction, and, if the answer supplied by the concurrency model is true, a request to write to memory, otherwise a finished computation with unit return value.

With the outcome type adapted and the monad operations defined as above the shallow embedding supports the same interaction between the instruction semantics and the concurrency model as the interpreter. What is missing in order to use the shallow embedding in the concurrent setting is the instruction instance footprint analysis required by the concurrency model. There are two options for doing this, currently:

- writing by hand functions that, given a concrete instruction instance, return the footprint analysis, or
- analysing the instruction footprint using the interpreter.

The ISA models integrated into rmem currently do the former, with the analysis functions written as Sail definitions. This approach, however, is not very scalable: for complex Sail specifications it requires too much effort and is too error-prone, especially in the face of updates to the Sail specifications that may have to be reflected in the analysis functions. It should be possible to automatically generate these analysis functions from the Sail sources and map them to shallowly-embedded Lem definitions to discharge the Sail user from the effort of writing these functions by hand. This has not been attempted yet, however.

The shallow embedding as described above is integrated into rmem. With the addition of the currently hand-written instruction footprint analysis definitions, the shallow embedding is a functional replacement for the interpreter. Using the shallow embedding for simulation to run small sequential tests in rmem's sequential mode improves on the interpreter's performance by a factor of 10. Thomas Bauereiss has worked on improving the shallow embedding. His changes include improving the mappings for certain Sail types into Lem (for examples bit vectors) and adding support for additional Sail features, such as exceptions, non-determinism and early function returns, more sophisticated pattern matching, and registers holding values of types other than bit vectors. Thomas Bauereiss reports promising results in using the Isabelle definitions generated from the Lem shallow embedding for formal reasoning: Thomas Bauereiss used the shallow embedding of the aforementioned ASL-generated Sail ARMv8 model in an Isabelle proof about the architecture's virtual memory management in the sequential model, detailed in Armstrong et al. [24]. For the purpose of facilitating formal reasoning this includes using a sequential interpretation of the outcome type's requests to allow regaining a simpler state monad model for proofs in the sequential setting.

Chapter A in the appendix describes a method for semi-automatically generating instruction tests that were used to validate the instruction semantics against hardware.

Chapter 3

Background: non-MCA concurrency

Power, ARMv7, and early versions of the ARMv8 architecture have non-multicopy-atomic concurrency semantics. This chapter introduces (non-)multicopy atomicity and gives an overview over the operational models for Power and the non-multicopy-atomic ARMv8 architecture from previous work of Sarkar et al. [110, 111], Gray et al. [60], and Flur et al. [51, 52].

3.1 Introduction

An important property of a memory model is whether it is *multicopy atomic* or *non-multicopy-atomic*. Informally, a memory model is multicopy atomic if the following property holds of all writes:

> *When a write is visible to one thread other than its originating thread, it is visible to all other threads.*

In the ARM documentation [22] a memory model with this property is called "other-multicopy-atomic" [104] to distinguish this definition from the original definition of multicopy atomicity by Collier [43] that requires a stronger notion of atomicity: whenever a write is visible to any thread — including the write's originating thread — it has to be visible to all. In the following this text uses "multicopy atomic" to refer to the weaker notion of atomicity, with respect only to the non-writer threads, following recent accepted usage. (The above only gives an intuition for the concept, it is not a precise definition. A precise definition requires a definition of what it means for a write to be visible to a thread, which differs between concurrency models.)

To illustrate this property, consider the following litmus test with four threads, IRIW+addrs. Here Thread 0 writes 1 to x (event a), while Thread 2 writes 1 to y (d). Thread 1 first reads

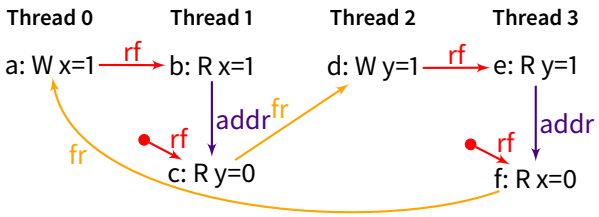


Figure 3.1: IRIW+addrs

from location x (with read b), before reading from location y (c), and Thread 3 symmetrically first reads from location y (e) and subsequently from x (f). On both threads the reads b and c, and e and f, respectively, are forced to satisfy in program order by an address dependency.

Despite the ordering of the reads on Threads 1 and 3 the concurrency models of Power and the early non-MCA ARMv8 architecture (also ARMv7) allow executions of this program where

Thread 1's read b reads from a and c reads from the initial write, coherence-before d (fr from c to d), while Thread 3's read e reads from d and f reads from the initial write, coherence-before a — Thread 1 "seeing" a but not d while Thread 3 sees d but not a. The reason the execution is allowed in these concurrency models is their non-multicopy-atomic semantics: it is possible for the writes a and d to propagate to Threads 1 and 3 in opposite orders — write a propagating to Thread 1 before propagating to Thread 3, and d propagating to Thread 3 before propagating to Thread 1.

The following sections introduce the non-multicopy-atomic concurrency models for Power and the early non-MCA ARMv8 architecture from previous work that the work in the following chapters is based on [110, 111, 60, 51, 52]. Since Chapter 4 on the NOP storage subsystem and Chapter 5 on mixed-size Sequential Consistency refer to details of the POP concurrency model [51, 52] for the non-multicopy-atomic ARMv8 architecture, and since the Flat concurrency model of Chapter 6 for the multicopy atomic ARMv8 architecture is a modification of POP, this chapter also gives a full definition of POP.

The Power and ARM concurrency models both only handle the concurrency aspects of user-mode programs, no system-level features. The models do not cover self-modifying code, the interaction of concurrency with address-translation, or exceptions.

3.2 Power

The Power architecture features a weak memory model that is non-multicopy-atomic: independent memory accesses to different addresses can be executed out of order, writes can be propagated between different threads out of order and can be observed to propagate to different threads separately. Ordering is provided by inter-instruction dependencies and through barriers that order thread-local execution. Some barriers also provide cumulativity, by which write propagation is constrained to ensure that threads "seeing" the barrier must also see certain writes previously propagated to the barrier's thread. Additionally, load reserve/store conditional instructions — Power's equivalent of ARM's load/store exclusive instructions — provide atomicity guarantees with respect to memory accesses by other threads useful in order to implement locks and other synchronisation primitives.

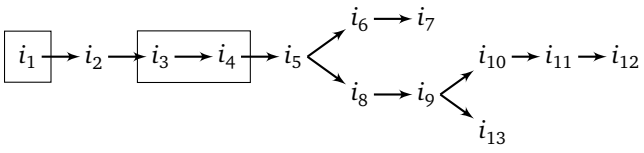
Sarkar et al. [110, 111] and Gray et al. [60] formally define the concurrency behaviour allowed by the Power architecture, in collaboration with IBM staff. This specification, henceforth called the *PLDI11 model*, is an operational model that explains the concurrency behaviour using mechanisms abstracting from those in real micro-architecture. The abstract machine model comprises two components, the thread subsystem and the storage subsystem.

3.2.1 Thread subsystem

The thread subsystem abstracts from the behaviour of individual hardware threads. In real CPUs the threads may execute instructions non-atomically, out-of-order, and speculatively (past conditional branch instructions). The PLDI11 model's thread subsystem abstractly captures the

effects of such optimisations by maintaining an *instruction tree* as part of the thread state: a tree of instruction instances corresponding to control flow unfoldings of the executed program. The thread subsystem enumerates transitions for the execution steps of instruction instances contained in this tree, as well as transitions for fetching new instruction instances from program memory as successor nodes to existing instruction instances as leaves of the tree. Instruction instances of conditional/computed branch instructions can have multiple successor nodes in the instruction tree corresponding to speculatively fetched possible successor instructions after the branch. (Some of the following is adapted from the more detailed description of the thread subsystem of the ARM models [51, 104]. Their thread subsystem in turn is based on the thread subsystem of the PLDI11 model by Sarkar et al. [110, 111].)

For example, the diagram below [110] shows a thread model state with instruction instances $i_1, \ldots, i_{13}$, and the program-order-successor relation between them. Instruction instances i_5 and i_9 are conditional branches for which the thread has fetched multiple possible successors. Both have only two successors each, but a branch with a computed address might have many possible successors. (The model has a parameter that enables or disables fetching multiple successors of branches. The possible final outcomes do not depend on this choice: for any trace that explores multiple subtrees of a branch instruction there is an equivalent one that only explores the "correct" one.) The instruction instances i_1, i_3, and i_4 (boxed) have been finished, the others are non-finished. Finishing a conditional or computed branch discards the subtrees of the branch corresponding to the untaken control flow choice(s). At any time some of the instruction instances in the tree may be finished, and others not. Instruction instances that follow (in program order) a non-finished



conditional/computed branch cannot finish until that branch is finished, and so any finished instruction instance's program-order prefix tree is linear. Non-finished instruction instances can also be subject to restart, e.g. if they depend on an out-of-order or speculative read that turns out to be unsound. If an instruction instance is sufficiently independent it can finish before some program-order-earlier instruction instances, such as i_3 and i_4 that are finished where i_2 is not.

The intra-instruction behaviour of a single instruction instance is mostly described by the sequential but non-atomic execution of its Sail pseudocode. Different instructions' Sail execution steps may be interleaved. Each instruction instance state includes a Sail execution state in the form of an element of the outcome type defined in Chapter 2. An instruction instance state also includes information about the instruction instance's memory and register footprints, its register and memory reads and writes, whether it is finished, etc.

Any transition of the thread subsystem is only enabled if its preconditions are met. Together with the transition preconditions in the storage subsystem these maintain the architecturally guaranteed properties such as coherence and the ordering given by barriers.

Whenever an instruction step produces a request to access memory or to commit a barrier the

thread subsystem generates an *event*, capturing the details of the request, such as the footprint of a memory access (address and size), the identifier of the requesting instruction, and the type of read, write, or barrier request. The thread subsystem synchronises with the storage subsystem on these memory access or barrier requests and generates transitions to hand the corresponding event over to the storage subsystem. When the Storage Subsystem synchronises with the thread subsystem on these transitions, the answer to read and write requests is instantaneous, whereas the storage subsystem has separate transitions for accepting a barrier request and for acknowledging a barrier in response. The thread subsystem records the barriers that are yet to be acknowledged by the storage subsystem.

In the context of mixed-size memory accesses, misaligned loads or stores do not provide the same atomicity guarantees as aligned loads and stores. To capture the correct behaviour for misaligned accesses a single misaligned load or store can generate multiple reads or writes in the thread subsystem, each of one byte size. In the following, a read or write *slice* will refer to a subfootprint of a read or write.

3.2.2 Storage subsystem

The storage subsystem abstracts from the details of store queues and multiple levels of caches. Instead of explicitly modelling memory and these components, the storage subsystem is defined in terms of sets of events or requests (in this section the terms *event* and *request* will be used interchangeably). It maintains

- a set of write events seen by the storage subsystem,
- per thread, a list of write slices and barriers propagated to that thread, including the writes done by this thread itself,
- a set of as yet unacknowledged thread barrier requests, and
- in order to maintain coherence, the constraints on the write coherence as a partial order on write events.

The storage subsystem is defined in terms of a number of transitions for receiving requests from the thread subsystem and some storage-subsystem-only transitions.

The storage subsystem can *accept write requests* and *accept barrier requests* from the thread subsystem, recording them in the storage subsystem state and marking them as propagated to their originating threads; *reply to a read request* from the thread subsystem by returning the writes most recently propagated to this thread that cover the read's footprint; and *acknowledge barrier requests* that have been propagated to all threads. In addition, it has transitions to *propagate a write event* and for the *partial coherence commitment*. Rather than establishing coherence implicitly as a consequence of the propagation of writes or other storage subsystem actions, the storage subsystem has an explicit transition in which it gradually refines the partial coherence order on writes to eventually totally order any two overlapping writes:

- For any two overlapping writes that are not yet (directly or transitively) coherence-related and subject to certain barrier conditions, the storage subsystem can take a *partial coherence commitment* transition, coherence-relating the two write-events.

- A write that has not been propagated to a certain thread yet and that has been made coherence-after all writes that are already propagated to this thread can be propagated to the thread using the *propagate write* transition, subject to certain barrier conditions.

Sarkar et al. [111] extend this model to handle load reserve/store conditional instructions. In Power, a store conditional can succeed or fail, and the architecture guarantees that a store conditional can succeed only if there is a program-order-preceding same-address load reserve and if the write conditional is guaranteed to become the immediate coherence successor of the write this load reserve read from (up to writes by the store conditional's thread). To model store conditional instructions, the model is extended with the concept of *coherence point*: when a write *reaches the coherence point*, its coherence prefix is linearly ordered and fixed [111]. Then the storage subsystem's transition for accepting a write-conditional request (among other things) orders the write conditional in the coherence order with other writes to the same address in a way that ensures there are no writes by other threads coherence-between the write the load reserve read from and the write conditional, and records the write-conditional as having reached coherence-point, guaranteeing no other write can later become its coherence predecessor.

The details of the transitions are omitted here but can be found in Sarkar et al. [110] and Sarkar et al. [111]; Flur et al. [52] describe the generalisation of this model to cover memory accesses of mixed sizes.

3.3 Non-MCA ARMv8

The non-multicopy-atomic ARMv8 architecture has a weak memory model broadly similar to that of the Power architecture. Whereas in Power ordering in the instruction execution and propagation of events is created using barriers, ARMv8 additionally has Load Acquire and Store Release instructions, so-called "half fences", that provide ordering with respect to memory accesses program-order-before (Store Release) or program-order-after (Load Acquire) them. The architectures agree on the allowed concurrency behaviour for many programs; in a number of important examples, however, they differ [51]. Moreover, Power and ARMv8 also differ in the architectural intent for the concurrency model, in how the architects explain the concurrency behaviour. Hence, non-MCA ARMv8 requires its own concurrency model formalisation: to model the precise architecturally allowed behaviour and to match the architecture's explanation for the concurrency behaviour, so the model can be discussed and confirmed with the architects. The Flowing and POP models of Flur et al. [51, 52] are operational concurrency models for the non-multicopy-atomic ARMv8 architecture designed in collaboration with ARM. Flowing and POP adopt the general structure of the PLDI11 model: an abstract machine model consisting of a thread subsystem and a storage subsystem.

3.3.1 Thread subsystem

Flowing and POP share the same thread subsystem, an adaptation of the PLDI11 thread subsystem: as in the PLDI11 model, Flowing and POP manage the out-of-order and speculative instruction

execution in the threads by maintaining a tree of instruction instances. The thread subsystem enumerates the possible transitions of the instructions in the instruction tree, and transitions for fetching new instruction instances into the tree; the intra-instruction semantics is mostly the sequential execution of its Sail code, where the multiple Sail execution steps of an instruction may be interleaved with those of other instructions; requests for memory accesses and barriers get passed to the storage subsystem.

There are two main sources of differences between the PLDI11 thread subsystem and that of Flowing and POP. Firstly, differences in the storage subsystems' handling of barrier and read requests lead to differences in the thread subsystem: in Power the transitions for the thread subsystem to commit a barrier into the storage subsystem and for the storage subsystem to respond back to the thread with a barrier acknowledgement are separate, in Flowing and POP, once the thread subsystem has committed the barriers into the storage subsystem there is no need for a barrier acknowledgement; conversely, whereas in Power a read request is immediately replied to by the storage subsystem in the same transition as the read request, in Flowing and POP read requests have a longer life time: the thread subsystem generates a read request that the storage subsystem accepts, and, after potentially several other storage-subsystem-internal transitions replies back for with a read reply. These storage subsystem differences lead to some differences in the details of the thread subsystem.

Secondly, the Power and ARMv8 architectures make different choices concerning the possible thread behaviours, some of which lead to different allowed behaviours in example programs. For instance, the MP+dmb.sy+fri-rfi-ctrlisb test [51] is forbidden by Power and allowed by (non-multicopy-atomic and multicopy atomic) ARMv8. In this example, the difference in the allowed behaviour between the two architectures comes from different choices concerning the mechanisms used to ensure coherence. (Test omitted; see [51, Section 3] for the details and other tests.) Due to these difference in certain architectural choices the details of the Power and Flowing/POP thread subsystems as well as some components of the thread state are different.

3.3.2 Flowing storage subsystem

The Flowing storage subsystem aims to abstractly capture the micro-architectural intuition described by the ARM architects for non-MCA ARMv8, and was developed in detailed discussion with ARM. Due to the difference in how ARM and IBM explain the architectures' concurrency behaviours, Flowing and POP specify the semantics using different mechanisms from the Power model. The Flowing storage subsystem state is a tree-shaped topology of buffers together with additional book-keeping information for load/store exclusive instructions. An example of a Flowing storage subsystem state is shown in Figure 3.2.

The buffer tree in the Flowing state has one buffer for each thread as its leaves — here Threads 0 through Thread 2 — and memory in the root — here holding values 0 for both x and y. The Flowing storage subsystem is parameterised in the exact shape of this tree topology. When events generated by thread subsystem requests enter the storage subsystem with the *accept event transitions* they are placed at the top of the buffer associated with their originating thread, in the

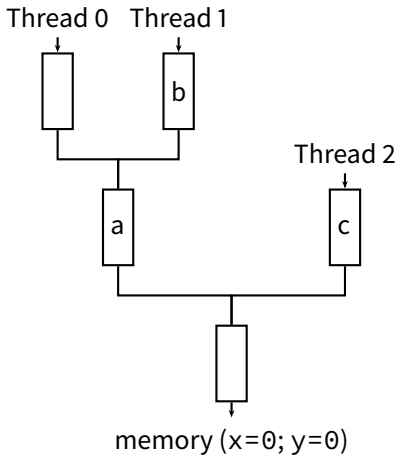


Figure 3.2: Example Flowing storage subsystem state

same way as *b* and *c*. These events in the storage subsystem can be writes and barriers but, in contrast to how the PLDI11 model works, also read events.

The storage subsystem has transitions for *flowing events down* the topology, and for re-ordering events with others, subject to the *re-order condition*. While writes and barriers flow down until they eventually reach memory in the root of the tree — writes updating the values held in memory — reads may be satisfied if they are above an adjacent overlapping write in the same buffer of the tree, which will remove the read request from the storage subsystem. Load acquire read requests leave an empty read request to create ordering.

The re-order condition specifies which events may re-order and which may not, in order to guarantee same-address event ordering as well as the ordering properties given by barriers and acquire/release instructions. For example, a plain read (one with no special ordering properties) and a plain write that do not overlap are allowed to re-order, but the re-order condition will forbid re-ordering any event with a `dmb sy` strong memory barrier.

The buffers in the Flowing buffer tree abstract from caches in the micro-architecture (e.g. per-thread L1 caches and shared higher-level caches). In Flowing a write event first enters its own thread's buffer and flows down towards memory, while correspondingly in micro-architecture writes update their own thread's cache first, before gradually propagating to the other caches of the system, and eventually memory. Similarly, in correspondence to reads flowing down in Flowing, in micro-architecture read requests typically will check for the availability of a value for their memory location in their own thread's cache first, and in the case of a cache miss fetch values from higher-level caches.

The PLDI11 model is a "coherence-by-fiat" model [110]: the PLDI11 storage subsystem maintains as part of the storage subsystem state the coherence relation as a partial order over the writes propagated into the storage subsystem so far, and that is gradually refined with an explicit transition for constraining the coherence order. In contrast to this, in the Flowing and POP models the coherence order is an emergent property of the execution resulting from the order of propagation of the writes in the storage subsystem: the coherence relation induced by a Flowing

storage subsystem state is the partial order given by the position of events along the tree topology restricted to overlapping writes — "further up in the tree" meaning coherence-later — and the order in which the writes flow into memory.

The transitions of Flowing are:

- *Accept event*. When an event is accepted the event is placed at the top of its originating thread's buffer.
- *Re-order event*. Two events can be re-ordered if the re-order condition is satisfied, swapping their positions in the buffer.
- *Flow event*. An event at the bottom of its containing buffer can flow into the buffer underneath, placing it at the top of that buffer.
- *Flow barrier or write to memory*. A barrier or write event at the end of the bottom-most buffer can flow into memory. This removes the event from the buffer and, in the case of a write event, updates memory.
- *Satisfy read*. A read can be entirely or partially satisfied either from an adjacent write event below the read in the same buffer, or from memory. Satisfying a read from memory always entirely satisfies the read — satisfies all previously unsatisfied slices of the read. If the read is fully satisfied after the transition its event is deleted from the storage subsystem; if not the read event is updated to record the newly satisfied slices, and the read event and the write event that partially satisfied it are swapped.

Similar to Power's load reserve/store conditional instructions ARMv8 has load/store exclusive instructions. A successful store exclusive guarantees that its write becomes the immediate coherence successor of the write its program-order-preceding load reserve read from (up to writes by the load and store exclusive's thread). Flowing provides the guarantees of store exclusives by maintaining the invariant that in any model step the path in the flowing topology from a successful write exclusive down to the write the load exclusive read from is free of same-address writes by other threads.

This description omits details of the model definition, e.g. related to release/acquire instructions and load/store exclusive instructions. The full definition can be found in Flur et al. [51, 52].

3.3.3 Examples

Simple first example We illustrate the Flowing model's behaviour with some example executions. Consider first, the `MP` test from the introduction, again shown in Figure 3.3 (which assumes on both threads `X1` holds the address of x and `X3` that of y). Here, Thread 0 first writes $x = 54$ and then $y = 1$; Thread 1 reads $y = 1$ (from Thread 1's write), but subsequently reads the (old) initial write $x = 0$.

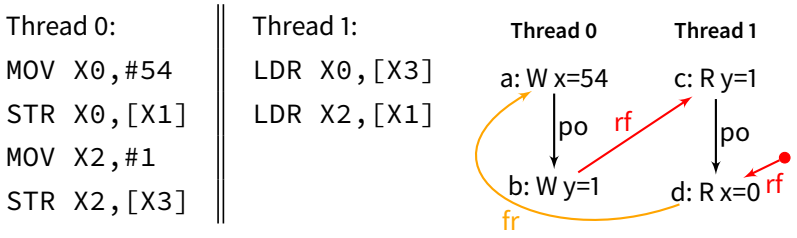


Figure 3.3

Flowing allows the behaviour as follows: Thread 0 executes both `MOV` instructions, saving the value of the stores into registers `X0` and `X2`; Thread 0 does the first steps of the store to x—reading register `X0` holding the address, announcing the address x, reading register `X0` holding the value, and committing the store—until it can propagate a into the storage subsystem; this places a onto the Flowing buffer belonging to Thread 0; similarly, Thread 0 executes the first steps of b's store instruction and propagates b into the storage subsystem, placing b above a on Thread 0's buffer; Thread 1 executes the first steps of c's load instructions (reading `X3` and announcing the address y) until it can issue c it to the storage subsystem; this does not satisfy c yet, but just inserts it into the Flowing buffer of Thread 1, leaving the load waiting for the storage subsystem to reply with a read value for c; c's load on Thread 1 waiting for c's return value does not stop other instructions from executing, and Thread 1 can execute d and also issue it to the storage subsystem. At this point the storage subsystem state is as shown in Figure 3.4a. Now, the behaviour can be allowed in two ways:

- a and b are to different addresses. Hence, the storage subsystem allows re-ordering a and b, flipping their positions on the buffer. With b now at the bottom of Thread 0's buffer, it can flow down to the buffer shared by Threads 0 and 1. Now c can flow down to the same buffer, making it adjacent above a (see state shown in Figure 3.4b). Now, c can read $y = 1$ from the adjacent same-address write b, removing it from the storage subsystem and sending a read reply to its load instruction. In the thread subsystem c's load is now satisfied and can finish. Now d can also flow down onto the bottom buffer, re-order with b (which is to a different address) and flow into memory, satisfying it with $x = 0$, before a reaches the bottom buffer. Finally, b and a separately flow into memory, updating the memory state to $x = 54$ and $y = 0$.
- Alternatively, instead of re-ordering a and b, the storage subsystem in the state shown in Figure 3.4a could re-order c and d. This swaps their position and places d below c in the buffer, where it can now flow to the buffer shared between Threads 0 and 1, and then into memory, reading the initial $x = 0$. After that, a and b can separately flow to the bottom buffer and then into memory, updating the memory state to $x = 54$ and $y = 1$; finally c can flow down into memory (via the bottom buffer) and read $y = 1$.

(The same behaviour can also be allowed with slightly different steps, including just the out-of-order execution of a and b or of c and d within the threads.) Placing `dmb sy` barriers between

both a and b and c and d prevents the execution: the barrier between a and b will not allow b to propagate to the storage subsystem until the `dmb sy` is also in the storage subsystem, which can only happen after a has propagated to the storage subsystem. The `dmb sy` is then placed between b and a on Thread 0's buffer and prevents a and b from re-ordering. Hence, the first way the MP execution was allowed without the barriers, shown before, does not work with a `dmb sy` between a and b. On Thread 1, the `dmb sy` stops d from issuing into the storage subsystem until c is satisfied, also preventing the second way of allowing the behaviour from before.

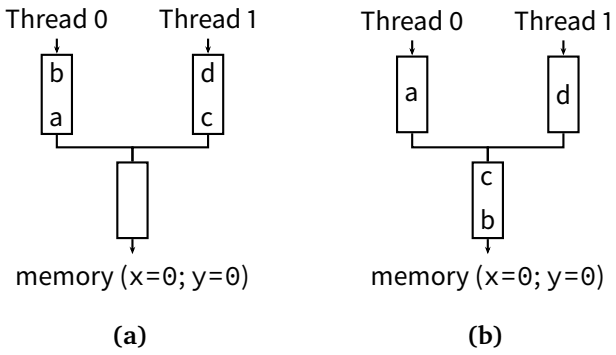


Figure 3.4

Non-multicopy-atomic storage subsystem Flowing's storage subsystem is non-multicopy-atomic. Consider the previously shown IRIW+addrs example, again shown in Figure 3.5. In Flowing this is allowed as follows, assuming a topology like the one shown in Figure 3.6a: (focussing on the storage subsystem behaviour)

1. Propagate a to the storage subsystem. This places it on Thread 0's buffer. Propagate d to the storage subsystem, placing it on Thread 2's buffer.
2. The write a can flow to the buffer shared between Threads 0 and 1, and d to that shared between Threads 2 and 3 (Figure 3.6a).
3. Now issue b and e, placing them on the buffers of their respective threads (Figure 3.6b).
4. b and e can now flow down one level, placing b above a on the buffer shared between Threads 0 and 1 and e above d on that shared between Threads 2 and 3.
5. Now b can read from a—removing it from the storage subsystem, sending a read reply to its thread, satisfying its load, and resolving c's address dependency. Similarly, e can read from d, resolving f's address dependency.
6. Now c and f can issue and flow down one level, placing c just above a and f above d (Figure 3.6c).
7. But since c and a, and f and d, are to different addresses they can, respectively, re-order with each other; c is then below a and can flow down and into memory (before d reaches memory), reading $y = 0$; similarly, f can flow into memory and read $x = 0$ (before a reaches memory), leading to the outcome shown in Figure 3.5.

To illustrate some aspects of the thread subsystem behaviour, consider the following two tests. The first illustrates restarts in the thread subsystem.

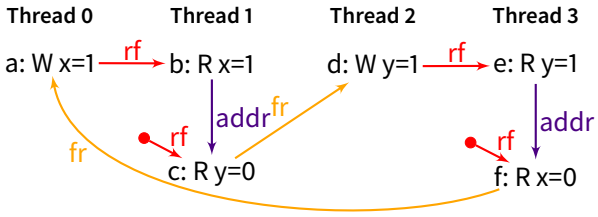


Figure 3.5: IRIW+addrs

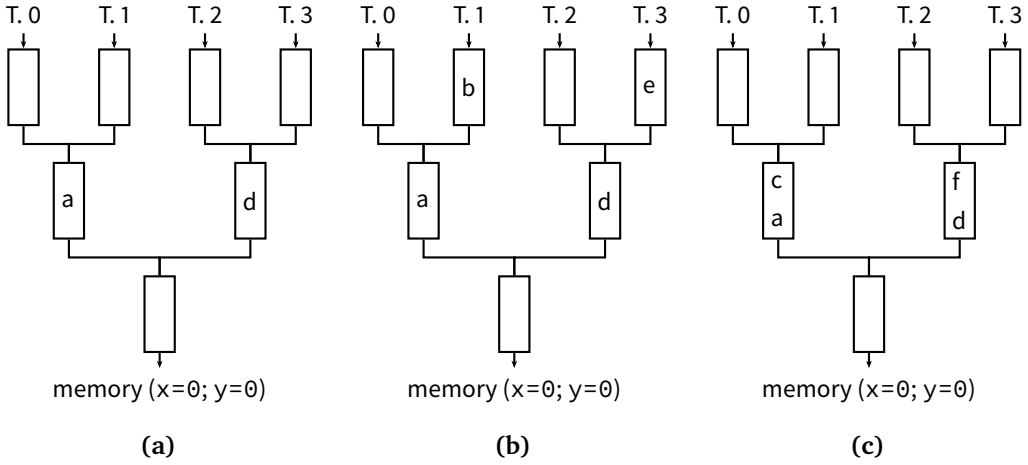


Figure 3.6

Thread subsystem restarts Assume Thread 0 and Thread 1's registers `X0` and `X3` hold the addresses of the locations x and y, respectively. When the program refers to `W0` rather than `X0` for the data register of a store this means a four-byte, rather than an eight-byte store. Correspondingly, a load with target register `W0` means a four-byte load rather than an eight-byte load.

In the test shown in Figure 3.7 Thread 0 does a write a of four bytes size to x (indicated by $x/4$), of value 54, and, after a strong barrier, does a four-byte write b of value 1 to y; Thread 1 reads four bytes at address y with c, uses the returned value in calculating the address of another four-byte load d, which in this example will always be to x, and subsequently reads eight bytes from location x with e. The test checks whether it is possible for c to read 1 from b, and e to see 0 for the eight bytes of x, whichever value d reads.

Flowing forbids this behaviour. The barrier between a and b enforces ordering between the propagation of a and b into the storage subsystem and between them flowing into memory. Hence, when Thread 1 can "see" b it must also see a. In order to allow the behaviour, e would have to satisfy before c reads from b. Consider the following execution:

1. The load of e does not have data dependencies on any of the preceding instructions. Therefore it can do its register reads, initiate, and issue the read e into memory.
2. The read e can flow down and into memory, and read from the initial value $x = 0$.
3. Thread 0 propagates a into the storage subsystem and flows it into memory, commits the barrier and flows it into memory, and propagates b into the storage subsystem and flows it

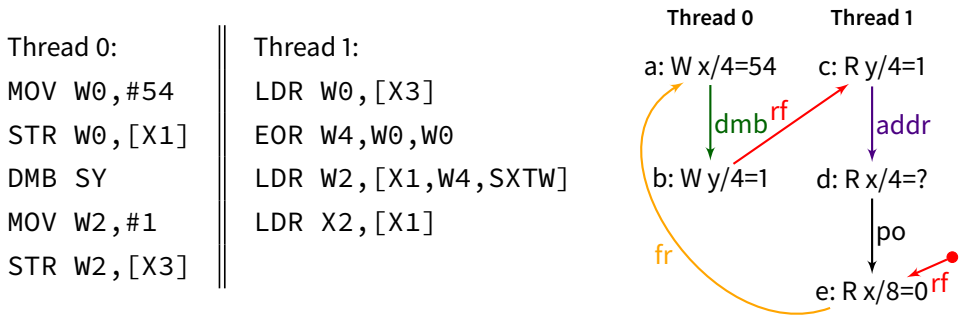


Figure 3.7

into memory.

4. Thread 1 issues read c, flows it into memory, and satisfies c from b.
5. Thread 1 issues read d. Since a is in memory, d will not be able to read from the initial write to x.
6. The read d flows into memory and is satisfied by reading from a.
7. Once d is satisfied by a, the thread subsystem finds a coherence violation: the read e is program-order-after d, and the part of the read e overlapping d was satisfied from a different write than the part of d, and this write is not program-order-after d. Therefore the load of e is restarted.
8. When e now issues into the storage subsystem again, it will not be able to read 0 (as a combined value for all bytes), since a is already in memory.
9. e flows into memory, reading the first four bytes from a and the other four bytes from the initial write to x.

Hence, in this execution where (initially) e is satisfied early, the resulting coherence violation causes a restart in the thread subsystem that prevents the test outcome.

Thread-internal forwarding The next example illustrates thread-internal forwarding. Recall the PPOCA test [110] from the introduction, again shown in Figure 3.8.

For the program on the left, assume on both threads the register `X1` initially holds the address x, `X3` holds address y, and `X7` holds address z. In this variant of the MP test, the reads c to y and d to x on Thread 1 are separated by a chain of thread-local dependencies. The return value of c's load is saved in register `W0`. Following the load, a conditional branch checks whether the returned value is 0. If so, it branches to the label `end`. If not, it executes a program-order-subsequent store to location z, and a load of z whose value in turn is used in calculating the address of the read d to x. The test assumes the load to z reads from its program-order predecessor store, and checks whether the dependency chain on Thread 1 requires d to execute after c and is thus forced to read $x = 1$. Despite this dependency chain, the execution in which d reads $x = 0$ after c reads $y = 1$ is allowed in Flowing: (skipping over some thread and storage subsysetm details)

1. The instructions of e, f, and d can be fetched speculatively.
2. Since the store of e has no address or data dependencies it can do its register reads, and

Thread 0:	Thread 1:
MOV W0,#54	LDR W0,[X3]
STR W0,[X1]	CBZ W0,end
DMB SY	MOV W2,#1
MOV W2,#1	STR W2,[X7]
STR W2,[X3]	LDR W4,[X7]
	EOR W5,W4,W4
	LDR W6,[X1,W5,SXTW]
	end:

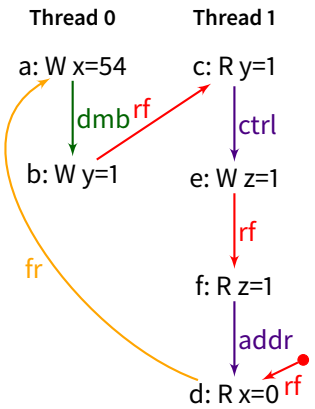


Figure 3.8: PPOCA

announce its address and value.

3. The load of f has no dependencies, and can do its register reads and also announce its address.
4. While e's store is not allowed to commit and propagate to the storage subsystem yet, due to the control dependency, it can thread-locally *forward* the write e to f. This satisfies f and f can resolve the address dependency of d.
5. Then d can announce its address, issue to the storage subsystem, flow down, and satisfy from the initial write to x.
6. Thread 0 propagates the write a into the storage subsystem and flows it into memory, commits the barrier and flows it into memory, and propagates b and flows it into memory.
7. Thread 1 issues c, flows it into memory, and satisfies it from b. It writes the return value to `W0`.
8. Since `W0` is non-zero, the conditional branch does not branch, and so the instructions in the instruction tree starting from e are not discarded.

Hence the execution of the test is allowed.

3.3.4 Topology-dependence

In Flowing, the tree topology of buffers connecting the threads with memory has an effect on the allowed behaviour and Flowing is parameterised in the choice of the topology. For example, the test IRIW+addrs of Figure 3.5 is only allowed with a topology that allows propagating the writes to locations x and y to Threads 1 and 3 in opposite orders. One such topology is that shown for the example execution of IRIW+addrs above. This behaviour is not possible, for example, in the topology in Figure 3.9 (swapping Threads 1 and 2): the previously shown execution relied on propagating a to Thread 1 without propagating it to Thread 3, and propagating d to Thread 3 without propagating it to Thread 1. In the topology below, Thread 1 can only see a when a is visible to all threads, and Thread 3 can only see d when d is visible to all threads.

Flowing's dependence on a choice of topology is undesirable for an architectural model. Therefore the POP model abstracts from Flowing in order to be topology-independent.

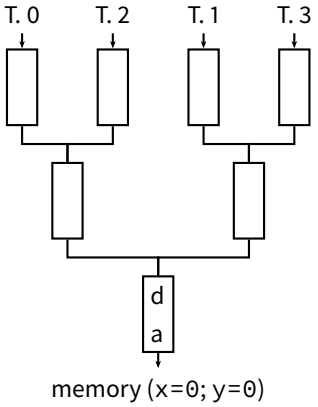


Figure 3.9: A topology forbidding IRIW+addrs

3.3.5 POP storage subsystem

The POP storage subsystem abstracts from Flowing in two ways. Firstly, in contrast to Flowing, POP is thread-topology independent, and secondly, POP has no explicit re-ordering of events. POP achieves both of these by replacing Flowing's buffer tree in which events flow down by a partially ordered set of events and, per-thread, the set of writes propagated to that thread. POP retains Flowing's explicit event propagation transition, but whereas in Flowing events are maintained in the buffers (that impose a linear order) and can be re-ordered, POP's order is partial on event pairs that can be re-ordered in Flowing. This partial order allows for "shapes" in the ordering and propagation of events corresponding to arbitrary thread topologies. In fact, POP allows certain executions not allowed by Flowing instantiated with any topology: allowing IRIW+addrs imposes some constraints on the buffer topology; sequencing IRIW+addrs four times using different thread permutations [51, Section 13] requires "switching buffer topologies" during the execution and therefore is allowed in POP but not Flowing. The following section gives the detailed definition of the POP model, since some later definitions and proofs are based on it, but for simplicity omits the defintions of load/store exclusive instructions (whose POP definition the rest of the thesis does not require).

3.4 POP full definition

Some details of POP have changed since the original publications on POP [51, 52] and the following presents the updated definition, in order to better align with the definition of the simplified operational concurrency model for the multicopy atomic ARMv8 architecture based on POP, in Chapter 6.[1] The definitions given here are the prose description of the rules of the

[1]Hence, much of the informal and formal description of the POP model here is based on that for the simplified model presented in Pulte et al. [104] and the POP definition given by Flur et al. [51, 52].

formal model that is written in the Lem specification language and integrated into rmem. The model description is written to closely follow the Lem definitions but sometimes, for the sake of readability, abstracts from specifics of the code, especially in places where the complexity in the formal model is only necessary for executability.

There are a few points to note concerning the version of POP as presented here: First, a difference from the definition given by Flur et al. [51] is in the thread subsystem transition types and terminology for load and store instructions, partly due to the handling of memory accesses of mixed sizes:

- Loads. Once a load's memory footprint is provisionally known, the load is said to be *initiated*, and the transition in which the footprint becomes known (before the load can be issued to the storage subsystem or satisfied by forwarding) is called *load initiation*. In the original POP definition there was no specific terminology for this.
  The read request of a misaligned load is split into several byte-sized read requests that are satisfied individually. Once all are satisfied the values are combined. In the earlier versions of POP, combining these values took place in the transition for satisfying the last of these read requests; in the version as presented here, there is a dedicated *complete load* transition for this.
  Previously loads were said to *commit* after all reads were satisfied and once certain conditions were met; the new terminology is that the load *finishes* after it completes and satisfies these conditions.
- Stores. Similarly to the case of loads, the transition in which a store's address becomes provisionally known is called *store initiation*, and the store is said to be *initiated* after that. Analogously to the case of loads, the write request of a misaligned store is split into several write events. In the earlier version of POP, committing the store placed all of the store's writes into the storage subsystem at once. In the version as presented here, store commitment is a purely thread-local action, that only checks the thread subsystem's conditions under which the store is known to happen; after this point, the writes belonging to the store can start propagating into the storage subsystem in individual *propagate write* transitions.
  Once all writes of a store are propagated to memory (in this new version of POP) the store can *complete*, and subsequently *finish*.

Also note that compared with [51] there are minor differences in whether certain predicates check for instructions being *finished* as opposed to being *committed*.

Second, since none of the following chapters refer to details in the definitions of load/store exclusive instructions this description omits these for simplicity. Third and last, POP was parameterised in a choice of whether to allow "write subsumption" — whether a write can propagate into memory before program-order-earlier overlapping writes have been propagated. Since the multicopy atomic ARMv8 architecture forbids this (see Chapter 6) this definition only describes the case where write subsumption is forbidden, for simplicity.

Model states A model POP state consists of the storage state and a tuple of thread model states. The storage subsystem state comprises a set of events seen by the storage subsystem, a partial

order on the events, and, per thread, the set of events propagated to this thread, together with some additional information needed for the atomicity guarantees given by load/store exclusives (omitted here). As in the PLDI11 model, each thread model state consists principally of a tree of instruction instances; each instruction instance state includes a pseudocode execution state of the outcome type. An instruction instance state also includes information, detailed below, about the instruction instance's memory and register footprints, its register and memory reads and writes, whether it is finished, etc.

Model transitions For any state, the model defines the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. The possible transitions in any state are the transitions initiated by the thread subsystem and those initiated by the storage subsystem; some transitions, such as the transition in which the storage subsystem responds to a thread's read request require the thread subsystem and the storage subsystem to synchronise. The interface is defined in terms of memory events: requests to read or write memory and to commit a barrier that are passed from the threads to the storage subsystem. Each thread transition arises from the next step of a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its thread state and/or synchronise with a storage subsystem state, but it will not change other thread states, and mostly not change other instruction instance states. Instructions cannot be treated as atomic units: complete execution of a single instruction instance may involve many transitions, which can be interleaved with those of other instances in the same or other threads, and some of this is programmer-visible. In some cases, one instruction's actions can cause its restart, or the restart of program-order-subsequent instructions: in order to avoid coherence violations due to memory instructions to overlapping footprints that were executed out of order.

The transitions are introduced below and defined in Section 3.4.6, with a precondition and a post-transition model state for each.

Thread transitions of all instructions

- Fetch instruction: This transition represents a 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.
- Register read: This is a read of a register value from the most recent program-order predecessor instruction instances that write to that register.
- Register write: This records a register write in the instruction instance state to make it available for other instruction instances to read.
- Pseudocode internal step: This covers Sail internal computation, function calls, etc.
- Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted or discarded, and all memory effects have taken place. For a conditional/computed branch, any non-taken po-successor branches are discarded.

Thread transitions of load instructions

- Initiate memory reads of load instruction: At this point the memory footprint of the load is provisionally known and its individual reads can start being satisfied.
- Satisfy memory read by forwarding from writes: This partially or entirely satisfies a single read by forwarding from its po-previous writes.
- Issue read request to the storage subsystem: This sends a read request for the unsatisfied read slices to memory.
- Satisfy memory read from memory: This satisfies the unsatisfied slices by a read response sent by the storage subsystem.
- Complete load instruction: At this point all reads of the load have been entirely satisfied and the instruction pseudocode can continue execution. A load instruction can be subject to being restarted until the Finish instruction transition. In some cases it is possible to tell that a load instruction will not be restarted or discarded before that, e.g. when all the instructions po-before the load instruction are finished. The Restart condition over-approximates the set of instructions that might be restarted.

Thread transitions of store instructions

- Initiate memory writes of store instruction: At this point the memory footprint of the store is provisionally known.
- Instantiate memory write values of store instruction: At this point the writes have their values and program-order-subsequent 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 the writes can start being propagated to memory.
- Propagate memory write to storage: This sends a write request for a single write to the storage subsystem.
- Complete store instruction: At the point when for all writes a request has been sent to memory, the store can complete and the instruction pseudocode can continue execution.

Thread transitions of barrier instructions

- Commit barrier: This commits a barrier, and, for barriers other than `dmb ld` and `isb`, sends a barrier request event to the storage subsystem.

Storage subsystem transitions

- Accept request: This accepts a new request from a thread subsystem to the storage subsystem.
- Propagate request to another thread: This propagates an event to a new thread.
- Partially or entirely satisfy read request: This partially or entirely satisfies a read request and sends a reply to the read request's threads.

Thread/storage subsystem interface The thread and storage subsystem synchronise on the following transitions [51, Section 7]:

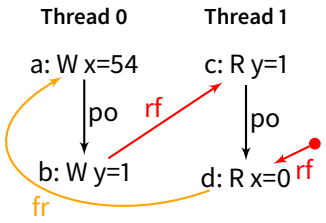
- a write request can be sent to the storage subsystem from a thread with a Propagate memory write to storage transition synchronising with a storage subsystem Accept

REQUEST transition;

- a thread can send a (memory) barrier request to the storage subsystem with a COMMIT BARRIER transition synchronising with a storage subsystem ACCEPT REQUEST transition;
- a thread can send a read request to the storage subsystem with a ISSUE READ REQUEST TO THE STORAGE SUBSYSTEM transition synchronising with a storage subsystem ACCEPT REQUEST transition; and
- the storage subsystem can send a read response to a thread with a storage subsystem PARTIALLY OR ENTIRELY SATISFY READ REQUEST transition synchronising with a thread SATISFY MEMORY READ FROM MEMORY transition.

3.4.1 POP examples

To give an intuition for the behaviour of POP, the following will revisit the examples previously shown for Flowing and describe how they behave in POP. For the purpose of these examples, assume that for every memory location in the program there exists an initial write of value 0 in memory, propagated to all threads. This matters only for the discussion of these example programs, since the rules and the later proofs do not consider such initial writes.



Simple first example Consider again the previously shown MP test. This execution is allowed in POP as follows: (This omits details on the thread steps, which are exactly as previously shown for Flowing for the same example.)

1. In the initial state, the writes iw_x and iw_y are in memory and propagated to all threads, providing initial 0 values for the locations x and y.
2. When propagating a into memory, POP relates it with iw_x, since iw_x is propagated to Thread 0, and the re-order condition does not hold between the events, as they overlap. Similarly, after propagating b, POP relates it with iw_y; but b is not related with a, as they are to different addresses and the re-order condition holds. The diagram in Figure 3.10a shows Order as a graph and indicates the propagation sets as a superscript.
3. When Thread 1 now issues c, it is Order related with iw_y; when Thread 1 also issues d, it is related to iw_x, but not c (Figure 3.10b).
4. The write b can propagate to Thread 1 even though a is not propagated there yet: since they are Order unrelated they can propagate independently. Propagating b adds an edge from b to c: c is propagated to Thread 1 but not b's thread, Thread 0, and the re-order condition does not hold.
5. The read c can propagate to Thread 0, adding no new edges (Figure 3.10c).
6. c can now read from b, satisfying the load and removing c from the storage subsystem

(Figure 3.10d).

7. Now d can propagate to Thread 0, adding an edge from d to a, since a is propagated to Thread 0 but not d's thread, Thread 1, yet. Then d can satisfy from the initial write iw_x.
8. Finally, a propagates to Thread 1, completing the execution.

This execution corresponds to the first execution shown for Flowing for the same example. Corresponding to the second execution shown for Flowing, it is also possible to allow the behaviour by — after the third step — propagating the read d to Thread 0 and making it read from the initial write iw_x, before propagating a and b to Thread 1, propagating c to Thread 0, and satisfying c from b. Just like in Flowing, the behaviour can also be allowed purely in the thread subsystem, not shown here.

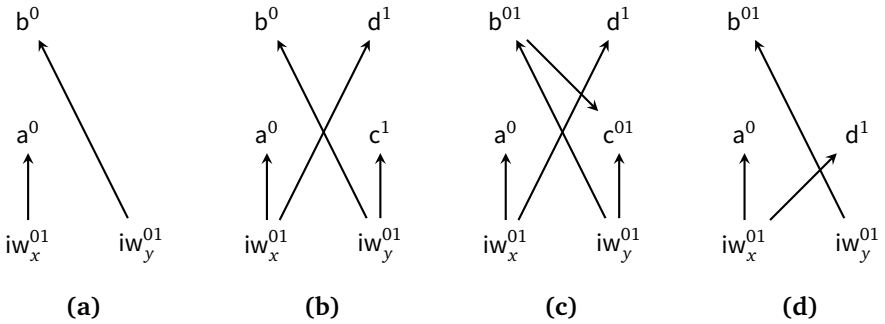


Figure 3.10

Non-multicopy-atomic storage subsystem Now consider the previous IRIW+addrs example. In POP, this execution is allowed as follows:

1. When propagating a into memory, POP relates it with iw_x, since iw_x is propagated to Thread 0, and the re-order condition does not hold between the events, since they overlap. Similarly, after propagating d, POP relates it with iw_y (Figure 3.11a).
2. The write a can propagate to Thread 1, and d to Thread 3 (not adding new edges).
3. If Thread 1 now issues the read request b, it is Order related after a, since they overlap; similarly for e: see Figure 3.11b (omitting transitive edges).
4. b and e can now propagate to Threads 0 and 2 and read from a and d, respectively — deleting them from the storage subsystem, and allowing c and f to resolve the address dependency and initiate (Figure 3.11c).
5. When c and f now issue, they are only ordered with the initial writes, since the re-order condition holds for c and a and for f and d (Figure 3.11d).
6. Now c and f can propagate to all other threads and read from the initial writes iw_y and iw_x, respectively, leading to the outcome shown in Figure 3.5.

Thread subsystem restarts and thread-internal forwarding For Flowing we considered two other tests, the test of Figure 3.7 illustrating thread subsystem restarts, and the PPOCA test of Figure 3.8 that shows thread-internal forwarding. Flowing and POP use the same thread subsystem. Hence, in the two tests, the example executions shown for Flowing are the same

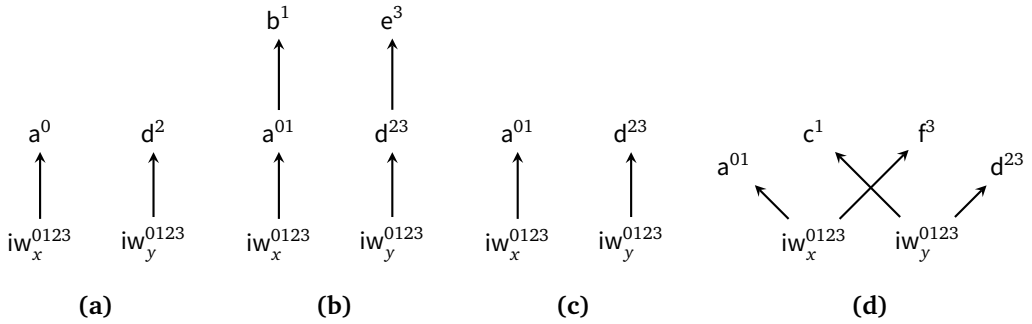


Figure 3.11

in POP — except for irrelevant storage subsystem differences. Hence, for a description of the behaviour for those examples see Section 3.3.3.

3.4.2 Intra-instruction pseudocode execution

To link the model transitions introduced above to the execution of the instructions, POP uses the outcome type defined in Chapter 2 as an interface between Sail and the rest of the concurrency model. The requests occurring in POP, without load/store exclusives, are:

Read_mem (read_kind, address, size, read_continuation)	Read request
Write_ea (write_kind, address, size, next_state)	Write effective address
Write_memv (memory_value, write_continuation)	Write value
Barrier (barrier_kind, next_state)	Barrier
Read_reg (reg_name, read_continuation)	Register read request
Write_reg (reg_name, register_value, next_state)	Write register
Internal (next_state)	Pseudocode internal step
Done ( )	End of pseudocode

The hand-written Sail code for the ARMv8 instructions covered by this model ensures that each instruction has at most one memory read, memory write, or barrier step, by rewriting the pseudocode to coalesce multiple reads or writes, which are then split apart into the architecturally atomic units by the thread semantics; this gives a single commit point for all memory writes of an instruction. Each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit, or from the thread's initial register state if there is no such. That instance may not have executed its register write yet, in which case the register read should block. As discussed in Chapter 2 the semantics has to know therefore the register write footprint of each instruction instance, which it calculates when the instruction instance is created. The pseudocode is adapted so that each instruction does exactly one register write to each bit of its register output footprint, and also so that instructions do not do register reads from their own register writes. In some cases, but not in the fragment of ARM covered by this model at present, register write footprints need to be dynamically recalculated, when the actual footprint only becomes known during pseudocode execution.

Data-flow dependencies in the model emerge from the fact that a register read has to wait for

the appropriate register write to be executed (as described above). In order for this not to create unintentional strength the pseudocode is adapted to ensure a sequencing of commands that gives the maximally liberal order.

Consider for instance, the example of a store instruction that, in addition to doing a memory write, writes the computed address of its memory access into a register (address write-back). Transcribing the pseudocode description in the architecture documentation naively into Sail one might arrive at the following high-level steps for such a store instruction:

1. read the address register,
2. compute the address,
3. announce the address,
4. read the data register,
5. compute the data,
6. do the write,
7. do the address write-back.

If, however, there is a program-order-succeeding instruction that reads the store instruction's write-back register, that instruction will only be able to access this register and resolve its data dependency when the preceding store has done its write, creating a dependency that is unwanted for Power and ARM. Thus, special care has to be taken in the ordering of the intra-instruction Sail actions in order not to create such dependencies, instead typically choosing the maximally liberal ordering. In this example of the store with address write-back this would mean ordering the register write-back to happen as early as the address is available:

1. read the address register,
2. compute the address,
3. do the address write-back,
4. announce the address,
5. read the data register,
6. compute the data,
7. do the write.

Additionally, the model needs to know when a register read value can no longer change (i.e. due to instruction restarts). This is approximated by recording, for each register write, the set of register and memory reads the instruction instance has performed at the point of executing the write. This information is then used as follows to determine whether a register read value is final: if the instruction instance that performed the register write from which the register reads from is finished, the value is final; otherwise check that the recorded reads for the register write do not include memory reads, and continue recursively with the recorded register reads. For the instructions covered by the model this approximation is exact. POP is integrated with both the Sail interpreter and the shallow embedding from Chapter 2.

3.4.3 Instruction instance states

Each instruction instance i has a state comprising:

- program_loc, the memory address from which the instruction was fetched;
- instruction_kind, identifying whether this is a load, store, or barrier instruction, each with the associated kind; or a conditional/computed branch; or a 'simple' instruction.
- regs_in, the set of input registers/register slices, as statically determined;
- regs_out, the output registers/register slices as statically determined;
- pseudocode_state (or sometimes just 'state' for short), one of
  - Plain (*next_state*), ready to make a pseudocode transition;
  - Pending_mem_reads (*read_cont*), performing the memory read(s) of a load; or
  - Pending_mem_writes (*write_cont*), performing the memory write(s) of a store;
- reg_reads, the accumulated register reads, including their sources and values, of this instance's execution so far;
- reg_writes, the accumulated register writes, including dependency information to identify the register reads and memory reads (by this instruction) that might have affected each;
- mem_reads, a set of memory read requests. Each request includes a memory footprint (an address and size) and, if the request has already been satisfied, the set of write slices (each consisting of a write and a set of its byte indices) that satisfied it.
- mem_writes, a set of memory write requests. Each request includes a memory footprint and, when available, the memory value to be written. In addition, each write has a flag that indicates whether the write has been propagated (passed to the memory) or not.
- information recording whether the instance is committed, finished, etc.

Read requests include their read kind and their memory footprint (their address and size), the as-yet-unsatisfied slices (the byte indices that have not been satisfied), and, for the satisfied slices, information about the write(s) that they were satisfied from. Write requests include their write kind, their memory footprint, and their value.

When the text refers to a write or read request without mentioning the kind of request this means the request can be of any kind. A load instruction which has initiated (so its read request list mem_reads is not empty) and for which all its read requests are satisfied (i.e. there are no unsatisfied slices) is said to be *entirely satisfied*.

3.4.4 Thread states

The model state of a single hardware thread comprises:

- thread_id, a unique identifier of the thread;
- register_data, the name, bit width, and start bit index for each register;
- initial_register_state, the initial register value for each register;
- initial_fetch_address, the initial fetch address for this thread;
- instruction_tree, a tree or list of the instruction instances that have been fetched (and not discarded), in program order; and
- read_issuing_order, a list of read requests recording the order in which the requests were issued to the storage subsystem.

3.4.5 Storage subsystem state

The storage subsystem state comprises:

- Threads, the set of thread IDs that exist in the system;
- Events, the set of requests (memory read/write requests and barrier requests) that have been seen by the subsystem;
- Order, the set of pairs of requests from Events. The pair (r_{old}, r_{new}) indicates that r_{old} is before r_{new} (r_{old} and r_{new} might be to different addresses and might even be of different kinds); and
- EProp, a map from thread identifiers to subsets of Events, associating with each thread the set of requests that have propagated to it.

3.4.6 Model transitions

Fetch instruction A possible program-order successor of instruction instance i can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of i in the thread's instruction_tree are from *loc*;
2. *loc* is a possible next fetch address for i:
   2.1. for a non-branch non-jump instruction, the successor address i.program_loc + 4;
   2.2. for an instruction that has performed a write to the program counter register _PC, the value that was written;
   2.3. for a conditional branch, either the successor address or the branch target address (in AArch64, all conditional branch instructions have statically determined target addresses); or
   2.4. for a jump to an address which is not yet determined, any address (which in the executable rmem tool is approximated); and
3. there is a decodable instruction in program memory at *loc*.

Note that this allows speculation past conditional branches and calculated jumps.

Action: Construct a freshly initialised instruction instance i' for the instruction in program memory at *loc*, including the static footprint information such as its instruction_kind, regs_in, and regs_out, and add i' to the thread's instruction_tree as a successor of i.

This involves only the thread, not the storage subsystem, as the model assumes a fixed program and does not model fetches with memory reads — the model does not handle self-modifying code.

Initiate memory reads of load instruction An instruction instance i with next state Read_mem (*read_kind*, *address*, *size*, *read_cont*) can initiate the corresponding memory reads.

Action:

1. Construct the appropriate read requests *rrs*:
   - if *address* is aligned to *size* then *rrs* is a single read request of *size* bytes from *address*;
   - otherwise, *rrs* is a set of *size* read requests, each of one byte, from the addresses *address*

$\ldots\ address + size - 1$.

2. set i.mem_reads to *rrs*; and
3. update the state of i to Pending_mem_reads (*read_cont*).

Satisfy memory read by forwarding from writes For a load instruction instance i in state Pending_mem_reads (*read_cont*), and a read request r in i.mem_reads that has unsatisfied slices and that has not been sent to memory yet, the read request can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances that are po-before i, if the *read-request-condition* predicate holds. This is if:

1. all po-previous `dmb sy` and `isb` instructions are finished;
2. all po-previous `dmb ld` instructions are finished;
3. if i is a load acquire, all po-previous store releases are finished; and
4. all non-finished po-previous load acquire instructions are entirely satisfied.

Let *wss* be the set of unpropagated write slices from store instruction instances po-before i that have already calculated the value to be written, that overlap with the unsatisfied slices of r, and which are not superseded by intervening writes (with known address) or writes that are read from by intervening loads. That last condition requires, for each write slice *ws* in *wss* from instruction i':

- that there is no store instruction po-between i and i' with a write overlapping *ws*, and
- that there is no load instruction po-between i and i' that was satisfied from an overlapping write slice from a different thread.

Action:

1. update r to indicate that it was satisfied by *wss*; and
2. restart any speculative instructions which have violated coherence as a result of this: for every non-finished instruction i' that is a po-successor of i, and every read request r' of i' that was issued before r and satisfied from wss', if there exists a write slice ws' in wss', and an overlapping write slice from a different write in *wss*, and ws' is not from an instruction that is a po-successor of i, restart i' and its data-flow dependents (including po-successors of load acquire instructions).

Issue read request to the storage subsystem For a load instruction instance i in pseudocode state Pending_mem_reads (*read_cont*), and a read request r in i.mem_reads that has not been issued to the storage subsystem yet, and that either has unsatisfied slices or is from a load acquire, the read request can be issued to the storage subsystem if the *read-request-condition* predicate holds. This is if (as above):

1. all po-previous `dmb sy` and `isb` instructions are finished;
2. all po-previous `dmb ld` instructions are finished;
3. if i is a load acquire, all po-previous store releases are finished; and
4. all non-finished po-previous load acquire instructions are entirely satisfied.

Action:

1. send a read-request to the storage subsystem;
2. update r to record it as issued to memory;

3. update read_issuing_order to note that the read was issued to the storage subsystem.

Note that this allows issuing read requests for entirely satisfied load acquires. The model issues this "empty read" token into memory to preserve the acquire ordering guarantees.

Satisfy memory read from memory A load instruction instance i in state Pending_mem_reads (*read_cont*) can always receive a response for a read request r from the storage subsystem. *Action*: let *wss* be the write slices from memory covering the unsatisfied slices of r.

1. if there exists a po-previous load instruction that has a read request r' that was issued after r (issued out of order) and there exists a write slice ws' in the set of write slices that satisfied r' and a write slice *ws* in *wss* such that *ws* and ws' overlap, are from different writes, and *ws* is from a write that is not program-order-before i, then: restart i and its data flow dependents; else
2. update r to indicate that it was satisfied by *wss*; and
3. restart any speculative instructions which have violated coherence as a result of this: for every non-finished instruction i' that is a po-successor of i, and every read request r' of i' that was issued before r and satisfied from wss', if there exists a write slice ws' in wss', and an overlapping write slice from a different write in *wss*, and ws' is not from an instruction that is a po-successor of i, restart i' and its data-flow dependents (including po-successors of load acquire instructions).

Note that Satisfy memory read by forwarding from writes might leave some slices of the read request unsatisfied. Satisfy memory read from memory, on the other hand, will always satisfy all unsatisfied slices of the read request.

Complete load instruction (when all its reads are entirely satisfied) A load instruction instance i in state *Pending_mem_reads (read_cont)* can be completed (not to be confused with finished) if:

- all the read requests i.*mem_reads* are entirely satisfied (i.e., there are no unsatisfied slices), and
- if i is a load acquire, then i has issued a read request to the storage subsystem.

Action: update the state of i to Plain (*read_cont memory_value*), where *memory_value* is assembled from all the write slices that satisfied i.mem_reads.

Note that the reason for the second condition is the load acquire token that is issued into memory also for entirely satisfied loads.

Initiate memory writes of store instruction, with their footprints An instruction instance i with next state Write_ea (*write_kind*, *address*, *size*, *next_state*$'$) can announce its pending write footprint.

Action:

1. construct the appropriate write requests:
   - if *address* is aligned to *size* then *ws* is a single write request of *size* bytes to *address*;
   - otherwise *ws* is a set of *size* write requests, each of one byte size, to the addresses $address \ldots address + size - 1$.

2. set i.mem_writes to ws; and
3. update the state of i to Plain ($next\_state'$).

Note that at this point the write requests do not yet have their values. This state allows non-overlapping po-following writes to propagate.

Instantiate memory write values of store instruction An instruction instance i with next state Write_memv ($memory\_value$, $write\_cont$) can instantiate the corresponding memory writes.

Action:

1. split $memory\_value$ between the write requests i.mem_writes; and
2. update the state of i to Pending_mem_writes ($write\_cont$).

Commit store instruction For an uncommitted store instruction i in state Pending_mem_writes ($write\_cont$), i can commit if:

1. i has fully determined data (i.e., the register reads cannot change, see Section 3.4.7);
2. all po-previous conditional/computed branch instructions are finished;
3. all po-previous `dmb sy` and `isb` instructions are finished;
4. all po-previous `dmb ld` instructions are finished;
5. all po-previous load acquire instructions are finished;
6. all po-previous store instructions have initiated and so have non-empty mem_writes; if i is a store release, all po-previous memory access instructions are finished;
7. all po-previous `dmb st` instructions are finished;
8. all po-previous memory access instructions have a fully determined memory footprint; and
9. all po-previous load instructions have initiated and so have non-empty mem_reads.

Action: record i as committed.

Propagate memory write to storage For an instruction i in pseudocode state Pending_mem_writes ($write\_cont$), and an unpropagated write w in i.mem_writes, w can be propagated if:

1. i is committed;
2. all memory writes of po-previous store instructions that overlap w are propagated;[2]
3. all read requests of po-previous load instructions that overlap w have been issued or the load is entirely satisfied, and the load instruction is non-restartable (see Section 3.4.7); and
4. all read requests that have read from w by forwarding and that are either not satisfied yet or are from a load acquire have issued the request.

Action:

1. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' po-after i and every read request r' of i' such that either:
   - r' was satisfied from wss' and there exists a write slice ws' in wss' that overlaps with w and is not from w, and ws' is not from a po-successor of i, or
   - r' overlaps w and has been issued but not satisfied yet,

   restart i' and its data-flow dependents (including po-successors of load acquire instructions);
2. send a write request to the storage subsystem; and

[2]This condition is needed for the model as presented here to forbid write subsumption, as discussed earlier.

3. record *w* as propagated.

Complete store instruction (when its writes are all propagated) A store instruction *i* in pseudocode state Pending_mem_writes (*write_cont*), for which all the memory writes in *i*.mem_writes have been propagated can be completed.

Action: update the state of *i* to Plain(*write_cont* true).

Commit barrier A barrier instruction *i* in pseudocode state Plain (*next_state*) where *next_state* is Barrier(*barrier_kind*, *next_state*′) can be committed if:

1. all po-previous conditional/computed branch instructions are finished;
2. if *i* is a `dmb ld` instruction, all po-previous load instructions are finished;
3. if *i* is a `dmb st` instruction, all po-previous store instructions are finished;
4. all po-previous barriers (of any kind) are finished;
5. if *i* is an `isb` instruction, all po-previous memory access instructions have fully determined memory footprints; and
6. if *i* is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished.

Action:

1. if *i* is not a `dmb ld` or `isb`, send a barrier request to the storage subsystem; and
2. update the state of *i* to Plain (*next_state*′).

Register read An instruction instance *i* with next state Read_reg (*reg_name*, *read_cont*) can do a register read if every instruction instance that it needs to read from has already performed the expected register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from initial_register_state. Let *register_value* be the assembled value from *read_sources*.

Action:

1. add *reg_name* to *i*.reg_reads with *read_sources* and *register_value*; and
2. update the state of *i* to Plain (*read_cont register_value*).

Register write An instruction instance *i* with next state
Write_reg (*reg_name*, *register_value*, *next_state*′) can do the register write.

Action:

1. add *reg_name* to *i*.reg_writes with *write_deps* and *register_value*; and
2. update the state of *i* to Plain (*next_state*′).

where *write_deps* is the set of all *read_sources* from *i*.reg_reads and a flag that is set to true if *i* is a load instruction that has already been entirely satisfied.

Pseudocode internal step An instruction instance *i* with next state Internal(*next_state*′) can do that pseudocode-internal step.

Action: update the state of *i* to Plain (*next_state*′).

Finish instruction A non-finished instruction i with next state `Done()` can be finished if:

1. if i is a load instruction:
   1.1. all po-previous `dmb sy` and `isb` instructions are finished;
   1.2. all po-previous `dmb ld` instructions are finished;
   1.3. all po-previous load acquire instructions are finished;
   1.4. it is guaranteed that the values read by the read requests of i will not cause coherence violations, i.e., for any po-previous instruction instance i', let *cfp* be the combined footprint of propagated writes from store instructions po-between i and i' and fixed writes that were forwarded to i from store instructions po-between i and i' including i', and let $\overline{cfp}$ be the complement of *cfp* in the memory footprint of i. If $\overline{cfp}$ is not empty:
      1.4.1. i' has a fully determined memory footprint;
      1.4.2. i' has no unpropagated memory write that overlaps with $\overline{cfp}$; and
      1.4.3. If i' is a load with a memory footprint that overlaps with $\overline{cfp}$, then all the read requests of i' that overlap with $\overline{cfp}$ are satisfied or issued in order and i' cannot be restarted (see Section 3.4.7).

      Here a memory write is called fixed if it is the write of an instruction that has fully determined data.
   1.5. if i is a load acquire, all po-previous store release instructions are finished;
2. i has fully determined data; and
3. all po-previous conditional/computed branches are finished.

Action:

1. if i is a branch instruction, discard any untaken path of execution, i.e., remove any (non-finished) instructions that are not reachable by the branch taken in instruction_tree; and
2. record the instruction as finished, i.e., set finished to true.

3.4.7 Auxiliary definitions

Fully determined Informally, an instruction is said to have *fully determined footprint* if the memory reads feeding into its footprint are finished. A register write *rw*, of instruction i, with the associated *write_deps* from i.reg_writes is said to be fully determined if one of the following conditions holds:

1. i is finished; or
2. the load flag in *write_deps* is false and every register write in *write_deps* is fully determined.

An instruction i is said to have *fully determined data* if all the register writes of *read_sources* in i.reg_reads are fully determined. An instruction i is said to have a *fully determined memory footprint* if i has done enough instruction steps to compute its footprint and all the register writes of *read_sources* in i.reg_reads that are associated with registers that feed into i's memory access footprint are fully determined.

Restart condition To determine if instruction i might be restarted the following recursive condition is used: i is a non-finished instruction and at least one of the following holds,

1. there exists an unpropagated write w such that applying the action of the PROPAGATE MEMORY WRITE TO STORAGE transition to w will result in the restart of i;
2. there exists a non-finished load instruction l such that applying the action of the SATISFY MEMORY READ FROM MEMORY transition to l will result in the restart of i (even if l is already entirely satisfied);
3. i has an outstanding read request r that has not been satisfied yet, and there exists a program-order-earlier load instruction i' that has an outstanding read request overlapping r (maybe already satisfied) issued after i; or
4. there exists a non-finished instruction i' that might be restarted and i is in its data-flow dependents (including po-successors of load acquire instructions).

3.4.8 Storage Subsystem Transitions

First define the re-order condition that is used in the definition of the transitions.

Re-order condition Two requests r_{new} and r_{old} are said to meet the *re-order condition* if all of the following hold:

1. neither r_{new} nor r_{old} is a `dmb sy`;
2. r_{new} is not a write release;
3. r_{old} is not an unsatisfied read acquire;
4. if r_{new} is a read acquire and r_{old} is a write release, r_{new} and r_{old} originated from different threads;
5. if r_{new} is a read acquire and r_{old} is a write release, then none of the slices the read acquire has read from and that overlap r_{old} are from the same thread as r_{old};
6. if both r_{new} and r_{old} are memory access requests, then they have a non-overlapping footprint, where the footprint is defined as the write footprint in the case of write requests and the unsatisfied slices in the case of read requests;
7. if r_{new} is a write request and r_{old} is a read request, then r_{old} has not been partially satisfied by r_{new};
8. if r_{new} is a `dmb st` then r_{old} is not a write from the same thread; and
9. if r_{old} is a `dmb st` then r_{new} is not a write.

Accept request A request r_{new} from thread r_{new}.tid can be accepted if:

1. r_{new} has not been accepted before, so r_{new} is not in Events; and
2. r_{new}.tid is in thread_ids.

Action:

1. add r_{new} to Events;
2. add r_{new} to EProp(r_{new}.tid); and
3. update Order to note that r_{new} is after every request r_{old} that has propagated to thread r_{new}.tid where r_{new} and r_{old} do not meet the Flowing *re-order condition* (see RE-ORDER CONDITION).

Propagate request to another thread The storage subsystem can propagate request r (by thread tid) to another thread tid' if:

1. r has been seen before (r is in Events);
2. r has not yet been propagated to thread tid'; and
3. every request that is before r in Order has already been propagated to thread tid'.

Action:

1. add r to EProp(tid'); and
2. update Order to note that r is before every request r_{new} that has propagated to thread tid' but not to thread tid, where r_{new} and r do not meet the Flowing *re-order condition* (see RE-ORDER CONDITION) and are not already ordered.

Partially or entirely satisfy read request The storage subsystem can partially or entirely satisfy a read request r_{read} from thread r_{read}.tid by a write request r_{write} and send a read response if:

1. r_{write} overlaps with the unsatisfied slices of r_{read};
2. r_{write} and r_{read} have been propagated to (exactly) the same threads;
3. r_{write} is *fully-propagated* if and only if r_{read} is fully-propagated (see FULLY PROPAGATED);
4. if r_{read} is a read acquire and r_{write} is a write release then r_{write} must be fully-propagated;
5. r_{write} is Order-before r_{read}; and
6. any request e that is Order-between r_{write} and r_{read} where there exists a thread that both e and r_{read} have been propagated to, is fully-propagated and does not overlap the unsatisfied slices of r_{read}.

Action:

1. update r_{read} to record the remaining unsatisfied slices and the write slices read from r_{write}, reading the maximal overlapping slices;
2. if r_{read} is now entirely satisfied send thread r_{read}.tid a read response for r_{read} containing the write slices r_{read} has read from (including the slices from r_{write});
3. if r_{read} is entirely satisfied and not an acquire read remove r_{read}; else unless r_{read} is fully-propagated, switch the positions of r_{read} and r_{write} in Order; and
4. remove from Order pairs that satisfy the Flowing *re-order condition*, and apply transitive closure to the result.

3.4.9 Auxiliary Definitions for Storage Subsystem

Fully propagated Request r is said to be fully-propagated if it has been propagated to all threads and so has every request that is Order-before it.

Removing read request When a read request is removed from the storage subsystem due to restart or discard of the instruction that generated the read request in the thread subsystem Order is updated in the following way: Order is restricted to exclude the read request; it is then restricted to only those (ordered) pairs of events that cannot be re-ordered according to the *re-order condition*; then Order is transitively closed.

Chapter 4

NOP: Abstracting from POP

This chapter describes work on a more abstract storage subsystem model for the non-MCA ARMv8 architecture that abstracts from the explicit event propagation in POP. The model is formalised in Lem. The following illustrates the idea underlying NOP, gives the detailed prose definitions of NOP, a hand proof that NOP allows all behaviours allowed by POP, and experimental POP/NOP equivalence results and performance data.

Flowing and POP explain the architecturally allowed behaviour of non-MCA ARMv8 in terms of the interleaving of thread and storage subsystem transitions. The thread subsystem executes the instruction semantics out of order, handles register reads and writes, and, for loads, stores, and barriers, sends requests to the storage subsystem; the storage subsystem handles the propagation of these events between the threads and replies to the thread subsystem's read requests. To understand the legal behaviours of a concurrent program one has to think about the possible ways in which the thread subsystem and the storage subsystem may interact, and while the thread transitions are initiated by the actions of an instruction, the storage subsystem transitions for propagating events and for replying to read requests are "spontaneous" from the point of view of the executed code, in order to match the micro-architectural intuition described by the ARM architects for the non-MCA architecture.

The goal of the NOP storage subsystem is to simplify the formalisation of the concurrency model of the non-MCA ARMv8 architecture and eliminate some of this spontaneous storage subsystem behaviour by removing the explicit propagation of events from POP. Removing the event-propagate transitions means abstracting over the different ways in which event-propagate transitions may be interleaved with each other and with the other transitions of the model. In the resulting NOP model all transitions except the storage subsystem's satisfy-read transition are initiated by the thread subsystem. Eliminating explicit event propagation makes the model less micro-architectural compared to Flowing and POP, but has two advantages. Firstly, the semantics is expressed more directly in terms of instruction actions (less in terms of storage subsystem behaviour). Secondly, the change has performance benefits: in the executable model's exhaustive search for Flowing and POP many of the different interleavings of certain model transitions with event-propagate transitions lead to equivalent traces — traces which eventually converge in the same state. Eliminating some of these equivalent traces improves the exhaustive search's complexity.

The NOP storage subsystem could thus have been a first step towards a programmer-friendlier model for the non-MCA ARMv8 architecture that soundly abstracts from POP in a way that expresses the semantics of non-MCA ARMv8 programs more directly in terms of the instruction actions. Since ARM have shifted to a multicopy-atomic architecture, the model is looser than the revised architecture intends, and the multicopy-atomicity allows for greater simplifications

than NOP (or some more abstract model based on NOP) offers, and the Flat and Promising-ARM models presented later both are simpler operational models for MCA ARMv8.

Abstracting from POP's explicit event propagation does not mean NOP does away with the concept of propagation (that is central in a non-multicopy-atomic concurrency model). Instead, the propagation of events is determined by the accept-event and satisfy-read transitions. While NOP's accept-event transition is the same as POP's, NOP shifts the work of managing the propagation of events between threads to the satisfy-read transition: the satisfy-read transition for a read R by a write W lazily applies the minimal set of event propagations necessary to allow R to read from W, and makes R and W sufficiently Order-adjacent for R to be able to read from W in a corresponding POP execution.

IRIW+addrs To illustrate this, consider, for instance, the previously discussed IRIW+addrs example. Recall the previous POP execution allowing this behaviour, as follows:

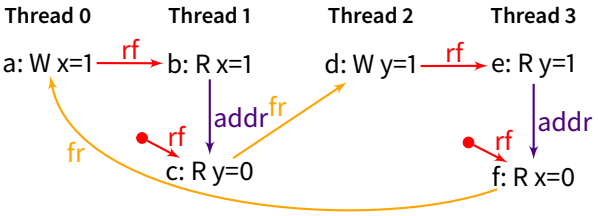


Figure 4.1: IRIW+addrs

1. Initially the writes iw_x and iw_y are in memory.
2. When propagating a into memory, POP relates it with iw_x.
3. When propagating d, POP relates d with iw_y.
4. The write a can propagate to Thread 1, and d to Thread 3.
5. When Thread 1 issues b, it is Order-related after a; similarly for e.
6. The read b can now propagate to Thread 0 and read from a; similarly, e can propagate to Thread 2 and read from d.
7. When c and f issue, they are only ordered with the initial writes, since the re-order condition holds for c and a and for f and d.
8. The reads c and f propagate to all other threads and read from the initial writes iw_y and iw_x, respectively.

In NOP the corresponding execution will be as follows:

1. Propagate a and d into memory.
2. Issue b. Issuing in NOP is the same as in POP. After issuing b, both a and b are in storage but Order-unrelated, since a is not propagated to b's thread yet.
3. Issue e. Then e and d are Order-unrelated (Figure 4.2a).
4. Now b can be satisfied from a. The transition computes the state after b read from a in a two-step process:
   4.1. NOP first computes (as an auxiliary step) a state roughly corresponding to a POP state in which b could read from a. In such a POP state, a must be Order-before b and a and b propagated to a common thread. So NOP constructs a state adding the edge

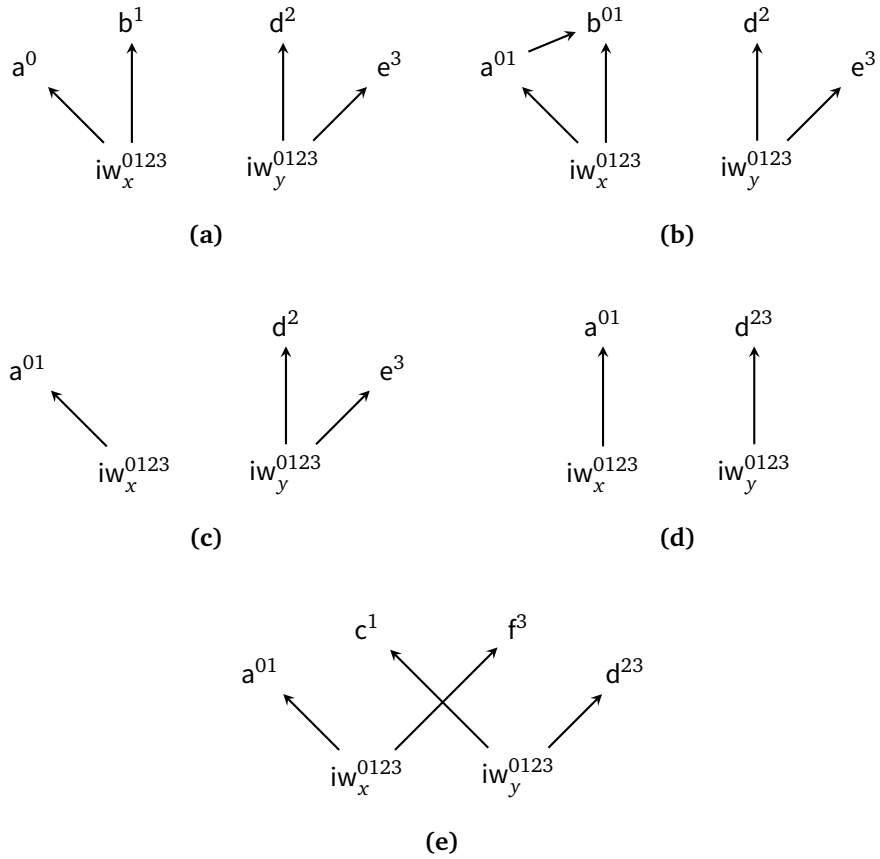


Figure 4.2

$a \rightarrow b$ to Order, and marking b, a, and all events Order-before them as propagated to any thread b or a have already propagated to before (Figure 4.2b).

4.2. To this auxiliary state, NOP applies an action corresponding to the action of POP's satisfy-read transition (Figure 4.2c).

Satisfying b from a is only allowed in NOP, if the state reached after this transition has an acyclic Order relation. Since in this case it does, the transition is enabled. Note that the auxiliary intermediate step in the computation of the post-transition state does not become exposed to the model.

5. In the same way, e can now read from d, resulting in the state shown in Figure 4.2d.
6. Now the addresses of c and f are resolved and they can issue (Figure 4.2e).
7. In the same way as previously, c can now read from the initial write to y, and f from the initial write to x, allowing the test behaviour.

IRIW+dmbs In contrast to this, the behaviour of the following test, IRIW+dmbs, that replaces the address dependencies with strong barriers, is forbidden. Let B_{bc} and B_{ef} be the barriers between b and c and between e and f, respectively. Now consider the following execution, for example:

1. As before, a and d propagate to memory, and b and e issue (Figure 4.4a).
2. Now b can read from a, and e can read from d, resulting in the state shown in Figure 4.4b

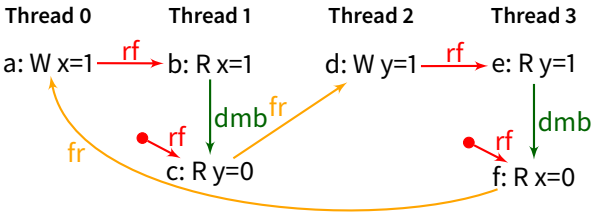


Figure 4.3: IRIW+dmbs

3. Commit both barriers (Figure 4.4c).
4. And issue c and f (Figure 4.4d).
5. Now c is allowed to read from iw_y. However, due to the barrier B_{bc} Order-between c and iw_y, the satisfy-read transition seen so far does not apply (the transition would lead to a cyclic Order, not shown here). In POP, c can only read from iw_y in the presence of this barrier B_{bc} between c and iw_y if B_{bc} and iw_y are fully propagated, and if there is no other same-address write Order-between c and iw_y. NOP has a second satisfy-read transition, corresponding to this case. The construction of the resulting state in this transition is as follows:
   Add the edge $iw_y \rightarrow c$, and make c and iw_y Order-adjacent with respect to all same-address writes: here d is to the same address as c and iw_y; in POP, if c is propagated to all threads it must be ordered with d, and since c would not be able to read from iw_y if d was Order-between them, d must be ordered after c; therefore NOP adds the edge $c \rightarrow d$. Finally, mark c, iw_y, and all events Order-before them as propagated to all threads, and record c as satisfied.
   The state is as shown in Figure 4.4e (omitting transitive edges).
6. The resulting state's Order has the transitive edge $a \rightarrow f$. This edge will now prevent f from reading from iw_x and force it to read from a instead, forbidding the example behaviour.

Model caveats The NOP storage subsystem does not handle mixed-size accesses, or load/store exclusive instructions, but the design should be able to accommodate both of these. NOP is proved sound with respect to POP: any behaviour allowed by POP is allowed by NOP. On the suite of non-mixed-size litmus tests, for the around 2200 that terminate in reasonable time POP and NOP are experimentally equivalent (allow the same final states).

There is no proof of soundness of POP with respect to NOP. Showing that NOP allows all behaviours allowed by POP is easier than the opposite. In order to show that all NOP behaviours are allowed by POP with a simulation argument (as in the proof of soundness of NOP given here), the proof would have to show how to construct a POP trace from any given NOP trace. This is difficult since NOP eliminates the propagation transitions, and since the propagation information in the NOP state is "less precise" than that of a POP state. Moreover, shortly after completing the work on the NOP model and its soundness proof presented here ARM switched to the non-MCA architecture.

As mentioned in the introduction, at the time of working on NOP, some details of the POP storage subsystem were different, and so NOP is defined and proved sound with respect to this

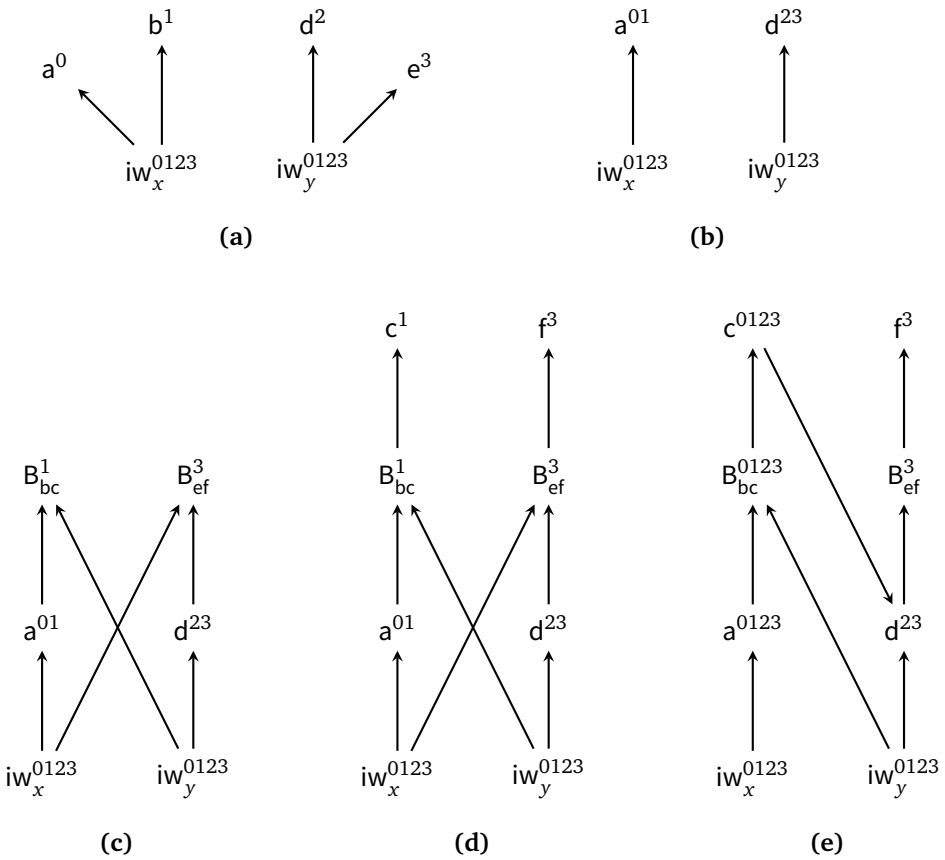

a0
b1
iwx0123
d2
e3
iwy0123
(a)
a01
iwx0123
d23
iwy0123
(b)
Bbc1
Bef3
a01
d23
iwx0123
iwy0123
(c)
c1
f3
Bbc1
Bef3
a01
d23
iwx0123
iwy0123
(d)
c0123
f3
Bbc0123
Bef3
a0123
d23
iwx0123
iwy0123
(e)

Figure 4.4

older version of POP. The remaining part of this chapter defines the version of POP that NOP is based on (relative to current POP) and the NOP storage subsystem, and gives the proof of soundness and the experimental performance results.

4.1 POP definition

At the time of working on NOP the definition of the POP storage subsystem was different from the current one (as defined in Section 3.4) in its treatment of acquire loads. In the current definition of POP, acquire loads are not removed from the storage subsystem state when they are satisfied, instead leaving an "acquire token" in the storage subsystem: a read event with empty footprint but that maintains the special acquire ordering properties. This is required to allow some events to re-order with the satisfied load acquire while still providing the desired ordering properties.

The POP version that NOP is based on provides stronger guarantees by maintaining the full ordering properties of any load after it is satisfied: ignoring the details concerned with mixed-size memory accesses, when a read is satisfied by a write that is not fully-propagated, the read is merged into the write; the updated write event has the ordering properties of the read and the write; if the write is already fully-propagated at the point of the read-satisfaction the read maintains its current position in `Order`. Moreover, originally POP merged the write and the read in both these cases. For convenience the following assumes POP does not merge the write and the read when the write is fully-propagated, which does not change the allowed behaviour but facilitates relating POP and NOP states in the proof of soundness of NOP with respect to POP.

The merging of the read and write events has no additional strength in the case of a plain read. When the read request and the write request have the same footprint, the write request already has all the ordering strength of the plain read (and more, in the case of a release write). Only in the case of a load acquire the read request creates additional ordering with respect to events from the load's thread, which the merging of the requests transfers to the write request that satisfied it.

This means this older version of POP is different with respect to the current one in the action of the satisfy-read transition, and the satisfy-read condition is relaxed to allow a read r to read from a write w also when there is a same-address read r' `Order`-between r and w if r' is fully-propagated and satisfied. The older POP version differs from the current one also in the re-order condition, which in the case of write events has to consider the read requests merged with the read requests. In the older version of POP, unlike in the current one, `dmb ld` barriers when committed enter the storage subsystem and provide extra ordering there. Moreover, the thread subsystem of the version of POP that NOP is based on follows [51] in forbidding thread-local forwarding to acquire loads.

The changes, informally described above and formally defined in the following section, are not essential to the way NOP abstracts from POP: while the concrete definition of NOP given here assumes that the satisfaction of a read request by a write request in the storage subsystem merges the read request and the write request, NOP's formal definition (in Lem, in rmem) has been adapted for current POP in a way that is experimentally equivalent for the non-mixed-size test suite. Since, however, the proof relating NOP and POP given in the following is not updated

to these changes, this chapter presents NOP defined with respect to this now out-of-date version of POP.

Caveats: NOP does not handle load/store exclusives and does not handle mixed-size memory accesses. Extending NOP to handle both these should not pose fundamental problems, but due to time constraints and since ARMv8 has moved to the multicopy atomic architecture NOP has not been extended/updated in this way. Since NOP does not handle mixed-size memory accesses the POP definition given here also assumes memory accesses of the same size, for a simpler presentation. Below are the differences of the old version of the POP storage subsystem to the one presented earlier.

Satisfy-read transition *Condition:* The satisfy-read condition holds for a read request r and write request w to the same address in state s if:

- r and w are propagated to the same threads,
- (w, r) in $s.\mathsf{Order}$,
- if w is a release write and r is an acquire read then both are fully-propagated, and
- for all e with $\{(w, e), (e, r)\} \in s.\mathsf{Order}$, if there is at least one thread that both e and r have been propagated to, then e is fully-propagated, e is not a same-address write, and e is not an unsatisfied same-address read.

Action: Let r be the read event and w the event for the write that r is being satisfied from. Let s be the current storage subsystem state. Then update r with the correct read value and record it as satisfied, update w to include r in the set of reads w satisfied and update the state with r and w.

And, if w is not fully-propagated delete r from the storage subsystem state's Events and Order and make w "inherit" r's edges:

$$\begin{aligned}
\mathsf{Events} &= s.\mathsf{Events} \backslash \{r\} \\
\mathsf{EProp}\ tid &= s.\mathsf{EProp}\ tid \backslash \{r\} \\
\mathsf{Order} &= (s.\mathsf{Order} \upharpoonright \mathsf{Events} \cup \{(e, w) | (e, r) \in s.\mathsf{Order}, e \neq w, (w, e) \notin s.\mathsf{Order}\})^+
\end{aligned}$$

Re-order condition: Let e' and e be events. The re-order condition reorder e' e holds if all of the following conditions hold:

- e' is not a strong memory barrier;
- e is not a strong memory barrier;
- e' is not a release write;
- if e is an acquire read, then e and e' are from different threads;
- if e is a write, then none of the reads it has satisfied is an acquire read from the same thread as e';
- if e' is an acquire read and e a release write, then they are from different threads;
- if e' is a write and e is a release write, then none of the reads that e' has satisfied is an acquire read from the same thread as e;
- if e' is a read or a write, then e is not a read or write to the same address;
- if e' is a `dmb ld` barrier, then e is not a read from the same thread as e';

- if e' is a `dmb ld` barrier and e is a write, then none of the reads e has satisfied is from the same thread as e';
- if e is a `dmb ld` barrier, then e' is not a read from the same thread as e;
- if e is a `dmb ld` barrier and e' is a write, then e and e' are from different threads and none of the reads e' has satisfied is from the same thread as e;
- if e' is a `dmb st` barrier and e a write, then e and e' are from different threads; and
- if e is a `dmb st` barrier, then e' is not a write.

4.2 NOP definition

As before, the definitions given here are prose descriptions closely following the formal Lem model.

NOP's state is the same as in POP:

$$\begin{aligned}\mathsf{Threads} &: \text{set thread-id}\\ \mathsf{Events} &: \text{set flowing-event}\\ \mathsf{Order} &: \text{set (flowing-event} \times \text{flowing-event)}\\ \mathsf{EProp} &: \text{thread-id} \mapsto \text{set flowing-event}\end{aligned}$$

The Events, Order, and EProp fields are used in a similar way as in POP. Without explicit event propagation the components Order and EProp are an under-approximation of those of a corresponding POP state: they describe a lower bound on the order and event propagations necessary in a POP trace that has the same sequence of accept-event and satisfy-read actions as the NOP trace leading to this state. The following defines NOP's initial state and transitions.

Initial state Initially, Threads contains the set of thread IDs, as determined by the input program. Events and Order are empty, and EProp is the pointwise empty map.

Accept transition The accept transition is the same as in POP. A new event e can be always accepted by a thread *tid*.

Action: When e is accepted it is inserted into Events and EProp and an edge (e', e) is added for any event e' that is already in EProp *tid*, unless the re-order condition holds.

$$\begin{aligned}\mathsf{Events} &= s.\mathsf{Events} \cup \{e\}\\ \mathsf{EProp}\ tid &= s.\mathsf{EProp}\ tid \cup \{e\} \quad \text{(otherwise unchanged)}\\ \mathsf{Order} &= (s.\mathsf{Order} \cup \{(e', e) | e' \in s.\mathsf{EProp}\ tid, \neg \mathsf{reorder}\ e'\ e\})^+\end{aligned}$$

There are two types of read transition, rf-segment and rf-memory, with the same intuition as the Flowing transitions: rf-segment is a transition that corresponds to a satisfy-read transition in Flowing in which the read event reads from a write event adjacent to it on the same buffer; rf-memory corresponds to a Flowing transition in which a read event reads in memory (rather than on the buffer).

While rf-segment requires that the read and the write can be made directly adjacent in Order, rf-memory allows reading "across" other events, including barriers, if the read and the write (and therefore the other events) are all propagated to all threads.

rf-segment This transition corresponds to a Flowing transition in which a read reads from a write on the same buffer, or a POP transition in which a read reads from a write that is directly Order-adjacent and where the write is not fully propagated yet. *Condition (rf-segment-cand):* The transition rf-segment is enabled in state s for a write event w and read-event r, if:

- w and r are to the same address,
- (r, w) is not in Order,
- r has not been satisfied yet,
- r and w are not a read acquire and a write release, and
- s'.Order is acyclic, where s' is the state NOP would reach after the transition.

Action (rf-segment-action): The action part of this transition consists of two steps:

1. All events e with (e, w) or (e, r) in s.Order are recorded as propagated to all threads that w or r have been propagated to, w is propagated to all threads r is propagated to, r is propagated to all threads w is propagated to; the edge (w, r) is added to Order.

$$
\begin{aligned}
B_{wr} &= \{e | (e, w) \in s.\mathsf{Order} \vee (e, r) \in s.\mathsf{Order}\} \\
T_{wr} &= \{tid | \{w, r\} \cap s.\mathsf{EProp}\ tid \neq \emptyset\} \\
\mathsf{EProp}\ tid &= s.\mathsf{EProp}\ tid \cup \{e | e \in B_{wr} \cup \{w, r\}, tid \in T_{wr}\} \quad \text{(otherwise unchanged)} \\
\mathsf{Order} &= (s.\mathsf{Order} \cup \{(w, r)\})^+
\end{aligned}
$$

2. In the resulting state a similar action is applied as in POP after satisfying a read:
   The read r is updated with its values and inserted into the read set of the write w, then deleted from Events, EProp, and Order is updated to remove r and to add edges so that w has all the incoming and outgoing edges of the write w and the read r together: add edges (e, w) for all e with $(e, r) \in s.\mathsf{Order}$ (where e is not w) and edges (w, e) for all edges $(r, e) \in s.\mathsf{Order}$. The latter kind of edges are already included in Order since it is transitively closed and includes the edge (w, r).

$$
\begin{aligned}
B_r &= \{e | (e, r) \in \mathsf{Order}\} \setminus \{w\} \\
\mathsf{Events} &= \mathsf{Events} \setminus \{r\} \\
\mathsf{EProp}\ tid &= \mathsf{EProp}\ tid \setminus \{r\} \quad \text{(otherwise unchanged)} \\
\mathsf{Order} &= ((\mathsf{Order} \downharpoonright \mathsf{Events}) \cup (B_r \times \{w\}))^+
\end{aligned}
$$

Note that this transition can only be done if there are no other events, such as barriers, in-between r and w in Order: Assume $\{(w, b), (b, r)\} \subseteq \mathsf{Order}$; then the state constructed by the rf-segment transition would have a cyclic Order relation in the state reached after the transition since there would be both an edge (w, b) and (b, w) in Order, the latter because of the edge (b, r).

rf-memory This transition corresponds to the Flowing transition in which a read reads from memory (rather than from a write on the same buffer), or a POP transition in which the write is fully propagated and the read and write event are not necessarily Order-adjacent. *Condition (rf-memory-cand):* An rf-memory transition is enabled in state s for a write event w and read event r if:

- r and w are to the same address,
- (r, w) is not in s.Order,
- r has not been satisfied yet, and
- s'.Order is acyclic, where s' is the state that NOP would reach after taking this transition.

Action (rf-memory-action): Add the following edges to Order:

- an edge (w, r) from the write to the read,
- edges (r, e) for all same-address writes e with $(w, e) \in$ Order,
- edges (e, w) for all same-address writes $e \neq w$ with $(e, r) \in$ Order,
- edges (e, w) for all events $e \in B$,
- edges (r, e) for all events $e \in A$,

where A and B are a partition of writes as described below, and propagate all events Order-before r including r and w to all threads and r is recorded as satisfied.

$$\begin{aligned}\mathsf{Order} = (&s.\mathsf{Order} \cup \{(w, r)\} \\ &\cup \{(r, e) | (w, e) \in s.\mathsf{Order}, \text{is-write } e, \text{address } e = \text{address } r\} \\ &\cup \{(e, w) | (e, r) \in s.\mathsf{Order}, \text{is-write } e, \text{address } e = \text{address } w, w \neq e\} \\ &\cup B \times \{w\} \\ &\cup \{r\} \times A)^+ \\ \mathsf{EProp}\ tid = &\ s.\mathsf{EProp}\ tid \cup \{e | (e, r) \in \mathsf{Order}\} \cup \{r\} \quad \text{(otherwise unchanged)}\end{aligned}$$

Here B *("order-before")* and A *("order-after")* are a non-deterministically chosen partition of all other writes to the same address as w (all legal possibilities enumerated).

This enumeration of all order-before/order-after partitions is necessary in order to guarantee that no other write to the same address can go Order-between w and r in a future state. In NOP, even if w and r are marked as propagated to all threads, this by itself does not by mean w and r have been ordered with respect to all other same-address events. It would be unsound to make w and r ordered before all other same-address events that are not yet Order-related with them when propagating w and r to all threads, as due to the lack of an event-propagate transition in NOP this would limit the possible coherence orders for writes that are never read from. Similar to POP and in contrast to the PLDI11 model, in NOP coherence is a derived property resulting from other storage subsystem actions: coherence is determined by the Order relation restricted to same-address writes. In order to allow for all coherence orders possible in POP, NOP allows all legal order-before/order-after partitions.

One other possible solution to this problem might be to replace the exhaustive enumeration of *order-before/order-after* partitions by distinguishing between two different kinds of edges in the Order component: the "normal" Order edges and edges additionally enforcing the adjacency of the

related events up to different-address events. The rf-memory transition could then Order-relate the read and the write using the latter and all transitions' preconditions would include checks to ensure that their post-condition preserves the read-write adjacency.

Another way to look at this, however, is that NOP not allowing for all possible orders of same-address write propagation would not be an issue: maybe the total coherence order of writes should not be considered a directly observable property of the operational model's state, but should instead be derivable as a partial order from an execution's reads-from relation: the coherence order could be made observable indirectly with an additional thread that has a read event for every write event in the program that observes the sequencing of writes. In this case, NOP would not need the order-before/order-after enumeration.

4.3 Proof

Assuming the POP version defined above, the NOP storage subsystem allows all behaviours allowed by finite POP executions. The proof uses a simulation argument: given a valid finite POP trace, a matching valid NOP trace can be constructed in which for each read the write it reads from is the same as in the POP trace and which reaches a corresponding state. The proof captures this correspondence of states in a simulation relation that relates a POP p and a NOP state n if n is less constrained than p, that is $p \gtrsim n$ if:

$$
\begin{aligned}
p.\mathsf{Threads} &= n.\mathsf{Threads} \\
p.\mathsf{Events} &= n.\mathsf{Events} \\
p.\mathsf{Order} &\supseteq n.\mathsf{Order} \\
p.\mathsf{EProp} &\supseteq n.\mathsf{EProp} \qquad \textit{(pointwise)}
\end{aligned}
$$

Note that since the coherence relation induced by a POP or a NOP trace is derived from the final state's Order relation, the simulation argument also covers coherence: by $p.\mathsf{Order} \supseteq n.\mathsf{Order}$ for the final POP state p and final NOP state n the coherence relation induced by the POP trace is also possible in the NOP trace — at the end of a NOP trace the coherence relation might be partial on same-address writes and by this property a subset of POP's coherence relation in the simulated trace; the possible coherence orders of NOP are the linearisations of same-address writes compatible with this partial order and therefore include the POP coherence relation.

The proof shows by induction on the trace length that given any POP trace, a valid corresponding NOP trace can be constructed: the initial states of NOP and POP are related by the simulation relation; and there exists a correspondence of POP and NOP transitions such that given a POP state s with outgoing transition t and a NOP state s' where s and s' are related according to the simulation relation, the NOP transition t' corresponding to t is enabled in s and preserves the simulation relation.

As discovered in earlier failed attempts at the proof, however, the simulation relation as defined above is insufficient — it is too liberal. In order to be strong enough with respect to POP (experimentally excluding all POP-forbidden behaviours) NOP's construction makes the propagation sets

Order-down-closed — whenever an event e is recorded as propagated to some thread *tid*, so are all events e' with $(e', e) \in \mathsf{Order}$. For the proof of soundness it has to be shown that it is safe for NOP to preserve this invariant: POP does not have this *subset property*. (In fact, there are examples that POP allows only because it does not have the subset property. See Figure B.1 in the appendix for an example.) As it turns out, however, all Order edges in NOP correspond to POP Order edges for which the subset property holds in the corresponding POP state of the simulation. The proof's induction step requires strengthening the induction hypothesis by capturing this property in the simulation relation $\sim$. Define $s \sim s'$ to hold, if:

$$s \gtrsim s' \quad \wedge \quad \forall (e', e) \in s'.\mathsf{Order}.\ e \in s.\mathsf{EProp}\ tid \rightarrow e' \in s.\mathsf{EProp}\ tid$$

With the addition of the subset property to the simulation relation it can be shown that for any POP trace there exists a corresponding NOP trace preserving the simulation relation $\sim$. For the proof details, see Section B.2 in the appendix.

4.4 Performance

Running the executable POP model on example tests, much of the non-determinism in the storage subsystem behaviour is in the order in which the model propagates different events to different threads, and the interleaving of such event-propagate transitions with other transition of the storage subsystem. This factor in the non-determinism is especially important in examples of more than two threads. In certain POP states, the sequencing of these transitions can lead to observably different final outcomes. In other cases the sequencing does not matter: for example, the order in which two different plain writes propagate to some thread may not matter.

NOP abstracting from the event propagation eliminates some of this non-determinism, and therefore improves on the performance of POP in such cases where the ordering of event-propagation transitions does not matter. A slightly different version of NOP than presented here, for instance, leads to the following performance results compared to POP when run on the around 2200 non-mixed-size litmus tests finishing in reasonable time: overall the accumulated runtime in POP for those tests is more than 12 times that of the runtime for NOP; on average, for those tests NOP is around 56 times faster than POP.

Chapter 5

Mixed-size Sequential Consistency

Whereas previous work in the relaxed memory concurrency literature has mostly assumed programs where memory accesses are all aligned and of the same size, Flur et al. [52] consider the concurrency behaviour of Power and ARMv8 programs with mixed-size memory accesses. The paper discusses the fact that, unlike the non-mixed-size case, barriers cannot restore Sequential Consistency [76] in Power and non-MCA ARMv8. This chapter proposes a weaker notion of mixed-size Sequential Consistency that can indeed be obtained in mixed-size Power and ARMv8 programs with barriers in-between all memory accesses. The following gives a characterisation of the behaviour of such programs and a proof that this is a sound characterisation for non-MCA ARMv8. Moreover, the following shows that non-ARMv8 programs where all loads are load acquire and all stores are store release also respect the stronger notion of SC.[1] The observation of the fact that barriers do not restore Sequential Consistency in Power and ARMv8 and the accompanying example are due to Shaked Flur.

5.1 Barriers do not recover SC for mixed-size ARM or Power

A standard result for relaxed memory models, and a property that architectures have normally been thought to intend and to guarantee, is that inserting enough barriers in a concurrent program restores Sequentially Consistent behaviour. (An exception to this is Itanium.[2]) Perhaps surprisingly, in the mixed-size setting neither non-MCA ARMv8 nor Power have this property, as the example of Figure 5.1 shows: there is no way to totally order these four events with each read reading each byte from the most recent write to that byte.

Whereas in the previous diagrams the events only have address and value, in the mixed size case every event has to be labelled with a *footprint*, comprising an address and a size in bytes, and reads-from edges are labelled with the relevant *slices* of the write: the sub-footprints of the write being read in this edge. The example assumes big-endian format, meaning that the most significant bytes of numbers are stored at lower addresses. In this execution there are two writes, a a write to x of 8 bytes (W x/8 denoting an 8-byte write to the 64-bit aligned address x) and b of 4 bytes to x. Thread 2 reads all 8 bytes of x twice. Whereas the first read c reads the first four bytes of x from b and the remaining four bytes from the initial write (indicated by rf[x+4/4=0]), the second read d observes the final value of x, reading the first four bytes of x from b and the

[1] The results of this chapter have been published in Flur et al. [52]; the following text and the proof text in the appendix adapts that of this paper [52].

[2] The Intel Itanium specification [65] defines a non-multi-copy-atomic model where the strongest barrier is not sufficient to regain multi-copy atomicity, for normal accesses, and hence insufficient to regain SC for them; regaining SC requires the Itanium store release and load acquire instructions. It is unclear whether Itanium implementations have actually exploited that weakness.

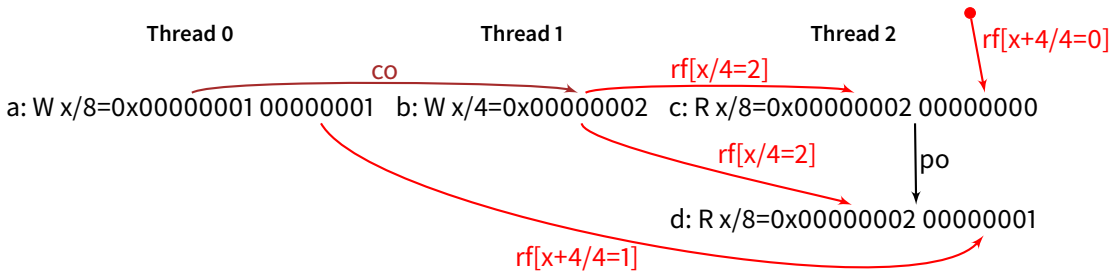


Figure 5.1: Test CO-MIXED-2b

remaining four bytes of x from a with a coherence-ordered before b.

This execution is architecturally allowed on both non-MCA ARMv8 and Power (but not MCA ARMv8) and observable on current Power implementations. Adding a barrier between these two reads makes no difference in the models, and the result remains observable with a `sync` barrier on Power 7 (test CO-MIXED-2b-sync, 48k/2.2G)[3]. Since ARMv8 implementations are multicopy atomic and this execution requires non-multicopy-atomic behaviour — it requires the possibility of propagating b to Thread 2 without propagating b to Thread 0 — it is not expected to be observable on ARMv8 implementations.

In Flowing, for example, this behaviour can be explained as follows. In a topology where Thread 2 and Thread 3 are closer to one another than to Thread 1, b can propagate to Thread 3 and c can propagate to Thread 2, both before b and c become visible to Thread 1 (see Figure 5.2 below). The read c can now be satisfied — half of it from b, the other half from the initial write — before a is propagated to Thread 2 and Thread 3 and becomes coherence-before b. The example works similarly for Power.

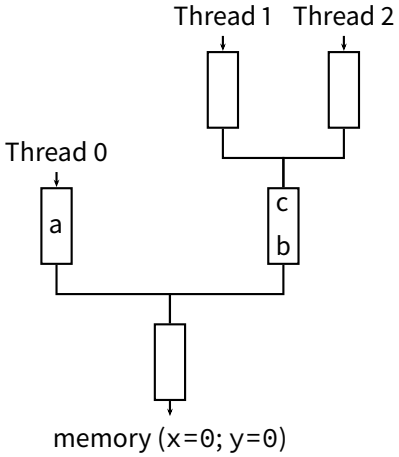


Figure 5.2: Flowing state in execution allowing the non-SC behaviour

[3]These observations are from an IBM POWER 730 server, with a POWER 7 CPU, 48 hardware threads

5.2 Characterising the behaviour of fully barriered programs

The following gives a characterisation of what guarantees Power and ARMv8 do give when inserting strong barriers (`sync` or `dmb sy`) between any two instructions in program order. For conciseness, these programs will be called "fully barriered". For simplicity the following only deals with the case of programs that have no misaligned memory accesses, which would also involve the store and load splitting of Section 3.4.6, Section 3.4.6. Since Flowing and POP do not handle the load-pair instruction, the proof also ignores these.

As a first attempt at an axiomatic characterisation of the hardware behaviour of fully barriered mixed-size programs (without misaligned accesses) consider the following, henceforth called Bytewise SC (BSC). Partition all read events and write events into *subevents* (also *subreads* and *subwrites*) of the smallest size supported by the architecture — for Power and ARM this is one byte — and record which subevents were generated by the same event in an irreflexive, symmetric relation si. In general si should only relate events of the same *single-copy-atomic event*, defined below. However, all aligned accessess considered here are single-copy-atomic in ARM and Power. Now define a candidate execution to consist of the subevents (subreads of reads, and subwrites of writes) and barriers, with the usual components po, rf, and co — but as per-byte relations, and with po lifting program order to a relation on the subevents (subreads and subwrites) and barriers. BSC requires that coherence be compatible with si in the following sense: $w_i \xrightarrow{\mathsf{co}} v_j \implies w_{i'} \xrightarrow{\mathsf{co}} v_{j'}$ whenever $\{(w_i, w_{i'}), (v_j, v_{j'})\} \subseteq \mathsf{si}$ and $w_{i'}$ and $v_{j'}$ have the same address. Then call a candidate execution BSC if there is a total order on the subreads and subwrites that agrees with po, co, and rf: the total order includes po lifted to subevents (restricted to subreads and subwrites), a subread has the value of the most recent preceding subwrite to the same byte-address in the order, and co is the restriction of the total order to same-address subwrites.

This can be shown to admit all behaviour of fully barriered mixed-size Power and ARM programs without misaligned accesses. The following gives the proof for non-MCA ARMv8; the proof for Power can be found in the POPL17 paper's supplementary material [52]. Figure 5.1 is witnessed by the subevent order $\mathsf{c}_7 \to \mathsf{a}_7 \to \mathsf{a}_3 \xrightarrow{\mathsf{co}} \mathsf{b}_3 \xrightarrow{\mathsf{rf}} \mathsf{c}_3$, suitably extended for the other subevents. Here x_i denotes the ith byte-sized subevent of an event x, e.g. c_3 is the subwrite 0x02 of c. However, this gives too weak a guarantee to be suitable for programming and is weaker than Power and ARM hardware for fully barriered programs; for example, the undesirable behaviour of the test Figure 5.3 below is allowed in BSC. Here the read c is satisfied from a combination of the writes a and b, while in ARM and Power one would want it to be satisfied completely by either only a or only b, whichever wins the race.

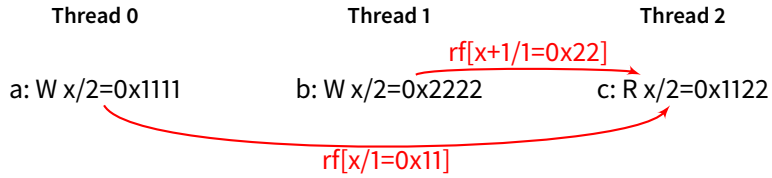


Figure 5.3: Test SCA-1

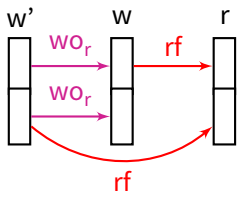


Figure 5.4: Violation of single-copy-atomicity. Each box represents a byte-sized subevent.

This execution violates the principle of single-copy-atomicity: for any read r there must be a total order wo_r over the writes it reads from such that each subread of r reads from the wo_r-maximal subwrite. This can be more formally defined as follows: an execution is single-copy-atomic if for each read r there exists a total order over all same-address subwrites wo_r compatible with si (no cycles of the form $\mathsf{wo}_r; \mathsf{si}; \mathsf{wo}_r; \mathsf{si}$) and such that there are no cycles of the form $\mathsf{rf}_r; \mathsf{si}; \mathsf{rf}_r^{-1}; \mathsf{wo}_r^{+}; \mathsf{si}$, where rf_r is rf restricted to r's subreads. Figure 5.4 illustrates a violation of single-copy-atomicity defined in this way.

The above definition allows the ordering of writes wo_r to be an arbitrary order (subject to the above constraints) and different for each read event r. In most cases, including Power and ARM, there is a notion of coherence that requires a global ordering of overlapping writes. In these cases single-copy-atomicity can be specialised to the following, where wo_r always coincides with coherence: $\mathsf{rf}; \mathsf{si}; \mathsf{rf}^{-1}; \mathsf{co}; \mathsf{si} = \mathsf{rf}; \mathsf{si}; \mathsf{fr}; \mathsf{si}$ must be acyclic (c.f. [20, B2.6.2]). Now define BSC+SCA as BSC with the latter single-copy-atomicity axiom added.

Theorem 1. *A fully barriered Power program with no misaligned accesses has BSC+SCA behaviour in the PLDI11 model. A fully barriered non-MCA ARMv8 program with no misaligned accesses has BSC+SCA behaviour in POP.*

The proof takes an arbitrary trace tr of the concurrency model, and constructs a total order on the byte-sized subevents that matches program order, coherence, and reads-from (and from-reads) of the trace. For ARM the proof uses a lemma that states that any two writes that are related by paths of coherence, reads-from, from-reads, and program-order edges are already related in the same way in order-constraints of the final state of tr. For Power the key result is that for any path in the graph of events with coherence, reads-from, from-reads, and program-order edges that ends with a program-order edge (e, e'), in the state in tr when e' is accepted into the storage subsystem all reads on the path have been satisfied and all writes from the path have been propagated to all threads. For the full proof for non-MCA ARMv8 see Chapter C in the appendix; the proof for Power is omitted but can be found in the POPL17 paper's supplementary material [52].

5.3 Recovering SC on ARM

If all memory accesses are aligned and have the same size, any complete execution totally orders same-address events (except for read-read pairs) in terms of the relations coherence, reads-from,

and from-reads. In the example of CO-MIXED-2b, however, there is no such total order: the read c observes a state where a and b are not ordered yet. What is necessary to prevent the behaviour of CO-MIXED-2b is multicopy atomicity: c must not be satisfied before b is visible to all threads and thus ordered with a. In non-ARMv8 this is exactly the behaviour that acquire reads in combination with release writes provide: replacing b with a write release and c with a read acquire in the test forbids the non-SC behaviour, because c can only be satisfied from b when both are propagated to all threads, at which point a and b are ordered: either a is ordered before b and c returns 0x0000000200000001, or b is ordered before a, but then it is $b \xrightarrow{co} a$. This gives an intuition for the following theorem.

Theorem 2. *A non-MCA ARMv8 program whose only reads are acquire reads and whose only writes are release writes and that has no misaligned memory accesses has Sequentially Consistent behaviour in POP.*

The intuition behind this is that if all memory accesses are release/acquire accesses, the thread semantics is forced to behave sequentially, the storage subsystem keeps all release/acquire events in the order they were accepted into storage, and multicopy atomicity ensures that the reads-from relation agrees with some total order on same-address events.

The proof, in Chapter C in the appendix, constructs a total order on the reads and writes of a given POP trace that matches program order, coherence, and reads-from (and from-reads). The key point of the proof is that at the point when a read is partially satisfied it has to be fully propagated, and therefore all writes it will read from fully propagated and totally ordered by POP's order-constraints.

Chapter 6

Multicopy atomic ARMv8 models

This chapter gives an overview of the changes introduced in the revised ARMv8 architecture. The main change, the move to a multicopy atomic semantics, greatly simplifies the concurrency behaviour, enabling a much simplified operational and axiomatic concurrency model. This chapter defines this simplified operational model, based on Flowing [51, 52], and also presents ARM's official axiomatic concurrency model. The contents of this chapter were presented in Pulte et al. [104] and this text copies and adapts the text of that paper. The operational and axiomatic models were co-developed; the operational model was developed jointly with Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell, with the author and Shaked Flur as main contributors; the axiomatic model was developed principally by Will Deacon and his colleagues at ARM. Our extended collaboration with ARM, especially the work of Shaked Flur, partly motivated the changes in the revised concurrency architecture.

6.1 Background of the architectural changes

ARMv7 defined a non-multicopy-atomic model, that was thought to be broadly similar to that of Power [13, 110, 111, 31]. The early ARMv8 architecture, announced in October 2011, had a broadly similar non-multicopy-atomic concurrency model, but introduced new ordering instructions [51]: where ARMv7 had `dmb sy` and `dmb st` barriers, ARMv8 added `dmb ld` barriers and load acquire and store release instructions.

The strong `dmb sy` barrier provides ordering between memory accesses of any kind program-order-before and after it, as well as strong cumulativity properties: guaranteeing the ordering of writes that have been propagated to the `dmb`'s thread before committing the barrier, with respect to reads and writes ordered after the barrier (program-order after or ordered via certain dependencies). In contrast to that, the exact cumulativity properties of `dmb st` and `dmb ld` in the non-MCA ARMv8 architecture were less clear [51]. Release/acquire instructions, in addition to providing certain thread-local ordering and cumulativity properties in non-MCA ARMv8, were intended to provide multicopy atomicity under particular conditions. The exact semantics for the interaction of store release and load acquire instructions with non-release/acquire instructions was also not clear [51].

The goal of specifying the concurrency semantics of the ARMv8 architecture led to the extended collaboration with ARM. In the process, in work principally by Shaked Flur, the Flowing and POP operational models were developed, with the aim to make the architecturally intended behaviour precise, in abstractly micro-architectural models based on detailed discussion with ARM staff. This operational modelling clarified open questions about the concurrency semantics, but also raised new issues, generating new questions of memory model design that were not

previously considered. Section 3 of [104] (principally the work of Shaked Flur) gives an account of some of the design choices and consequences for the operational modelling that illustrate the difficulties arising from the concurrency model design of the non-MCA ARMv8 architecture. These originate in part from the architecture's intention of defining a concurrency semantics that is as liberal as possible without concrete implementations that exploit those relaxations for guidance, while at the same time providing the architecturally intended guarantees — such as certain ordering, coherence, and atomicity guarantees, multicopy atomicity for release/acquire instructions in an otherwise non-multicopy-atomic model, the guarantees necessary for the C/C++ 11 compilation scheme, and others. The resulting Flowing and POP models expose the complexity of the architectural intention of the non-MCA architecture.

In parallel to the later stages of the ongoing operational formalisation efforts, ARM internally developed an axiomatic specification for the non-MCA ARMv8 architecture. The operational and axiomatic models, however, experimentally mismatched, and it was unclear how they related or could be made equivalent. Moreover, in the axiomatic model, handling certain "detour" examples — examples in which the interaction of one thread with another implies a certain transitive ordering within the former thread [17] — proved difficult. According to Will Deacon: "Capturing such 'big detour' examples in axiomatic models required the definition of the relations in the model to be mutually recursive. This was the straw that broke the camel's back in terms of readability among industry colleagues: . . . " [104, Section 3].

Based on feedback from ARM partners, informed by our ongoing collaboration with ARM and the complexity of the Flowing and POP operational models, and based on the simplicity of Will Deacon's draft axiomatic multicopy atomic model, ARM decided the flexibility and possible optimisations that non-multicopy-atomicity enable did not outweigh the cost of complexity of the non-MCA concurrency architecture. The fact that existing ARMv8 implementations had multicopy atomic behaviour (as previously indicated by hardware test data for some implementations [93, 17, 51, 52]) allowed ARM's decision to strengthen the ARMv8 concurrency architecture to be multicopy atomic, in early 2017. The revision of the architecture documentation that also for the first time covered version 8.2 of the architecture [22] included the shift to multicopy atomicity, "backporting" the change to existing ARMv8 implementations.

The following discusses the changes of the revised architecture.

6.2 Architectural changes in ARMv8

The most notable change in the revised ARMv8 architecture is the switch to an MCA model: when a write is propagated from one thread to another, it has to be propagated to all other threads as well. This makes tests such as IRIW+addrs (see Figure 6.1) forbidden. As discussed earlier, for this behaviour to be allowed it has to be possible to propagate a to Thread 1 without immediately also propagating it to Thread 3 and conversely, to propagate d to Thread 3 without propagating it to Thread 1. Multicopy atomicity forbids this and means this execution is forbidden in the revised architecture.

The remaining relaxed-memory effects are all due to thread-local out-of-order and speculative

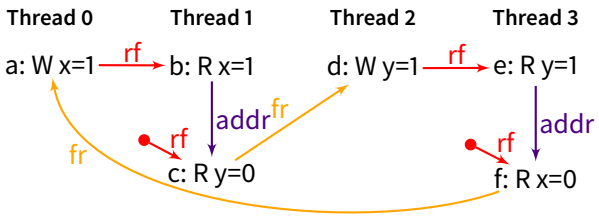


Figure 6.1: IRIW+addrs

execution and thread-local buffering. Writes can still be visible to program-order-later reads on the same thread before becoming visible to other threads (even on speculative paths, as in the PPOCA previously shown in Figure 1.5).

The previous ARMv8 concurrency architecture also allowed *write subsumption*: a write was allowed to propagate to memory even if a program-order-earlier write to the same location had not propagated to memory yet, thereby "subsuming" it, allowing, for example, the behaviour of the LB+data+data-wsi test of Figure 6.3. In LB+datas (Figure 6.2) Thread 0 reads x and writes the value read to y, Thread 1 symmetrically, with locations x and y swapped. This test was forbidden in the previous architecture and remains forbidden now. The variation LB+data+data-wsi adds a second, non-data-dependent, write to x on Thread 1 that Thread 0's event a reads from. This latter behaviour was allowed in the previous architecture where the write e to x on Thread 1 was allowed to propagate into memory early, before its same-address program-order predecessor propagated. (As discussed earlier, the version of POP previously defined in Section 3.4, however, for simplicity does not permit this behaviour.) The revised ARMv8 memory model forbids write

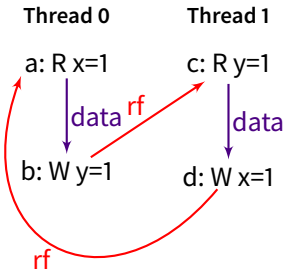


Figure 6.2: LB+datas

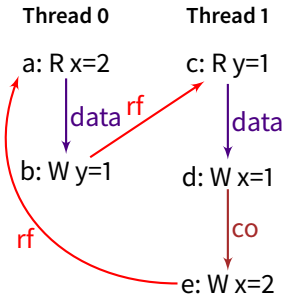


Figure 6.3: LB+data+data-wsi

subsumption, and thereby creates order from a data-dependent write to program-order successor writes to the same address, forbidding the behaviour of LB+data+data-wsi.

The last change is concerned with the definition of dependencies (read-to-read address and control+isb/isync dependencies[1], and read-to-write address, data, and control dependencies). Historically other architectures, e.g. IBM Power, have explicitly respected all syntactic dependencies. Previous versions of the ARM architecture text introduced notions of "true" and "false" dependencies, aiming to require processors only to preserve "true" dependencies, to allow optimi-

[1] a control dependency followed by an `isb` (ARMv8)/`isync` (Power) instruction barrier

sations in value computations. For example, in a computation such as `AND x1 x2 x3`, in which `x1` is assigned the value of the bitwise AND of the values of registers `x2` and `x3`, if `x3` is known to be holding value 0 the syntactic register dependency from `x2` to `x1` was meant to be treated as a false dependency and not create memory model ordering, since a CPU might be able to determine that the value of `x1` is unaffected by that of `x2`. The architecture specified this with wording such as the following: *"A False Register data dependency is a Register data dependency where no register in the system holds a variable for which a change of the first data value causes a change of the second data value."* [21, B2-92]. It is unclear how this could be made precise in a satisfactory way, as "causes a change" itself involves the whole concurrency and non-deterministic semantics. Recent work on C/C++ concurrency illustrates the difficulties of defining envelopes around such optimisations [32, 100, 71]. The revised ARMv8 architecture makes no such distinction.

The changes in the revised ARMv8 architecture greatly simplify the concurrency behaviour, most importantly due to the shift to a multicopy atomic concurrency model. The architectural changes enable the simple official axiomatic concurrency model presented later (simple relative to axiomatic models for non-multicopy-atomic architectures, that is), but also a much simplified operational concurrency model. For a good understanding of the architectural concurrency semantics it is desirable to have both presentations, axiomatic and operational, and both have advantages and disadvantages depending on the use case.

The next section describes the simplified operational model for the revised ARMv8 architecture, the subsequent one the ARMv8-axiomatic model.

6.3 A simpler operational model

In Flowing, the requirement of multicopy atomicity corresponds to a flat storage subsystem topology: a topology where each thread has its own buffer and all thread buffers are connected with a single buffer below before main memory, as shown in Figure 6.4. Thus Flowing, with some adaptations to accommodate for the other changes of the revised architecture, is a suitable model also for the revised architecture. Since multicopy atomicity is a big conceptual simplification, however, the revised ARMv8's concurrency behaviour can be described more simply than this. This section describes this simplified operational model.[2]

The idea underlying the simplified operational model, called *Flat*, is that in the multicopy atomic ARMv8 architecture the propagation of events is simple enough that it can be explained without a storage subsystem model with explicit re-ordering and "flowing" of events. Analysing the re-order condition of the Flowing storage subsystem to see which event re-orderings can occur in a Flat topology shows that most of these are subsumed by the out-of-order execution within the threads. Thus, the concurrency behaviour of Flowing with a flat topology can be explained in terms of slightly adapted Flowing/POP thread subsystems in parallel composition above a flat memory.

[2] As before, the model was developed principally together with Shaked Flur, in collaboration with Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell, based on Flur et al. [51, 52], in turn based on Sarkar et al. [110, 111] and Gray et al. [60]. The description closely follows Pulte et al. [104]. Since the thread subsystem adapts of that of Flur et al. [51, 52] the description closely follows that of those papers. We thank Alan Stern for interesting discussions.

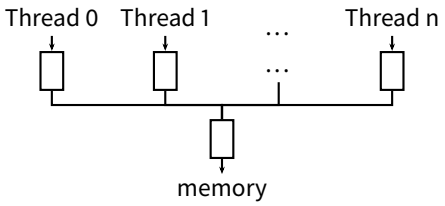


Figure 6.4: Flat flowing topology

Motivation The result is a simpler explanation of the concurrency behaviour. In Flowing, the concurrency behaviour is explained in terms of the synchronising transition system of the two main components: the thread subsystem and the storage subsystem. Both of these are complicated "pieces". The thread subsystem enumerates the possible transitions of the thread's instructions, allowing them to execute in many steps per instruction, out-of-order with respect to preceding instructions, and speculatively with respect to preceding conditional and computed branches. The storage subsystem receives requests from this thread subsystem, for reading and writing memory, and for barriers; inside the storage subsystem these requests (which are potentially received out-of-program-order) are again subject to possible re-ordering and out-of-order propagation (from their thread to the buffer shared between all threads).

Flat completely removes the complexity of the storage subsystem component: while the thread subsystem component is still a complicated model, the concurrency behaviour in Flat can now be understood just directly in terms of the actions of the instructions in the threads, without having to consider the effects of the out-of-order propagation and re-ordering in the storage subsystem. In Flowing with flat topology there is a large overlap between the relaxed behaviours in the threads and the relaxed behaviours inside the storage subsystem, leading to the situation where many concurrency tests can be explained in a number of different ways. Flat removes this overlap between the thread and the storage subsystem.

One example of a behaviour that in Flowing can be explained by the re-ordering inside the storage subsystem but also by the out-of-order execution in the threads is that of the `MP` test, shown in the following.

6.3.1 Examples

Simple first example We now revisit the examples previously used in illustrating Flowing and POP, but now with Flat. First, we consider the `MP` test (see Figure 6.5; as before, assuming on both threads `X1` holds the address of x and `X3` that of y).

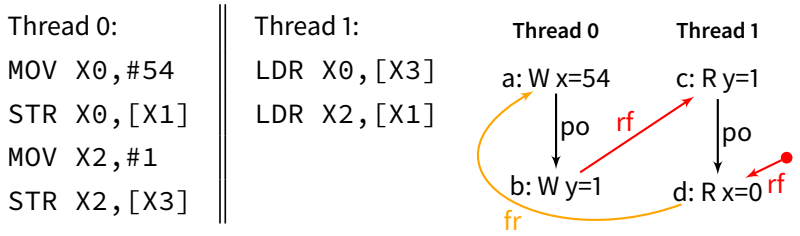


Figure 6.5

Flowing allowed this behaviour by propagating/issuing a and b, and c and d into the storage subsystem in program-order, but then allowing them to re-order in the storage subsystem. Another way the same behaviour is allowed in Flowing is with the out-of-order execution of instructions in the threads; and that is how Flat explains the behaviour, but with the simpler flat memory state:

1. Execute the `MOV` instructions on Thread 0, saving the values of the stores in the registers `X0` and `X2`.
2. Execute the store to y (in a sequence of multiple transitions): read its registers, and initiate, instantiate, and commit it, and propagate b into memory, updating the flat memory state to $y = 1$.
3. Execute the load to y (in multiple steps): read its registers and initiate the load; then satisfy it from memory, reading $y = 1$ from the memory state; then finish the load.
4. Similarly, execute the load to x, reading $x = 0$ in memory (since a has not propagated yet).
5. Finally, execute the store to x on Thread 0 and propagate a to memory, updating the memory state to $x = 54$.

Hence, in this execution, Flat explains the behaviour by executing the store to y early. Another way Flat allows this behaviour is by allowing the load of d to execute early: d can be satisfied from memory — reading $x = 0$ — before c, and before a or b have propagated to memory.

Placing `dmb sy` barriers on both threads prevents the behaviour and both these executions, simply by constraining a and b, and c and d to execute in program order.

Multicopy-atomic storage subsystem Flat's storage subsystem is multicopy-atomic, and hence disallows the previous IRIW+addrs example (in Figure 6.6), which was allowed in the non-multicopy-atomic Flowing and POP storage subsystem models for the early ARMv8 architecture.

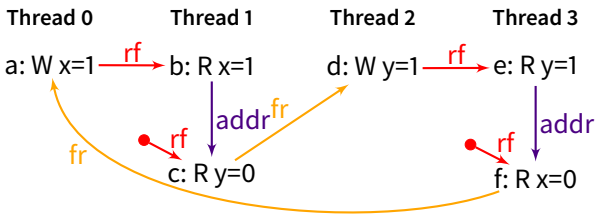


Figure 6.6: IRIW+addrs

We first show how Flowing with flat topology achieves multicopy atomicity and forbids this

example. Consider, for instance, the following Flowing execution (omitting some irrelevant thread subsystem details):

- Commit and propagate a and d.
- Issue b and e.
- Now either b has to first read from a, or e from d. Assume the former. (The latter case is symmetric.) Then a has to propagate to a common buffer with b. So flow a down.
- Flow b down. Then the Flowing storage subsystem state is as shown in Figure 6.7a:

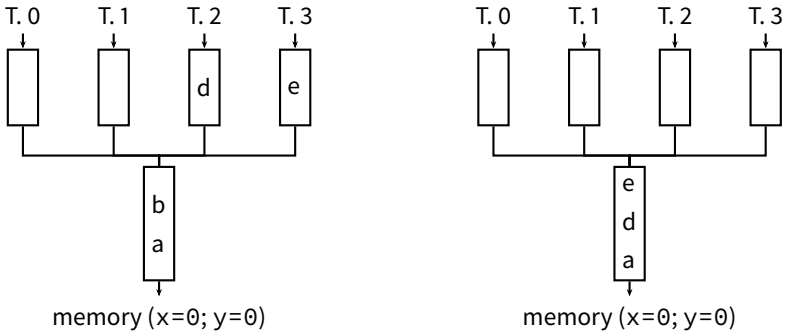


Figure 6.7

- Now it is possible for b to read from a and resolve the address dependency of c.
- Issue c, and flow it down to the buffer shared by all threads.
- To flow to memory, c has to pass a. The events c and a can re-order, since they have disjoint footprints, and so c can subsequently flow down and read the initial $y = 0$.
- For e to read from d, d must flow down, and subsequently e flow down, leading to the state shown in Figure 6.7b.
- Now e can read from d and resolve the address dependency of f.
- f can issue and flow down, but in this state, a is already propagated to the common buffer. And since f cannot re-order with a it must now read from a, leading to a different outcome than in Figure 6.6.

As seen in the above example, multicopy atomicity is given in Flowing with flat topology by the fact that the only way one thread can make a write visible to another thread is by propagating it to the (only) common buffer, above memory, that is visible to all threads.

In Flat the multicopy-atomicity is captured by the flat memory state. Replaying the analogous execution in Flat is as follows:

- As before, either b first reads from a or e first reads from d. In the former case (the latter is symmetrical), start by committing and propagating a into memory, updating x to value 1.
- Satisfy b in memory, resolving c's address dependency.
- Satisfy c in memory, reading the initial $y = 0$.
- Commit and propagate d.
- Satisfy e in memory, resolving f's address dependency.
- Now satisfy f. But since memory was updated to $x = 1$ by a, the read f is unable to

read $x = 0$.

Thread 0:	Thread 1:
`MOV W0,#54`	`LDR W0,[X3]`
`STR W0,[X1]`	`EOR W4,W0,W0`
`DMB SY`	`LDR W2,[X1,W4,SXTW]`
`MOV W2,#1`	`LDR X2,[X1]`
`STR W2,[X3]`	

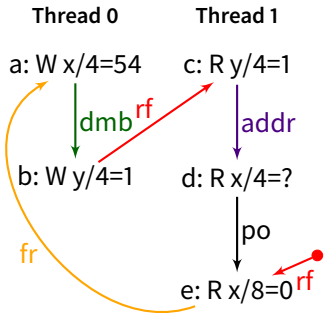


Figure 6.8

Thread subsystem restarts The Flat thread subsystem is almost the same as Flowing and POP's, and hence Flat behaves on this example just as seen in Section 3.3.3. Assume, as before, Thread 0 and Thread 1's registers `X0` and `X3` hold the addresses of the locations x and y, respectively.

Like Flowing, Flat forbids the above behaviour. Due to the barrier between them, a and b execute in program order. So whenever Thread 1 "sees" b, it must also see a. Hence, in order to allow the behaviour, e would have to satisfy before c reads from b. Consider the following execution:

1. Execute the load of e: read the registers, initiate the load, and satisfy e in memory with $x = 0$.
2. Execute the store of a and propagate a into memory.
3. Execute the `dmb sy` (which does not change the storage subsystem state).
4. Execute the store of b and propagate b into memory.
5. Satisfy c in memory, from b, resolving the address dependency of d.
6. Satisfy d. Since a has already propagated to memory, d must read from a.
7. Coherence violation: e is program-order-after d, and the part of e overlapping d was satisfied from a different write than that part of d, and this write is not program-order-after d. Therefore the load of e is restarted.
8. When e is satisfied in memory again it reads the first four bytes from a and the other four bytes from the initial write to x.

So initially e is satisfied early, but then Flat restarts e's load due to the resulting coherence violation, preventing the test outcome.

Thread-internal forwarding Recall the PPOCA test [110], in Figure 6.9. Assume on both threads register `X1` holds address x, `X3` address y, and `X7` address z. Flat allows the execution where c reads $y = 1$, f reads from e, and d reads $x = 0$: (just as before for Flowing)

1. Speculatively fetch the instructions of e, f, and d.
2. Execute the store of e partially: do its register reads, and announce its address and value.
3. The load of f can read its registers and announce its address.
4. e's store cannot commit and propagate e to memory yet, because of the control dependency.

Thread 0:	Thread 1:
MOV W0,#54	LDR W0,[X3]
STR W0,[X1]	CBZ W0,end
DMB SY	MOV W2,#1
MOV W2,#1	STR W2,[X7]
STR W2,[X3]	LDR W4,[X7]
	EOR W5,W4,W4
	LDR W6,[X1,W5,SXTW]
	end:

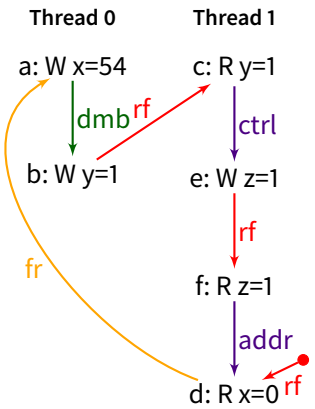


Figure 6.9: PPOCA

But it can thread-locally *forward* the write e to f. This satisfies f resolves the address dependency of d.

5. Then d can announce its address and satisfy in memory, from the initial write to x.
6. Thread 0 propagates a into memory, commits the barrier, and propagates b into memory.
7. Thread 1 satisfies c in memory, from b. It then writes c's return value to `W0`.
8. `W0` is non-zero, so the conditional branch does not branch, and the instructions in the instruction tree starting from e are not discarded, allowing the execution.

6.3.2 Slight relaxation of thread subsystem

Flat's thread subsystem is almost the same as that of Flowing and POP. The main change in Flat's thread subsystem compared to Flowing and POP is necessary to handle some re-ordering with respect to certain barriers that Flowing and POP allow: in the Flowing and POP thread subsystem barriers commit in program order, but the storage subsystem allows some re-ordering with respect to these barriers; Flat does not have a storage subsystem that allows such re-ordering and instead makes up for this by allowing those barriers to execute out-of-order in the thread subsystem. Consider, for instance, the example shown in Figure 6.10:

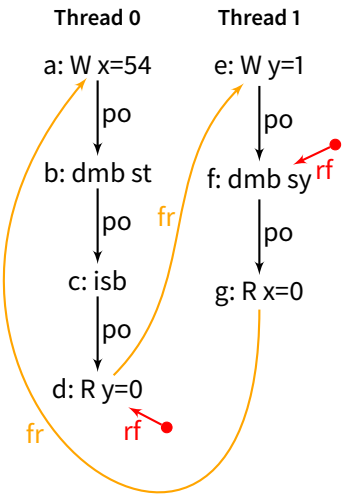


Figure 6.10

This test is a variant of the store buffering (SB) litmus test. Thread 0 writes 54 to x with a, and subsequently reads y with d. The store of a and load of d are separated by a `dmb st` and an `isb` barrier. Thread 1 writes $y = 1$ and, after a `dmb sy` barrier f, reads x with g. The execution in Figure 6.10 is one where both d and g read 0 from the initial writes to y and x, respectively.

This execution is allowed in the MCA ARMv8 architecture (also in non-MCA ARMv8). One allowing execution in Flowing with flat topology is as follows:

- Commit and propagate a, placing a at the top of Thread 0's Flowing buffer.
- Commit and finish b, placing it at the top of Thread 0's Flowing buffer.
- Commit and finish c (no effect on the storage state).
- Issue d, placing d at the top of Thread 0's Flowing buffer.
- Commit and propagate e (similar to before, placing e at the top of Thread 1's buffer).
- Commit f.
- Issue g. After these steps, the Flowing storage subsystem state is as shown in Figure 6.11.
- Since the re-order condition holds for d and b, they can re-order, swapping their positions in the buffer.
- Similarly, since d and a are to different footprints they can re-order, placing d at the bottom of Thread 0's buffer.
- Now d can flow down and read the initial write in memory.
- Events e, f, and g flow down and into memory, g reading from the initial write to x.
- Finally, a and b flow down.

The key point here is that in Flowing there is some thread-local ordering between a and d, due to the chain of barriers b and c, but the storage subsystem still allows their re-ordering: d can only issue when c is finished and c can only finish when b is finished; and b in turn can only finish once a is finished. Despite this ordering in the thread subsystem, the relaxed behaviour of this test outcome is still allowed because in the storage subsystem the chain of b and c does not create ordering from a to d: the barrier c does not enter the storage subsystem, and d can re-order with b and subsequently a.

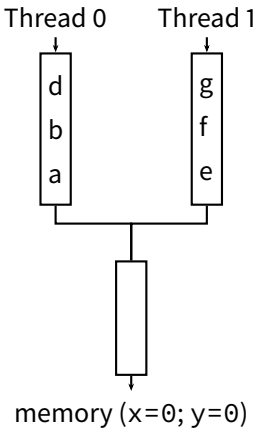


Figure 6.11

If Flat adopted the Flowing and POP thread subsystem with no changes, the local ordering induced by b and c would mean d could only be satisfied after a is propagated to memory, preventing the execution, since Flat has no storage subsystem re-ordering mechanism that would allow the events of a and d to propagate to Thread 1 out-of-order. Instead, Flat's thread subsystem relaxes that of Flowing to allow the barriers b and c to execute out of order and thus explains the behaviour as follows:

- Finish c (before a and b are finished).
- Satisfy d from the initial write in memory.
- Commit and propagate e, updating the value at location y to 1.
- Commit and finish f.
- Satisfy g from the initial write to x.
- Commit and propagate a, updating the value of x in memory to 54.
- Commit and finish b.

This illustrates how the Flat thread subsystem subsumes the relaxed behaviours enabled by the re-ordering in the Flowing buffers. It is worth noting, that Flat, as currently defined, does not completely subsume the behaviour of Flowing, in the mixed-size setting: for an example [due to Shaked Flur] see the paragraph Section 8.1 in Chapter 8, where this example and is discussed in the context of mismatches between the Flat RISC-V model and the RISC-V operational model, including how it is handled in Flowing. Some of the details of mixed-size accesses in MCA ARMv8, including this example, are still being clarified.

6.3.3 Informal model definition

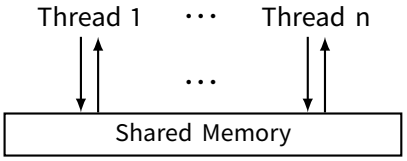
The following describes the simplified model, starting with an informal description of the model states and transitions.

Model caveats The model does not handle the full architecture. The Sail ARMv8 ISA model covers only non-floating-point, non-vector instructions from the application-level ISA. The concurrency model does not yet handle the weaker form of load acquire (`LDAPR`) introduced in

ARMv8.3, the load pair instruction, and the atomic read-modify-write instructions introduced in ARMv8.1. The model does not handle virtual memory, and does not deal with self-modifying code. Moreover, the operational model can deadlock for certain executions involving load/store exclusive instructions, which will be discussed in detail in Section 6.3.10.

Model states The following description is a prose description of the formal Lem model, that is written to closely follow the formal definitions but sometimes abstracts certain detail for the sake of a simpler presentation.

A Flat model state consists of a shared memory and a tuple of thread model states. The shared memory state records all the memory writes that have propagated so far, in the order they propagated, together with some additional information needed for the atomicity guarantees given by load/store exclusives: the *exclusives map*, a map from read requests to sets of write slices. This map maps a load exclusive's read request to the write slices it read from and which have already reached memory. For the purpose of non-exclusive instructions, a flat memory recording only the most recent write to each byte location is sufficient.



As before, each thread model state consists principally of a list or tree of instruction instances, some of which have been finished, and some of which have not, where the intra-instruction behaviour of a single instruction can largely be treated as sequential (but not atomic) execution of its Sail pseudocode. Each instruction instance state includes the pseudocode execution state, and information, about the instruction instance's memory and register footprints, its register and memory reads and writes, whether it is finished, etc., as before. For the sake of readability this section repeats the description of some ideas and definitions shared by Flowing/POP and this model.

Model transitions For any state, the Flat model defines the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Each transition arises from the next step of a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its thread state and/or the shared memory state. Instructions cannot be treated as atomic units: complete execution of a single instruction instance may involve many transitions, which can be interleaved with those of other instances in the same or other threads, and some of this is programmer-visible. The transitions are introduced below and defined in Section 6.3.8, with a precondition and a construction of the post-transition model state for each. The transitions labelled ∘ can always be taken eagerly, as soon as they are enabled, without excluding other behaviour; the • cannot.

Transitions for all instructions

- FETCH INSTRUCTION: This transition represents a 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.
  - REGISTER READ: This is a read of a register value from the most recent program-order predecessor instruction instances that write to that register.
  - REGISTER WRITE: This records a register write in the instruction instance state to make it available for other instruction instances to read.
  - PSEUDOCODE INTERNAL STEP: this covers Sail internal computation, function calls, etc.
  - FINISH INSTRUCTION: At this point the instruction pseudocode is done, the instruction cannot be restarted or discarded, and all memory effects have taken place. For a conditional/computed branch, any non-taken po-successor branches are discarded.

Load instructions

  - INITIATE MEMORY READS OF LOAD INSTRUCTION: At this point the memory footprint of the load is provisionally known and its individual reads can start being satisfied.
- SATISFY MEMORY READ BY FORWARDING FROM WRITES: This partially or entirely satisfies a single read by forwarding from its po-previous writes.
- SATISFY MEMORY READ FROM MEMORY: This entirely satisfies the outstanding slices of a single read, from memory.
  - COMPLETE LOAD INSTRUCTION: At this point all the reads of the load have been entirely satisfied and the instruction pseudocode can continue execution. A load instruction can be subject to being restarted until the FINISH INSTRUCTION transition. In some cases it is possible to tell that a load instruction will not be restarted or discarded before that, e.g. when all the instructions po-before the load instruction are finished. The RESTART CONDITION over-approximates the set of instructions that might be restarted.

Store instructions

  - INITIATE MEMORY WRITES OF STORE INSTRUCTION: At this point the memory footprint of the store is provisionally known.
  - INSTANTIATE MEMORY WRITE VALUES OF STORE INSTRUCTION: At this point the writes have their values and program-order-subsequent 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 the writes can start being propagated to memory.
- PROPAGATE MEMORY WRITE: This propagates a single write to memory.
  - COMPLETE STORE INSTRUCTION: At this point all writes have been propagated to memory, and the instruction pseudocode can continue execution.

Store exclusive instructions For a store-exclusive instruction, before reaching the above store-instruction transitions, the model takes a transition for determining whether the store exclusive will succeed or fail.

- GUARANTEE THE SUCCESS OF STORE EXCLUSIVE: If determined to succeed, the store exclusive can afterwards proceed its execution with transitions for writing the status register and the above store-instruction transitions.
- MAKE A STORE EXCLUSIVE FAIL: If determined to fail, the store exclusive proceeds with the transitions for writing the status register and finishing.

Barrier instructions

- COMMIT BARRIER: This commits the barrier, after which point certain program-order-succeeding instructions waiting for the barrier commitment can proceed with their execution.

6.3.4 Intra-instruction pseudocode execution

As in the case of Flowing and POP, the interface between the instruction semantics and the concurrency model is given by the outcome type of Chapter 2. The particular outcome values used by this model are:

Read_mem(*read_kind*, *address*, *size*, *read_continuation*)	Read request
Excl_res(*res_continuation*)	Store exclusive result
Write_ea(*write_kind*, *address*, *size*, *next_state*)	Write effective address
Write_memv(*memory_value*, *write_continuation*)	Write value
Barrier(*barrier_kind*, *next_state*)	Barrier
Read_reg(*reg_name*, *read_continuation*)	Register read request
Write_reg (*reg_name*, *register_value*, *next_state*)	Write register
Internal(*next_state*)	Pseudocode internal step
Done ( )	End of pseudocode

As before, the ISA model ensures that each instruction instance has at most one memory read, memory write, or barrier step and writes each register bit in its register write footprint exactly once, by rewriting the pseudocode. Every instruction instance then has a single commit point for all memory writes of an instruction. Moreover, since data-flow dependencies in the model emerge from the ordering of register and memory accesses in the Sail pseudocode, the Sail pseudocode is arranged in the maximally liberal order to allow the most relaxed possible concurrency behaviour.

For instance, the high-level Sail steps of a store instruction might be written as follows if not taking the concurrency behaviour into account:

1. read the registers holding the address and data/value of the store;
2. announce the store's address;
3. create a write request.

However, with this ordering of intra-instruction steps, some instruction i program-order-after the store that is blocked on knowing the store's address cannot make progress until the store has read the registers for the address and the data/value of the store. This can create unwanted ordering from the store's data dependencies to i.Hence, for concurrency purposes, the Sail code must instead be ordered in the following way:

1. read the register holding the address of the store;
2. announce the store's address;
3. read the register holding the data/value of the store;
4. create a write request.

Arranging the store's intra-instruction steps in this way allows i to know the address of the store as soon as it is determined and thus potentially some relaxed concurrency behaviour not otherwise possible.

The footprint of each instruction is computed using the exhaustive interpreter when using the Sail interpreter and the hand-written (in Sail) footprint analysis function, as discussed before.

Moreover, the model has to be able to know when a register read value can no longer change (i.e. due to instruction restarts). The model approximates that by recording, for each register write, the set of register and memory reads the instruction instance has performed at the point of executing the write. This information is then used as follows to determine whether a register read value is final: if the instruction instance that performed the register write from which the register read reads is finished, the value is final; otherwise check that the recorded reads for the register write do not include memory reads, and continue recursively with the recorded register reads. For the instructions covered by the model this approximation is exact.

6.3.5 Instruction instance states

Each instruction instance i has a state comprising:

- program_loc, the address from which the instruction was fetched;
- instruction_kind, identifying whether this is a load, store, or barrier instruction, each with the associated kind; or a conditional/computed branch; or a 'simple' instruction.
- regs_in, the set of input registers and slices, as statically determined;
- regs_out, the output registers and slices, as statically determined;
- pseudocode_state (or sometimes just 'state' for short), one of
  - Plain (*next_state*), ready to make a pseudocode transition;
  - Pending_mem_reads (*read_cont*), performing the read(s) from memory of a load; or
  - Pending_mem_writes (*write_cont*), performing the write(s) to memory of a store;
- reg_reads, the accumulated register reads, including their sources and values, of this instance's execution so far;
- reg_writes, the accumulated register writes, including dependency information to identify the register reads and memory reads (by this instruction) that might have affected each;
- mem_reads, a set of memory read requests. Each request includes a memory footprint (an address and size) and, if the request has already been satisfied, the set of write slices (each consisting of a write and a set of its byte indices) that satisfied it.
- mem_writes, a set of memory write requests. Each request includes a memory footprint and, when available, the memory value to be written. In addition, each write has a flag that indicates whether the write has been propagated (passed to the memory) or not.
- successful_exclusive, for store exclusives, indicates whether it was previously guaranteed to

succeed or to fail. If i is restarted this bit is left unchanged ("survives the restart").

- information recording whether the instance is committed, finished, etc.

Read requests include their read kind and their memory footprint (their address and size), the as-yet-unsatisfied slices (the byte indices that have not been satisfied), and, for the satisfied slices, information about the write(s) that they were satisfied from. Write requests include their write kind, their memory footprint, and their value. When referring to a write or read request without mentioning the kind of request this means the request can be of any kind. A load instruction which has initiated (so whose read request list mem_reads is not empty) and for which all its read requests are satisfied (i.e. with no unsatisfied slices) is called *entirely satisfied*. A load exclusive is called *successful* if the first po-following exclusive instruction is a store exclusive that has been guaranteed to succeed (as opposed to not existing, not having been determined to succeed or fail, having been determined to fail, or being a load exclusive). The successful load exclusive and the successful store exclusive are said to be *paired*. If a successful load exclusive has a read request that the exclusives map maps to a write slice *ws* the load exclusive is said to have an outstanding lock on *ws*.

6.3.6 Thread states

The model state of a single hardware thread includes:

- thread_id, a unique identifier of the thread;
- register_data, the name, bit width, and start bit index for each register;
- initial_register_state, the initial register value for each register;
- initial_fetch_address, the initial fetch address for this thread;
- instruction_tree, a tree or list of the instruction instances that have been fetched (and not discarded), in program order.

6.3.7 Shared memory state

The model state of the shared memory comprises a list of memory writes, in the order they propagated to the shared memory.[3] When a write is propagated to the shared memory it is added to the end of the list. When a load operation is satisfied from memory, for each byte of the load operation, the most recent corresponding write slice is returned.

For most purposes, it is simpler to think of the shared memory as an array, i.e., a map from memory locations to write slices, where each memory location is mapped to a one-byte slice of the most recent memory write to that location. However, this abstraction is not detailed enough to properly handle store exclusive instructions. ARMv8 guarantees that if a store exclusive to some location x succeeds, no writes to x from other threads have entered memory between the write the paired load exclusive read from and the store exclusive's write. The architecture does, however, allow such intervening writes if they are from the same thread as the load/store exclusive pair.

For example, assume r_1 is a read exclusive to x from some thead 1, w_1 a write exclusive to x

[3] This description is adapted from the description of the Flat RISC-V model, see Section E.2 or Waterman and Asanović [121], from joint work with Shaked Flur and Peter Sewell.

from the same thread, and $\widetilde{w}$ the write r_1 read from. Then ARMv8 allows w_1 to succeed in the case of the following sequencing of writes to x in memory

$$\widetilde{w}, w'_1, w''_1, w_1,$$

provided that w'_1 and w''_1 are writes by the same thread as w_1. The sequencing

$$\widetilde{w}, w'_2, w''_1, w_1,$$

however, is forbidden if w'_2 is a write by a different thread.

To capture this correctly, treating memory just as an array is insufficient; the model must record more information. The Flat model uses a list of writes as its memory state, for simplicity. Other representations, that do not record the full write history, would also be possible.

6.3.8 Model transitions

Note that in the following, store exclusive instructions that have already been determined to fail are treated like "plain" non-memory instructions.

Fetch instruction A possible program-order successor of instruction instance i can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of i in the thread's instruction_tree are from *loc*;
2. *loc* is a possible next fetch address for i, i.e.:
   2.1. for a non-branch/jump instruction, the successor instruction address (i.program_loc + 4);
   2.2. for an instruction that has performed a write to the program counter register (_PC), the value that was written;
   2.3. for a conditional branch, either the successor address or the branch target address[4]; or
   2.4. for a jump to an address which is not yet determined, any address (this is approximated in the rmem tool implementation, necessarily); and
3. there is a decodable instruction in program memory at *loc*.

Note that this allows speculation past conditional branches and calculated jumps.

Action: Construct a freshly initialised instruction instance i' for the instruction in the program memory at *loc*, including the static information available from the ISA model such as its instruction_kind, regs_in, and regs_out, and add i' to the thread's instruction_tree as a successor of i.

This involves only the thread, not the storage subsystem, as the model assumes a fixed program rather than modelling fetches with memory reads; the model does not handle self-modifying code.

Initiate memory reads of load instruction An instruction instance i with next state Read_mem(*read_kind*, *address*, *size*, *read_cont*) can initiate the corresponding memory reads.

[4] As noted before, in AArch64, all conditional branch (non-computed-branch) instructions have statically determined addresses.

Action:

1. Construct the appropriate read requests *rrs*:
   - if *address* is aligned to *size* then *rrs* is a single read request of *size* bytes from *address*;
   - otherwise, *rrs* is a set of *size* read requests, each of one byte, from the addresses *address* ... *address* + *size* − 1.
2. set *i*.*mem_reads* to *rrs*; and
3. update the state of *i* to Pending_mem_reads (*read_cont*).

Satisfy memory read by forwarding from writes For a load instruction instance *i* in state Pending_mem_reads *read_cont*, and a read request, *r* in *i*.mem_reads that has unsatisfied slices, the read request can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances that are po-before *i*, if the *read-request-condition* predicate holds. This is if:

1. all po-previous `dmb sy` and `isb` instructions are finished;
2. all po-previous `dmb ld` instructions are finished;
3. if *i* is a load acquire, all po-previous store releases are finished; and
   all non-finished po-previous load acquire instructions are entirely satisfied.

Let *wss* be the set of unpropagated write slices from store instruction instances po-before *i* (if *i* is a load acquire, exclude store exclusive writes) that have already calculated their write value, that overlap with the unsatisfied slices of *r*, and which are not superseded by intervening writes (with known address) or writes read from by intervening loads. That last condition requires, for each write slice *ws* in *wss* from instruction i':

- that there is no store instruction po-between *i* and i' with a write overlapping *ws*, and
- that there is no load instruction po-between *i* and i' that was satisfied from an overlapping write slice from a different thread.

Action:

1. update *r* to indicate that it was satisfied by *wss*; and
2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a po-successor of *i*, and every read request r' of i' that was satisfied from wss', if there exists a write slice ws' in wss', and an overlapping write slice from a different write in *wss*, and ws' is not from an instruction that is a po-successor of *i*, restart i' and its data-flow dependents (including po-successors of load acquire instructions).

Note that store release writes cannot be forwarded to load acquires: a load acquire instruction cannot be satisfied before all po-previous store release instructions are finished, and *wss* does not include writes from finished stores (as those must be propagated).

Satisfy memory read from memory For a load instruction instance *i* in state Pending_mem_reads (*read_cont*), and a read request *r* in *i*.mem_reads, that has unsatisfied slices, the read request can be satisfied from memory unless *i* is a successful load exclusive and another successful load exclusive from a different thread has an outstanding lock on writes overlapping with those *r* is

trying to read from. If: the read-request-condition holds (see previous transition).

Action: let *wss* be the write slices from memory covering the unsatisfied slices of r, and apply the action of SATISFY MEMORY READ BY FORWARDING FROM WRITES. In addition, if i is a successful load exclusive, union the slices r is mapped to in the exclusives map with *wss*.

Note that SATISFY MEMORY READ BY FORWARDING FROM WRITES might leave some slices of the read request unsatisfied. SATISFY MEMORY READ FROM MEMORY, on the other hand, will always satisfy all the unsatisfied slices of the read request.

Complete load instruction (when all its reads are entirely satisfied) A load instruction instance i in state Pending_mem_reads (*read_cont*) can be completed (not to be confused with finished) if all the read requests i.mem_reads are entirely satisfied, so if there are no unsatisfied slices.

Action: update the state of i to Plain (*read_cont memory_value*), where *memory_value* is assembled from all the write slices that satisfied i.mem_reads.

Guarantee the success of store exclusive A store exclusive instruction instance i with next state Excl_res(*res_cont*) can be guaranteed to succeed if:

1. the store exclusive has not been determined to fail (as recorded in i.successful_exclusive);
2. assuming i is successful, it can be paired with a load exclusive i' (see INSTRUCTION INSTANCE STATES); and
3. if i' has already been satisfied (not necessarily entirely), let *wss* be the set of propagated write slices i' has read from, then, no slice in *wss* has been overwritten (in memory) by a write from another thread, and no other successful load exclusive from a different thread has an outstanding lock on a write slice from *wss*.

Action:

1. record in i.successful_exclusive that the store exclusive will be successful;
2. if i' has already been satisfied, union the set of write slices the read request of i' is mapped to in the exclusives map with *wss*, where *wss* is as above; and
3. update the state of i to Plain (*res_cont* true).

Make a store exclusive fail A store exclusive instruction instance i with next state Excl_res(*res_continuation*) can be determined to fail if the store exclusive has not been guaranteed to succeed (as recorded in i.successful_exclusive).

Action:

1. record in i.successful_exclusive that the store exclusive was determined to fail; and
2. update the state of i to Plain (*res_cont* false).

Note that a store exclusive instruction determines success or failure as its first non-pseudo-code-internal action after fetching. The promise-success transition must happen before the store exclusive can commit. It does not require the store to have a fully-determined address or to be non-restartable. As a result, a store exclusive that has already promised its success might be restarted. Since other instructions may rely on its promise, the restart will not affect the value of i.successful_exclusive. Instead, when the store exclusive is restarted it will take the same

promise/failure transition as before its restart — based on the value of i.successful_exclusive.

Initiate memory writes of store instruction, with their footprints An instruction instance i with next state

Write_ea(*write_kind*, *address*, *size*, *next_state*′) can announce its pending write footprint.

Action:

1. construct the appropriate write requests:
   - if *address* is aligned to *size* then *ws* is a single write request of *size* bytes to *address*;
   - otherwise *ws* is a set of *size* write requests, each of one byte size, to the addresses $address \ldots address + size - 1$.
2. set i.*mem_writes* to *ws*; and
3. update the state of i to Plain (*next_state*′).

Note that at this point the write requests do not yet have their values. This state allows non-overlapping po-following writes to propagate.

Instantiate memory write values of store instruction An instruction instance i with next state Write_memv (*memory_value*, *write_cont*) can instantiate the corresponding memory writes.

Action:

1. split *memory_value* between the write requests i.mem_writes; and
2. update the state of i to Pending_mem_writes (*write_cont*).

Commit store instruction For an uncommitted store instruction i in state Pending_mem_writes (*write_cont*), i can commit if:

1. i has fully determined data (i.e., the register reads cannot change, see AUXILIARY DEFINITIONS);
2. all po-previous conditional/computed branch instructions are finished;
3. all po-previous `dmb sy` and `isb` instructions are finished;
4. all po-previous `dmb ld` instructions are finished;
5. all po-previous load acquire instructions are finished;
6. all po-previous store instructions, except for store exclusives that failed, have initiated and so have non-empty mem_writes;
   if i is a store release, all po-previous memory access instructions are finished;
7. all po-previous `dmb st` instructions are finished;
8. all po-previous memory access instructions have a fully determined memory footprint;
9. all po-previous load instructions have initiated and so have non-empty mem_reads; and
10. if i is a successful store exclusive then the paired load exclusive i' is finished, and all same-thread writes i' read from are propagated.

Action: record i as committed.

Propagate memory write For an instruction i in state Pending_mem_writes (*write_cont*), and an unpropagated write w in i.mem_writes the write can be propagated if:

1. i is committed;
2. all memory writes of po-previous store instructions that overlap w have already propagated

3. all read requests of po-previous load instructions that overlap with w have already been satisfied, and the load instruction is non-restartable (see RESTART CONDITION);
4. all read requests satisfied by forwarding w are entirely satisfied; and
5. no successful load exclusive from a different thread has an outstanding lock on a write slice that overlaps with w.

Action:

1. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' po-after i and every read request r' of i' that was satisfied from wss', if there exists a write slice ws' in wss' that overlaps with w and is not from w, and ws' is not from a po-successor of i, restart i' and its data-flow dependents;
2. record w as propagated;
3. update the memory with w;
4. if i is a store exclusive that is successfully paired with a load exclusive i', remove the read request of i' from the exclusives map; and
5. for every successful load exclusive that has read from w (by forwarding), add the slices of w this load exclusive read from to the set of write slices the read request of the load exclusive is mapped to in the exclusives map.

Complete store instruction (when its writes are all propagated) A store instruction i in state Pending_mem_writes (*write_cont*), for which all of the memory writes in i.mem_writes have been propagated, can be completed.

Action: update the state of i to Plain(*write_cont* true).

Commit barrier A barrier instruction i in state Plain (*next_state*) where *next_state* is Barrier(*barrier_kind*, *next_state*$'$) can be committed if:

1. all po-previous conditional/computed branch instructions are finished;
2. if i is a `dmb ld` instruction, all po-previous load instructions are finished;
3. if i is a `dmb st` instruction, all po-previous store instructions are finished;
4. all po-previous `dmb sy` barriers are finished;
5. if i is an `isb` instruction, all po-previous memory access instructions have fully determined memory footprints; and
6. if i is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished.

Note that this differs from the previous Flowing and POP models: in Flowing and POP, barriers commit in program-order and potentially re-order in the storage subsystem. Here the thread subsystem is weakened to subsume the re-ordering of Flowing and POP's storage subsystem.

Action: update the state of i to Plain (*next_state*$'$).

Register read An instruction instance i with next state Read_reg (*reg_name*, *read_cont*) can do a register read if every instruction instance that it needs to read from has already performed the expected register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in

program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from initial_register_state. Let *register_value* be the assembled value from *read_sources*.

Action:

1. add *reg_name* to *i*.reg_reads with *read_sources* and *register_value*; and
2. update the state of *i* to Plain (*read_cont register_value*).

Register write An instruction instance *i* with next state Write_reg (*reg_name*, *register_value*, *next_state*′) can do the register write.

Action:

1. add *reg_name* to *i.reg_writes* with *write_deps* and *register_value*; and
2. update the state of *i* to Plain (*next_state*′).

where *write_deps* is the set of all *read_sources* from *i*.reg_reads and a flag that is set to true if *i* is a load instruction that has already been entirely satisfied.

Pseudocode internal step An instruction instance *i* with next state Internal (*next_state*′) can do that pseudocode-internal step.

Action: update the state of *i* to Plain (*next_state*′).

Finish instruction A non-finished instruction *i* with next state Done () can be finished if:

1. if *i* is a load instruction:
   - 1.1. all po-previous `dmb sy` and `isb` instructions are finished;
   - 1.2. all po-previous `dmb ld` instructions are finished;
   - 1.3. all po-previous load acquire instructions are finished;
   - 1.4. it is guaranteed that the values read by the read requests of *i* will not cause coherence violations, i.e., for any po-previous instruction instance *i*′, let *cfp* be the combined footprint of propagated writes from store instructions po-between *i* and *i*′ and fixed writes that were forwarded to *i* from store instructions po-between *i* and *i*′ including *i*′, and let $\overline{cfp}$ be the complement of *cfp* in the memory footprint of *i*. If $\overline{cfp}$ is not empty:
     - 1.4.1. *i*′ has a fully determined memory footprint;
     - 1.4.2. *i*′ has no unpropagated memory write that overlaps with $\overline{cfp}$; and
     - 1.4.3. If *i*′ is a load with a memory footprint that overlaps with $\overline{cfp}$, then all the read requests of *i*′ that overlap with $\overline{cfp}$ are satisfied and *i*′ cannot be restarted (see Restart condition).

     Here a memory write is called fixed if it is the write of a store instruction that has fully determined data.
   - 1.5. if *i* is a load acquire, all po-previous store release instructions are finished;
2. *i* has fully determined data; and
3. all po-previous conditional/computed branches are finished.

Action:

1. if *i* is a branch instruction, discard any untaken path of execution, i.e., remove any (non-

finished) instructions that are not reachable by the branch taken in instruction_tree; and
2. record the instruction as finished: set finished to true.

6.3.9 Auxiliary definitions

Fully determined Informally, an instruction is said to have *fully determined footprint* if the memory reads feeding into its footprint are finished. A register write w, of instruction i, with the associated *write_deps* from i.reg_writes is said to be fully determined if one of the following conditions hold:

1. i is finished; or
2. the load flag in *write_deps* is false and every register write in *write_deps* is fully determined.

An instruction i is said to have *fully determined data* if all the register writes of *read_sources* in i.reg_reads are fully determined. An instruction i is said to have a *fully determined memory footprint* if i has done enough instruction steps to compute its footprint and all the register writes of *read_sources* in i.reg_reads that are associated with registers that feed into i's memory access footprint are fully determined.

Address/data known A load is said to have *known footprint/address* or to have *computed its footprint/address* when it has done the Initiate memory reads of load instruction transition or is in a state where the next transition is the Initiate memory reads of load instruction transition.

A store is said to have *known footprint/address* or to have *computed its footprint/address* when it has done the Initiate memory writes of store instruction transition. Its *data* is said to be *known/computed* when the store has taken the Instantiate memory write values of store instruction transition.

Restart condition To determine if instruction i might be restarted the model uses the following recursive condition: i is a non-finished instruction and at least one of the following holds,

1. there exists an unpropagated write w such that applying the action of the Propagate memory write transition to w will result in the restart of i;
2. there exists a non-finished load instruction l such that applying the action of the Satisfy memory read from memory transition to l will result in the restart of i (even if l is already entirely satisfied); or
3. there exists a non-finished instruction i' that might be restarted and i is in its data-flow dependents (including po-successors of load acquire instructions).

6.3.10 Store exclusive issues

The MCA ARMv8 architecture intends for the success bit of store exclusives not to introduce dependencies, to allow (e.g.) hardware optimisations that dynamically replace load/store exclusive pairs by atomic read-modify-write operations that can execute in the memory subsystem and therefore be guaranteed to succeed.[5] Therefore, the official ARMv8-axiomatic model assumes

[5]This description is taken from that found in the supplementary material of Pulte et al. [104].

all address/data/control dependencies to be from reads, not writes. In the operational model, matching this weakness has proved to be difficult: it means the operational model must be able to promise the success or failure of a store exclusive instruction even before any of its registers reads/writes have been done, so before the store exclusive's address and data are available. The

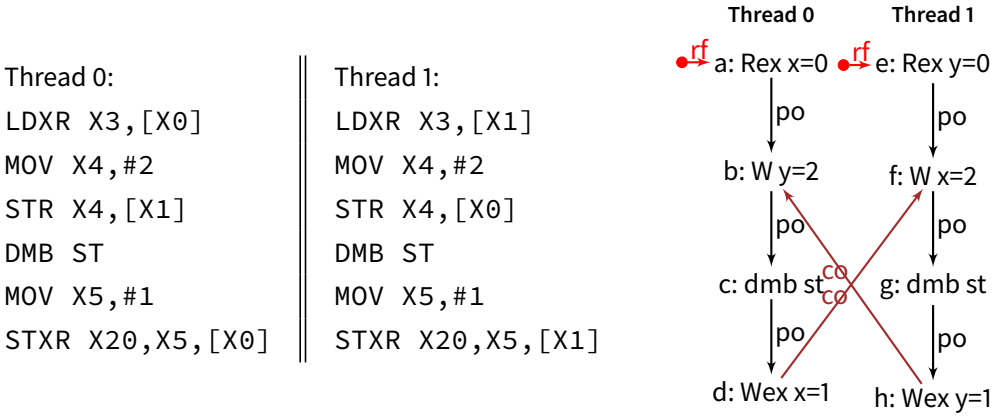


Figure 6.12: Example of a test with deadlocking behaviour in the operational model. The assembly program on the left assumes that on both threads register `X0` holds the address of x, `X1` that of y. `LDXR` is the assembly syntax for load exclusive, `STXR` for store exclusive, and the first argument of `STXR` is the status register that will contain the information about whether the store suceeded.

early success promises are the source of deadlocks in the operational model. To illustrate this consider, for example, the litmus test from Figure 6.12 and a state where both a and e are satisfied and finished, and where b and f are not propagated. Then d can promise its success, locking memory location x, and h can promise its success, locking location y. But now there is a deadlock:

- For d to propagate and release the lock on x, c has to be committed and hence b propagated. But b cannot propagate since y is locked.
- For h to propagate and release the lock on y, g has to be committed and hence f propagated. But f cannot propagate since x is locked.

Similar situations arise from cases where there are other barriers or release/acquire instructions in-between the load and the store exclusive, or if the store exclusive has additional dependencies that the load exclusive does not have. These are cases that are not really intended to be supported by the architecture.

The model can also currently deadlock if a load and a store exclusive are paired successfully but later turn out to have different addresses: if the store exclusive promises its success before its address is known it locks the matched load exclusive's memory location; when they later turn out to be to a different addresses it never unlocks it. This issue can be fixed, but it is currently still being clarified what exactly the architecturally allowed behaviour should be.

These model deadlocks do not compromise the soundness of the model — the model still allows all the final outcomes it should — and do not impact exhaustive exploration of concurrent

programs in the model (such deadlocking executions can be ignored, although some execution time may be wasted exploring executions that will eventually deadlock), but mean the model is not entirely incremental. As a result, the model intuition is less clear, and the deadlocks are confusing for users when interactively stepping through executions in the rmem tool.

6.4 ARMv8-Axiomatic

The revised multicopy atomic ARMv8 architecture for the first time has an official formal concurrency model. The following section gives the definition of this official model. The official ARMv8 concurrency model is the ARMv8-axiomatic model specified in herd [17], the architecture documentation contains a prose version of the same.

6.4.1 Herd and candidate executions

In the operational model, the possible concurrency behaviours of a program are a fact derived from the legal traces of the operational model with that program as input, i.e. the operational model is a function from programs to legal executions. The axiomatic model, on the other hand, checks whether a given *candidate execution* is a legal execution for a given program. A candidate execution is a complete concrete execution of the concurrent program that is described in terms of a set of events and relations over them, abstracting from the instruction semantics. In the case of herd [14, 17] the event set consists of read events for load instructions, write events for store instructions, and barrier events for barrier instructions, and the candidate execution contains a number of relations on these:

program-order (po) relates two events in the order of their originating instructions in the program in a concrete control-flow unfolding.

reads-from (rf) relates a write event W with a read R that reads from W.

coherence (co) relates a write W with another W' in the order the writes are sequenced in memory.

read-modify-write (rmw) relates a R read (*read exclusive*) with a write W (*write exclusive*) if W originates from a successful store exclusive that is paired with the load exclusive R originates from.

address dependency (addr) relates a read event R with a read or a write event E if the value read by the load instruction R originates from is written to a register that affects the address calculation of the load or store instruction of E.

data dependency (data) relates a read event R with a write event W if the value read by the load instruction R originates from is written to a register that affects the calculation of the value written to memory by the store instruction of W.

control dependency (ctrl) relates a read event R with an event E if the value read by the load instruction R originates from is written to a register that affects the condition or target address of a conditional or computed branch program-order-before the instruction E originates from.

The candidate executions have to satisfy only minimal consistency requirements to be compatible with the instruction semantics, and it is the axiomatic concurrency model that decides whether a candidate execution is legal. The possible behaviours of an input program are then the set of candidate executions that satisfy the predicate defined by the axiomatic model. The main part of the official ARMv8 axiomatic herd model [47] is below. This model is defined for programs in which all memory accesses have the same size, or "non-mixed-size programs", only.

6.4.2 Definition

The ARMv8-axiomatic model specifies which candidate executions are legal by defining a number of relations (ca, obs, dob, ...), and then requiring certain axioms to hold of legal executions — in this case the three axioms at the end of the model internal, external, and atomic — named, using the syntax "... as ...". The primitives for defining relations used here (and in the equivalence proof later) are the following (where S and S' are arbitrary relations):

- Union of relations $S|S'$.
- Intersection of relations $S\&S'$.
- Sequential composition of relations S; S'.
- Transitive closure $S+$.
- Reflexive transitive closure S^*.
- Option operation $S?$, denoting the union of S and the identity relation.
- Set difference $S \setminus S'$.
- Relation inversion: S^{-1}.
- Binary identity relation for a given domain: $[M]$ denotes the set $\{(E,E) \mid E \in M\}$. This can be used in combination with sequential composition to restrict relations to events of certain sets on the left or right-hand side of the edges: for example ctrl;[W] is the set of pairs (E,E') where E' control-depends on E, and E' is restricted to be a write.
- Sets of all events of a particular type:
  - W: for write events
  - R: for read events
  - F: for barrier events
  - ISB: for `isb` events
  - DMB.SY: for `dmb sy` events
  - DMB.LD: for `dmb ld` events
  - DMB.ST: for `dmb st` events
  - L: for write release events
  - A: for acquire read events
  - Q: for weak acquire read events from LDAPR instructions

In addition, the ARMv8 model assumes some definitions derived using these primitives. The from-reads relation (fr) [7, 14] relates a read R with any write W' that is coherence-after the write W it reads from; this is defined as $\mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$, so $(W,R) \in \mathsf{rf}$ and $(W,W') \in \mathsf{co}$ implies $(R,W') \in \mathsf{fr}$. The relation po-loc is the program order relation restricted to read and write events

to the same address. Additionally, the rf, co, and fr relations are each subdivided into their "internal" (same-thread) and "external" (different-thread) parts, suffixed i and e respectively.

Using these built-in and derived operations and sets the ARMv8-axiomatic model requires legal executions to satisfy three axioms:

internal This is a standard axiom, sometimes called *SC-per-location*, that rules out coherence violations in individual threads and requires the acyclicity of the union of po-loc, co, rf, and fr.

external This axiom is the main axiom, that restricts the possible interaction between different threads. The relations dependency-ordered-before (dob), atomic-ordered-before (aob), and barrier-ordered-before (bob), define partial orders of same-thread events for which program order is preserved despite the out-of-order execution. The relation dob specifies the dependencies that are respected within the threads, aob defines ordering related to load/store exclusive instructions, and bob describes which ordering guarantees barrier instructions give. The relation observed-by (obs) on the other hand captures ordering resulting from the interactions of events of different threads, given by the relations rf, fr, and co restricted to events from different threads. The external axioms now requires compatibility of the ordering within the threads defined by the relation dob|aob|bob with the ordering across threads defined by obs by requiring the irreflexivity of the transitive closure of their union (or the acyclicity of their union).

atomic The atomic axiom captures the atomicity guarantees given by successful load/store exclusive pairs: if a load exclusive's read *RE* reads from some write W and successfully pairs with a store exclusive, there must be no write W' coherence-between W and the store exclusive's write *WE*. So if *RE* and *WE* are rmw-related ("paired"), then these must not be related by fre; coe via a different thread write W'.

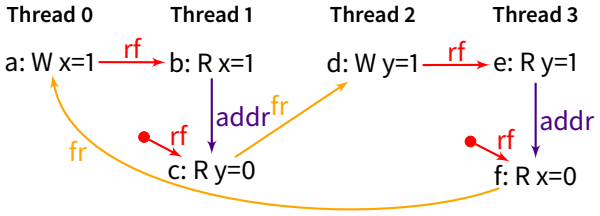
```
(* Coherence—after *)
let ca = fr | co

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

(* Dependency—ordered—before *)
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [R]
        | addr; po; [W]
        | (ctrl | data); coi
        | (addr | data); rfi

(* Atomic—ordered—before *)
let aob = rmw
        | [range(rmw)]; rfi; [A | Q]
```

```
(* Barrier—ordered—before *)
let bob = po; [DMB.Sy]; po
        | [L]; po; [A]
        | [R]; po; [DMB.LD]; po
        | [A | Q]; po
        | [W]; po; [DMB.ST]; po; [W]
        | po; [L]
        | po; [L]; coi
(* Ordered—before *)
let rec ob = obs | dob | aob | bob | ob; ob
(* Internal visibility requirement *)
acyclic po—loc | ca | rf as internal
(* External visibility requirement *)
irreflexive ob as external
(* Atomic: Basic LDXR/STXR constraint to forbid intervening writes. *)
empty rmw & (fre; coe) as atomic
```



To illustrate the definition, recall the previous IRIW+addrs test. This non-multicopy-atomic behaviour requires a to propagated to Thread 1 without immediately also propagating to Thread 3, and d to propagated to Thread 3 without immediately also propagating to Thread 1, and is forbidden in the revised ARMv8 architecture. The ARMv8-axiomatic model forbids this behaviour using the main axiom: there is a cycle of the shape rfe; addr; fre; rfe; addr; fre: the addr edges are part of the dependencies dob that enforce memory model ordering, rfe and fre are interaction across threads that create memory model ordering; the edges form a cycle in the relation ob that the main axiom external requires to be acyclic.

Chapter 7

ARMv8 model equivalence

With the models defined as in Chapter 6, the Flat operational model and the ARMv8-axiomatic model are equivalent presentations of the same concurrency behaviour, for non-mixed-size programs.[1]This chapter discusses the relation between the two models in detail, and, as the main result of this thesis, gives a proof of equivalence of the two models for non-mixed-size programs and finite executions. First we first present the experimental equivalence results (that provide additional evidence for the equivalence theorem).

The two models are experimentally validated on a test suite of 11310 litmus tests, consisting mostly of families of tests systematically generated using diy [12], together with some hand-written tests; it includes the tests used in Flur et al. [51, 52]. Of these, 2369 are mixed-size tests and so cannot be used for the comparison between the operational and the axiomatic model; another 3 tests use instructions that are not supported by the axiomatic model; 2 tests are too big for both models, and additional 2 are too big for Flat; 11 tests make use of a -1 value that is interpreted inconsistently by the tools.

Running the Flat model in rmem, and the ARMv8-axiomatic model in herd for the above tests, the two models allow the same set of final states for each test, with the exception of six tests due to herd's handling of dependencies from the result register of a store exclusive. As discussed earlier, it is the intention of the revised ARMv8 architecture for the success bit register write of store exclusive instructions not to introduce memory model ordering (the dependency is not preserved); at the time of writing, in herd the dependency coming out of the store exclusive translates to a dependency out of the paired load exclusive (in a way that does not match the architectural intention). The six tests in which the two models experimentally mismatch are tests in which this handling of dependencies becomes observable.

Assuming, however, a fixed handling of the store exclusive dependencies in the axiomatic model — whereby a syntactic dependency out of the store exclusive's register write does not create memory model ordering from the load exclusive or the store exclusive — the models can be shown to allow the same behaviour for any given non-mixed-size program. This chapter gives a proof of equivalence for the intersection of the features covered by the two models for finite executions. In particular, the proof only covers the behaviour of programs in which all memory accesses are not misaligned and of the same size (this also excludes the load-pair and store-pair instructions), and it does not cover the `LDAPR` weaker acquire instructions (which at the time of working on the proof was not covered by the Flat operational model). Additionally, the proof assumes that if a load and store exclusive instruction are successfully paired (rmw-related in the axiomatic model),

[1]The results of this chapter have been published in Pulte et al. [104] and the chapter text including the relation of the models, the informal description of the proof, the proof text, and the experimental results are minor adaptations of that of Pulte et al. [104] and the supplementary material of same paper.

then they are to the same address — as discussed earlier, the case of different-address paired load/store exclusive instructions is still being clarified. The proof assumes no reading from initial memory, i.e. that every memory read is satisfied from a write that originates from a store in the input program. Finally, the proof assumes the two models share the definition of the instruction semantics and thus agree on the definition of dependencies.

In order to state the equivalence of the two models, one first has to define a way to relate operational and axiomatic model executions. Given this definition the chapter proves the equivalence of the two models under the assumptions above.

7.1 Relating axiomatic and operational models

The basic problem in relating the Flat model of Section 6.3 and the ARMv8-axiomatic model of [22, 47] (ARMv8-ax for short) is the mismatch between the events the two models refer to: whereas the axiomatic model has a single event per instruction (one read per load, one write per store), the operational model has several transitions associated with each instruction. Moreover, those transitions have subtle ordering properties: for example, a write can be read from by forwarding after its instantiate-memory-write transition (when an address and value are provisionally available), which is before it is propagated (and hence visible to be read from memory by other threads), while it can be propagated only when (among other conditions) all previous memory writes have announced their address.

The idea underlying the equivalence proof is that there is a correspondence between particular Flat transitions and events in ARMv8-ax's ob relation:

ARMv8-ax event	**Flat transition**
Write	Propagate memory write
Read	final Satisfy memory read (from memory or by forwarding)
Barrier	Commit barrier

Under this correspondence the relations of ARMv8-ax can be viewed as ranging over transitions in a Flat trace: in particular, the main axiom external that checks for the acyclicity of certain such relations can be interpreted as describing the order of transitions in a Flat trace for a given execution. For example, [L];po;[A] can be read as saying: in Flat an acquire read can only be satisfied if all program-order-preceding release writes are propagated. The proof establishes that under this interpretation the axioms of ARMv8-ax are a sound and complete characterisation of Flat, for finite executions.

However, to state this formally requires defining under which conditions executions of ARMv8-ax and Flat should be considered equivalent, since the two models have different notions of execution: ARMv8-ax has candidate executions; Flat has traces from the initial state. To relate the two notions of executions, one has to define how a Flat trace induces po, co, rf, and rmw of the ARMv8-ax candidate execution, and then call a Flat trace equivalent to a candidate execution for the same input program if their po, co, rf, and rmw relations are the same. We give the definition of a candidate execution induced by a Flat trace, for the non-mixed-size case the proof

is concerned with.

Let *Tr* be a finite Flat trace for the given program to a final state. Define:

- $(E, E') \in \mathsf{po}_{Tr}$ for two (read/write/barrier) events (E, E') if E is before E' in one of the thread's instruction trees after *Tr* (in a final state all instructions are finished, and since finished branch instructions have only one successor in the tree, each tree is a linear order of instructions)
- $(W, W') \in \mathsf{co}_{Tr}$ for two same-address writes W and W' if W propagates to memory before W' in *Tr*
- $(W, R) \in \mathsf{rf}_{Tr}$ for a write W and a read R if, in the final satisfy-read transition of R in *Tr*, the read R is satisfied by W ("final" since if the trace *Tr* involves a restart of R it might be satisfied multiple times)
- $(RE, WE) \in \mathsf{rmw}_{Tr}$ for a read exclusive *RE* and write exclusive *WE* if in *Tr* the write exclusive *WE* is successfully paired with the read exclusive *RE*.

Given this definition one can state:

Theorem 3. *Let* $x = (\mathsf{po}, \mathsf{co}, \mathsf{rf}, \mathsf{rmw})$ *be a finite candidate execution for a given program P. The execution x is valid under ARMv8-axiomatic if and only if there exists a valid finite trace Tr of Flat Operational for the program P such that* $(\mathsf{po}_{Tr}, \mathsf{co}_{Tr}, \mathsf{rf}_{Tr}, \mathsf{rmw}_{Tr}) = (\mathsf{po}, \mathsf{co}, \mathsf{rf}, \mathsf{rmw})$.

The following gives the high-level ideas of the proof for both directions of the implication.

7.2 Proof overview

7.2.1 "If": Flat Operational behaviour included in ARMv8 Axiomatic

To show that the candidate execution $x = (\mathsf{po}_{Tr}, \mathsf{co}_{Tr}, \mathsf{rf}_{Tr}, \mathsf{rmw}_{Tr})$ induced by an arbitrary valid finite Flat trace *Tr* is allowed by ARMv8-ax one has to prove that any such x satisfies the three axioms: external, internal, and atomic. The following assumes that for any finite trace of Flat there is an equivalent one that has no restarts or discarded instructions, which intuitively holds because instruction restarts and discards are down-closed with respect to the information flow in the model, and hence, without loss of generality, that *Tr* involves no instruction restarts or discards.

First consider external (which can be regarded as the main axiom). The basic idea of this part of the proof is that, interpreting ob as a relation between Flat transitions using the correspondence given above, one can show that *Tr* (viewed as a relation) contains each edge of ob for the candidate execution x. Since Flat traces with no restarts are acyclic by construction, it follows that ob as a subset of *Tr* must be acyclic as well. To illustrate the proof of $\mathsf{ob} \subseteq Tr$ consider an edge $e \in \mathsf{addr;po;[ISB];po;[R]}$. Then $e = (R, R')$ for a read R and a po-later read R', there is an `isb` B po-between R and R', and R is a memory read that feeds into the address of a memory instruction po-before B. Under the above correspondence e is an edge from the satisfaction of R to that of R'. The proof is now as follows: in Flat

- for R' to be satisfied, all po-earlier `isb` s, including B, must be finished and hence committed;
- for B to commit (since B is an `isb`) all memory reads feeding into the address of a memory instruction po-before B, including R, have to be finished;

- and hence each of those memory reads, including R, has to be satisfied.

Therefore R is satisfied before R', and Tr contains all edges of addr;po;[ISB];po;[R]. The proof proceeds in this way for the other edges to show that ob is a subset of Tr, and hence that ob is acyclic.

The proof that the atomic axiom is satisfied shows that Flat preserves the invariant that for any successful load/store exclusive pair $(RE, WE) \in$ rmw their location in memory is locked by RE for other-thread writes until WE reaches memory, and thereby guarantees the atomicity property of exclusives. The proof for the internal axiom shows that any per-thread coherence violation (as excluded by internal) during a Flat Operational trace leads to a restart of the violating instruction. Since by assumption Tr has no restarts there can be no such coherence violation.

7.2.2 "Only If": ARMv8 Axiomatic behaviour included in Flat Operational

The other direction of the proof is more difficult. Here one has to show that, given a finite candidate execution x allowed by ARMv8-ax, there exists a finite Flat trace Tr that induces x. One would like to do this by defining Tr by induction on a linearisation S of ob, according to the above correspondence: start with an empty trace and for each next event of S extend it with the corresponding transition. However, ARMv8-ax's ob does not give enough detail to easily construct a legal Flat trace: ob can be interpreted as describing an order of Flat satisfy-read/propagate-write/commit-barrier transitions in a trace for x, but it lacks information about the read/write/barrier finish transitions, for example.

To illustrate this, consider the step case for an `isb` barrier event B in the inductive definition of Tr: after constructing the trace Tr' for a prefix of S, one needs to extend it for the next element B in S. The trace Tr' should be extended with the barrier-commit transition for B. But for this to be a legal Flat trace the barrier-commitment condition has to hold. This requires, among other things, that all reads R that the `isb` B is control-flow dependent on are finished. This in turns means that, for each of these reads R, that all reads R' po-before R to the same address, where there is no same-address write po-between R' and R, have to be satisfied and non-restartable. But from the definition of ob it is not clear why those reads R' would be satisfied and non-restartable, and therefore why committing B is a legal transition in Flat in the state reached with Tr'.

To address this problem the gap between the Flat operational and ARMv8-ax models can be bridged by introducing an intermediate model, *Flat-axiomatic*, and splitting the proof into two parts: (a) that a candidate execution accepted by ARMv8-ax is accepted by Flat-axiomatic, and (b) that for each legal candidate execution of Flat-axiomatic there exists a Flat Operational trace. Flat-axiomatic has the same structure as ARMv8-ax — it has the internal and atomic axioms, and the main axiom external that requires the acyclicity of the relation Order — making it possible to relate ARMv8-ax and Flat-axiomatic, for (a). The relation Order, on the other hand, tries to capture the order of transitions in Flat as closely as possible, to make it easier to construct a trace from its candidate executions, for (b).

Starting with (b), the above problem becomes manageable when constructing Tr by induction on a linearisation of Flat-axiomatic's Order relation, since Order contains additional information

about read-finish transitions (among others). In this case, for example, the following holds:

1. $(R, B) \in$ [R];ctrl;[ISB] $\subseteq$ Order for each such read R feeding into B's control-flow; and
2. $(R', B) \in$ [R];(po-loc\(po-loc;[W];po-loc));[R];ctrl;[ISB] $\subseteq$ Order for each read R' that is po-loc-before R with no same-address write between R and R'.

Then by construction of Tr' all such R and R' are satisfied; and in combination with other edges of Order one can show that the preconditions for finishing R' are met (so R' is non-restartable), and R and therefore B can be finished.

The last part of the proof, (a), shows that Flat-axiomatic accepts all candidate executions ARMv8-axiomatic accepts. Since the two have the same internal and atomic axioms this only requires proving that Flat-axiomatic's Order relation has a cycle only if ob from ARMv8-ax has one. This proof therefore has to deal with the edges of Flat-axiomatic that ARMv8-ax does not include.

The fundamental reason why Flat-axiomatic has edges that ARMv8-ax does not mention is that per-thread coherence in Flat Operational and ARMv8-ax are necessarily handled differently: the axiomatic model has the full candidate execution available and rules out coherence violation by fiat, while the operational model computes incrementally and preserves per-thread coherence using its restart mechanism. Some transitions, however, are only allowed in Flat Operational when certain instructions cannot be restarted anymore. Recall the example from the description of part (b) of the proof: here, committing an `isb` B requires finishing the read R, and in turn that all reads R' po-before R are satisfied and non-restartable.

The proof of (a) now shows that each edge of Flat-axiomatic's Order that is not mentioned in ob of ARMv8-ax is subsumed by other edges contained in the transitive closure of ob. In this example, for the edge $(R', B) \in$ Order this involves reasoning about the composition of (R', B) with other edges in Order: since this edge cannot create a cycle by itself it can only participate in a cycle when composed with other edges, for example addr. But the edge set addr;[R];po-R-loc;[R];ctrl;[ISB] is subsumed by addr;po;[ISB] which in turn can be shown to be subsumed by ob. Proceeding in a similar way for the other edges shows any cycle in Order also implies one in ob.

7.3 Proof

Assumptions This proof assumes

1. Finite candidate executions and finite Flat traces.
2. No mixed-size accesses and no misaligned accesses. No load/store-pair instructions.
3. No initial writes/initial memory state: any write in the execution originates from one of the input program's threads.
4. That if a write exclusive is successfully paired with a read exclusive, then they are to the same address.
5. ARMv8 Axiomatic and Flat Operational use the same instruction semantics.
6. $(E, E') \in$ addr if and only if: E is a read and E' is a read or a write and E feeds into a register write that affects the footprint/address of E'. Note: this means $(E, E') \in$ addr does not hold if E is a write exclusive.

7. $(E, E') \in \mathsf{data}$: E is a read and E' is a write and E feeds into a register write that affects the value written by E'. Note: this means $(E, E') \in \mathsf{data}$ does not hold if E is a write exclusive.
8. $(E, E') \in \mathsf{ctrl}$: E is a read and E' is a write and E feeds into a register write that affects a conditional branch or a computed branch instruction program-order-before E'. Note: this means $(E, E') \in \mathsf{ctrl}$ does not hold if E is a write exclusive, and that control dependencies are not delimited — $\mathsf{ctrl}; \mathsf{po} \subseteq \mathsf{ctrl}$.
9. All register reads of a load affect its footprint/address.
10. All register reads of a store affect its footprint/address and data.

For the latter points, recall the aforementioned remarks on load/store exclusive dependencies: the proof assumes, following the architectural intention, that the success bit register write of a store exclusive instruction introduces no memory ordering. Therefore the above dependency relations are assumed to always have a read on the left-hand side. (In herd, dependencies out of store exclusive instructions are translated to dependencies from the paired load exclusive. The proof assumes this is not the case, as intended in ARMv8.)

The assumption that a load or store only reads registers affecting its footprint/address or value is important to guarantee that after a load or store is satisfied/propagated and completed it has no additional dependencies to resolve: it does not need to wait for other instructions to write a register it depends on before it can finish. This is true for the instructions currently covered by the model, where the only actions after completion of a load or store are so-called *pseudo-register* reads: reads of registers whose value is statically determined in the parts of the architecture covered by the Sail models, and which thus do not create ordering. Once the Sail models cover more aspects of the architecture this question may have to be revisited and the operational model relaxed so such dependencies do not create unwanted ordering.

The full proof can be found in the appendix, with the structure as follows:

1. Section D.1 (p. 215) gives the proof that ARMv8-axiomatic allows all behaviours allowed by the Flat operational model. It first slightly simplifies the axiomatic model, then shows that for every candidate execution induced by Flat the main axiom external holds (in Section D.1.1), it then proves the intra-thread coherence axiom internal holds (Section D.1.2), and last shows the atomic axiom holds (Section D.1.3).
2. Section D.2 (p. 225) defines the Flat-axiomatic intermediate model.
3. Section D.3 (p. 230) proves that the Flat operational model allows the behaviours allowed by the Flat-axiomatic model.
4. Finally, Section D.4 (p. 245) shows that the Flat-axiomatic model allows all behaviours allowed by the ARMv8-axiomatic model.

The statement then follows by Theorem 11, Theorem 13, and Theorem 12.

Chapter 8

RISC-V concurrency

The community around the RISC-V architecture has recently formed the Memory Model Task Group, chaired by Daniel Lustig, for the purpose of designing and specifying the architecture's new concurrency semantics.

Earlier versions of the RISC-V ISA manual had a relaxed non-multicopy-atomic concurrency model [120]. In the process of checking the C11 compilation scheme for RISC-V, Trippel et al. [120] identified several issues in this earlier memory model: it had no cumulative barriers, did not enforce ordering resulting from dependencies, and allowed read-read coherence violations [120]. To address these shortcomings the RISC-V Memory Model Task Group was formed [26].

Shaked Flur, Peter Sewell, and the author joined this Task Group in August 2017 and have since been active participants. At that point the group was discussing various design choices for a hypothetical multicopy atomic relaxed model enforcing most syntactic dependencies, with draft axiomatic and operational formalisations, but also the question of whether RISC-V should adopt the TSO memory model. The Task Group's work took place in online mail discussions and weekly phone conference meetings, and involved debating various semantics choices: whether RISC-V should allow load-store re-ordering, the TSO question, and various details of a potential relaxed model, including same-address ordering choices, dependency questions, the semantics of certain barriers and release/acquire, compilation schemes for C/C++, etc.

After much discussion the Task Group has converged on a proposal for the RISC-V memory model that closely follows that of MCA ARMv8, but also incorporates some changes motivated by the deadlock issues in the operational modelling of ARMv8's load/store exclusive instructions described in Section 6.3.10 (which similarly affect atomic memory operations). The RISC-V architecture also has an optional Ztso extension that identifies CPUs implementing the simpler TSO memory model. The group's chair Daniel Lustig presented the proposed concurrency architecture at the RISC-V workshop in Barcelona, 7-10 May 2018. After the group's informal internal voting process in July 2018 that voted in favour of the proposal the model was officially ratified in late September 2018. The group has since continued work, discussing details of mixed-size memory accesses and a potential extension of the memory model to cover the concurrency semantics of self-modifying code.

The following part of this chapter gives an overview over some important choices that were discussed.

8.1 Semantic choices

In the process, a number of choices in RISC-V's concurrency model were discussed. The following gives an overview.

TSO Some discussion was concerned with a proposal for RISC-V to adopt the TSO memory model instead of a more relaxed ARMv8-like model. Much of the controversy revolved around the feasibility of developing efficient hardware under the TSO memory model, and the potential benefits it would present for software due to the simpler concurrency model. The group did not reach a conclusion on this question. As a compromise, RISC-V has adopted a relaxed ARMv8-like model, but offers the Ztso extension: an extension that allows a processor to expose that it has TSO concurrency behaviour to the operating system and software running on the processor; software in turn can declare to require the TSO-extension if it is not written to handle the more relaxed concurrency model. The relaxed ARMv8-like model, however, is the base model, and the Unix platform, for instance, will require software to handle the relaxed memory model.

Multicopy atomicity The question of whether RISC-V should have a non-multicopy-atomic model was mostly settled at the time we joined the Task Group. Although non-multicopy-atomicity was deemed to be potentially beneficial for highly-parallel many-core systems, and "minion cores" (small cores to which the main core offloads certain I/O intensive tasks, see for example [36]), the complexity non-multicopy-atomicity would likely have added to the memory model was considered too great. Consequently, the RISC-V model is multicopy atomic.

Load-buffering Due to the potential advantages for programming language concurrency models, the Task Group also discussed forbidding load-store re-ordering (load-buffering) behaviour. The combination of hardware-allowed load-buffering behaviour and the necessity for language-level concurrency models to be liberal enough to allow for the effects of compiler optimisations leads to the difficulty of defining language-level concurrency models that do not allow certain undesirable behaviour, called out-of-thin-air behaviour, as discussed in Batty et al. [32]. Since, however, there are a number of existing implementations — released or in production — exhibiting load-buffering behaviour this non-backwards-compatible change was not considered an option.

With these higher-level choices fixed, the group converged on a multicopy atomic model allowing load buffering, of a similar shape as ARMv8's, and the discussion focused on the details of the preserved ordering within the threads, and the semantics of load/store exclusive instructions and atomic operations.

Same address ordering Earlier proposals for the RISC-V concurrency model enforced stronger thread-internal ordering for loads: the earlier draft RISC-V-axiomatic model enforced the orderings (addr|data);po-loc;[R] and [R];po-loc \ (po-loc; po-loc);[R]: making all loads wait for program-order-previous memory accesses to the same address to have their address and data determined, and making same-address loads that are not separated by another same-address access in program order satisfy in order. This forbids certain behaviours resulting from same-address load-load out-of-order satisfaction as well as some store-forwarding allowed in ARMv8. As a consequence, this model, for example, disallowed the RSW behaviour shown in the introduction's Figure 1.7. The reason for this was mostly that earlier versions of the draft specification featured a different operational model in which preventing coherence violations in the presence of such behaviours

would have been more difficult. However, the fact that RSW behaviour is widely observed in practice in both POWER and ARMv8 implementations may indicate it naturally arises in micro-architecture, and forbidding it on the grounds of minor modelling simplifications did not seem justified. RISC-V has therefore shifted to adopt ARMv8's more relaxed semantics here (i.e. including the weaker ordering (addr|data);rfi;[R] for thread-internal forwarding instead of the two previously shown edges).

Fence.i RISC-V has an instruction synchronisation barrier `fence.i`. In ARMv8 and POWER, the instruction synchronisation barriers isb and isync, respectively, enforce certain dependency ordering: committing those barriers can only happen when the control flow is resolved and all program-order-earlier memory accesses have their addresses determined. RISC-V decided not to adopt this behaviour: the main arguments against adopting the same semantics here were: (1) hypothetical implementations may handle instruction synchronisation without enforcing this additional ordering; and (2) the instruction synchronisation barrier is likely to be an expensive operation, and so programmers should not be encouraged to use it for other memory ordering purposes (such as the ordering resulting from such dependencies). This means that `fence.i` is a weaker fence than `isync` or `isb`.

Weaker barriers RISC-V has a number of other barriers, some equivalent to those of MCA ARMv8, but additionally also a number of weaker barrier instructions, as well as `fence.tso`. RISC-V has an instruction `fence` that takes two arguments, indicating which program-order-earlier and later instructions the fence creates ordering with: for example, $\texttt{fence}_{\texttt{R,RW}}$ is a fence that creates ordering from program-order-preceding loads (or reads R) to program-order-succeeding loads and stores (or reads and writes RW). The possible arguments for the predecessor and successor set are R, W, and RW. Then the following ARMv8 and RISC-V barriers correspond:

- $\texttt{fence}_{\texttt{RW,RW}}$ – `dmb sy`
- $\texttt{fence}_{\texttt{R,RW}}$ – `dmb ld`
- $\texttt{fence}_{\texttt{W,W}}$ – `dmb st`

RISC-V's `fence` also allows for additional barriers, however, that ARMv8 does not have, such as $\texttt{fence}_{\texttt{R,R}}$, enforcing load-load ordering, and $\texttt{fence}_{\texttt{W,R}}$, ordering earlier stores and later loads. The `fence.tso` fence provides TSO ordering when placed between any two memory accesses.

The addition of these new barriers poses questions concerning their interaction with control dependencies: in the ARMv8 Flat model, committing any barrier is only possible when its control flow is completely determined (when the loads feeding into the register writes affecting program-order-earlier branches are finished); the ARMv8-axiomatic model, on the other hand, does not specify this ordering. But (as shown by the equivalence proof between Flat-axiomatic and ARMv8-axiomatic) this intensional difference between the models does not extensionally make the operational model stronger than the ARMv8-axiomatic model: e.g. the ARMv8-axiomatic model does not explicitly have the ordering ctrl; [dmb.sy]; po; [R|W]; but since it has po;[dmb.sy];po, the choice of whether or not to include this ordering is not observable.

RISC-V's additional fences, however, make this observable, and the Task Group discussed whether control dependencies of barriers should create ordering. The group decided on the more

relaxed variant in which control dependencies create no ordering. This corresponds to allowing the "speculative" execution of barriers in the operational model.

Acquire/release The base RISC-V ISA and the current RISC-V extensions do not have "normal" load acquire and store release instructions; it only supports acquire/release annotations on load reserve/store conditional and atomic memory operations in the A-extension. The memory model, however, is prepared to handle acquire/release accesses in two variants:

- Weak RCpc acquire/release accesses. A weak acquire creates order with all program-order-later memory accesses, a weak release with all program-order-earlier ones; an RCpc store release and subsequent RCpc load acquire are not ordered.
- Strong RCsc acquire/release accesses. A strong acquire access is additionally ordered with all program-order-preceding strong release accesses.

As a result of the base ISA and current extensions lacking non-exclusive load acquire/store release instructions, the group discussed whether for the purpose of the C/C++11 compilation scheme RISC-V assembly should offer "fake" release/acquire instructions, implemented using other instructions. Eventually the group decided on the simpler option of a purely fence-based compilation scheme for the time being.

Syntactic CSR dependencies RISC-V has particular special purpose registers, CSR registers, that keep track of certain status information, including the number of instructions executed so far, timing information, etc. Some of these registers have special semantics, and in many programs write access to these registers is frequent and thus has to be fast. Some discussion in the Task Group revolved around whether syntactic dependencies originating from accesses to CSR registers should create no memory ordering, in order to allow for faster CPU implementations.

This would lead to memory model to allowing problematic out-of-thin-air executions. Consider, for example, the following program, and assume on both threads register `x1` initially holds the address of memory location x and `x2` that of y. In this program, Thread 0 reads x, subsequently

```
lw x3, 0(x1)  ||  lw x3, 0(x2)
CSRW csr, x3  ||  sw x3, 0(x1)
CSRR x3, csr  ||
sw x3, 0(x2)  ||
```

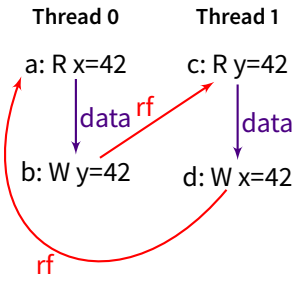


Figure 8.1: LB+datas

writes the returned value to some CSR register `csr` (with `CSRW`) and reads it back into `x3` (with `CSRR`), and writes `x3` to y. Thread 1 reads y and writes what it reads to x. If the memory model does not preserve ordering from data dependencies via the csr register on the first thread, it allows

the execution in which, for example, both loads of the program read 42, such as in Figure 8.1, even though this value is not justified by the code of the program. Another option the Task Group discussed was to preserve CSR dependencies through explicit CSR reads and writes, but to not preserve dependencies resulting from certain implicit CSR accesses, which would lead to similar problems.

Eventually, however, the Task Group decided that, as read access to these registers is typically infrequent, preserving all syntactic CSR dependencies would lead to acceptable constraints on hardware implementations. Since not preserving syntactic dependencies for these registers would likely have rendered equivalent operational modelling infeasible the group eventually decided the memory model should preserve syntactic dependencies via CSR registers, forbidding the above example.

Store conditional dependencies In ARMv8, the architecture intends for the register write of a store exclusive instruction indicating its success not to introduce memory ordering. This results in the operational model deadlocks discussed in Section 6.3.10. Moreover, the fact that these dependencies are not preserved in ARMv8 leads to surprising behaviours, almost resembling the "self-fulfilling" executions that constitute out-of-thin-air behaviour.

Consider, for instance, the following example, due to Andy Wright of the Memory Model Task Group. Thread 0 executes a load reserve to some location z; it then reads x with a plain load and writes the value it read to z using a store conditional; the register write indicating the success or failure of the store conditional is then used in computing the value written by a store to location y. Thread 1 reads y and writes what it read to x. If the memory model does not preserve the ordering resulting from syntactic dependencies via the store conditional's status register (such as is the case for store exclusive instructions for ARMv8), the execution shown in Figure 8.2 is allowed, in which the store conditional's write value depends on its own success.

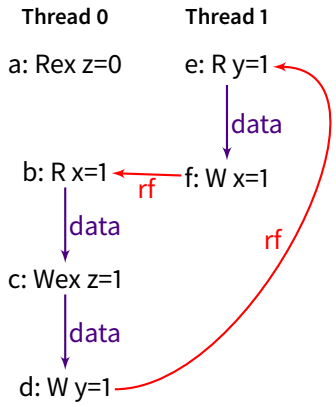


Figure 8.2

In order to prevent these issues, after much discussion RISC-V has adopted a stronger semantics for load reserve/store conditional instructions in which the store conditional's success bit register write creates memory ordering: the status register write of a store conditional instruction — in the successful case — is only made available once the store is propagated to memory. In the case

of a failed store conditional the status register write does not create memory model ordering, corresponding to allowing the store conditional to fail immediately after fetching it.

Atomicity of atomic operations Similar issues as those for ARMv8's exclusive instructions also arise for atomic memory operations. The atomic operations include instructions such as atomic swap and atomic increment; these instructions load a value from a location in memory, if v is the returned value, they perform some arithmetic on v, and write the result to the same location in memory, as well as writing the original (unmodified) load value v to a register.

In the ARMv8-axiomatic model these instructions are treated as generating separate read and write events, that do not have to be adjacent in the memory ordering: in ARMv8, the read of an AMO, for example, can happen before its write, and later instructions depending on the value returned from this read can execute before the AMO write is propagated, as long as it is guaranteed there is no intervening write coherence-between the write the AMO read and the AMO's write. This corresponds to an operational model in which for an atomic operation the model has separate transitions for reading memory and for subsequently writing memory.

This can lead to surprising behaviours, and operational model deadlock issues similar to those arising for load/store exclusive instructions in ARMv8. Consider, for instance, the example in

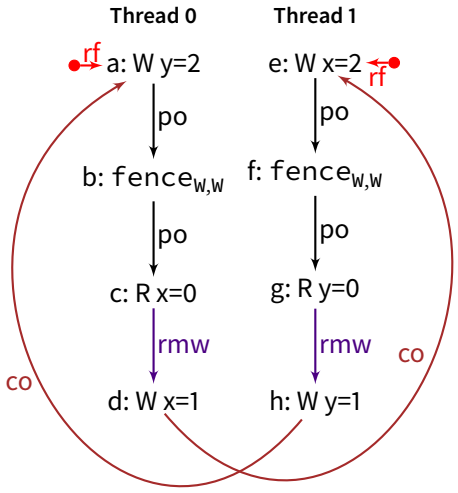


Figure 8.3

Figure 8.3, an adaptation of the example in Figure 6.12 for load/store exclusive instructions in ARMv8. Assume c and d, and g and h, respectively, originate from an atomic swap instruction, indicated here with the rmw edge.

If the reads and writes of an AMO are treated as separate events in the memory model, the following partial execution of the operational model is possible. Since the read part of an AMO should not be affected by the store fence b, the read part of the atomic swap on Thread 0 (c) can be satisfied, from the initial write to x, without propagating the write (d) of the atomic swap. Likewise, satisfy g from the initial write to y without propagating h. Now, due to the atomicity guarantee given by atomic operations, d must become the next write to x, and h the next write to y. Therefore, the model locks locations x and y. To "unlock" x, d has to propagate. The write

d — due to the fence — waits for a to propagate, but a cannot propagate because y is locked. To unlock y, h has to propagate. The write h, in turn, waits for e to propagate, but e cannot propagate because x is locked. Thus, the model is stuck, with the atomic swaps on both threads partially executed and unable to make progress.

In order to avoid deadlock issues of this kind and certain surprising behaviours that would otherwise be allowed by the memory model, RISC-V has adopted a stronger semantics also for atomic operations, in which the read and write of an atomic operation "happen at the same time", thus with a stronger atomicity guarantee.

Ordering of store conditional and atomic operations In addition to the atomicity guarantees, ARMv8 gives ordering guarantees for loads reading from a store exclusive or atomic operation: the ARMv8-axiomatic model has the ordering [range(rmw)]; rfi; [A | Q], effectively preventing thread-internal forwarding from a store exclusive or atomic operation in case the load is a (weak or strong) load acquire. Originally, the draft RISC-V proposal did not have this ordering, making its semantics more relaxed than ARM's and potentially complicating porting code from ARMv8 to RISC-V and the compilation scheme from C/C++11 to RISC-V. After long discussion, the Task Group decided on the simpler and stronger semantics that forbids forwarding from store exclusive and atomic memory operations altogether.

Mixed-size load finishing The task of the group was primarily defining the concurrency semantics for the setting in which all memory accesses have the same size, not the semantics mixed-size accesses. The Flat operational model does handle mixed-size accesses, and so does the RISC-V-axiomatic model if its relations and axioms are interpreted as ranging over byte-sized memory events. These two mixed-size models, however, experimentally mismatch: the condition to *finish* a load instruction in the operational model has some checks necessary in order to prevent coherence violations with respect to program-order-preceding memory accesses. While in the non-mixed-size case this local ordering strength is not observable, in the mixed-size case it is, leading to stronger behaviour in the operational model than in the axiomatic model.

Consider, for instance, the example [due to Shaked Flur] of Section 8.1, and assume big endian format. In this example, Thread 0 reads four bytes at address x, and subsequently writes what it read to location y. Thread 1 stores a four bytes encoding the value 5 at location $y + 4$, reads eight bytes at location y and subsequently four bytes at location $y + 4$, and then uses the value returned from the latter load in calculating the address of a memory write of four bytes to location x encoding the value 1. The particular execution of interest is one where a reads 1 from f, b writes the same value, d reads the most significant four bytes from b and the least significant bytes from c, and e is satisfied by c.

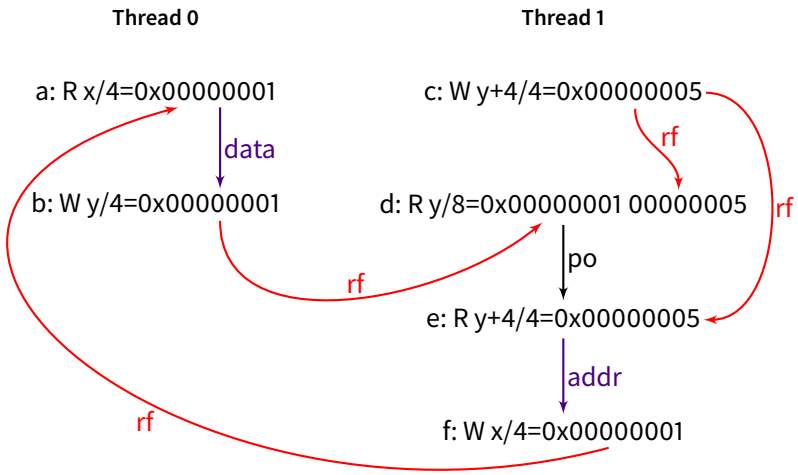


Figure 8.4

Interpreting the current RISC-V axiomatic model as ranging over byte-sized events allows this example. While the data dependency on Thread 0 orders a and b, there is no dependency or direct ordering between d and f: d and e can both read from c "early", by forwarding, and thus resolve the address dependency of f; and so, one may think, f should be permitted to propagate before d has read from b, thus allowing the execution.

In the operational model this execution currently is not allowed. The reason for this is the Flat model's coherence mechanism. Consider the following execution: since c, d, and e have no dependencies, their instructions can do all register reads to determine their address and (in the case of c) data; now c can be forwarded to d and e, leaving the most significant bytes of d unsatisfied but entirely satisfying e; thus e resolves the address dependency of f. But after f has initiated, it cannot commit yet, because it does not have *fully determined* data dependencies — the read of e that feeds into its register reads is not finished yet. In order for e to finish, the Flat model has to ensure that e cannot be forced to restart due to coherence violations in the future. Since, however, the read d overlaps e and is not entirely satisfied yet, the model cannot guarantee this yet, thus preventing e from finishing and f from committing and propagating.

This test equally applies for ARMv8: here, interpreting the ARMv8-axiomatic model as ranging over byte-sized events also allows the test. The issue is currently unresolved. It is not clear yet whether the execution should architecturally be allowed or forbidden. Moreover, both possible answers pose some difficulties in the specification. If the example behaviour is to be forbidden by the architecture, it is not obvious how the axiomatic model should be strengthened to match this choice: currently the only proposal is a complicated addition to the axiomatic model, that specifies the thread-internal ordering with mutually recursive definitions. If the example behaviour is to be allowed, it is not clear how to relax the Flat operational model. It is worth noting, however, that this particular example is one that the Flowing model with Flat topology allows: here, once d is *issued* after d and e have read from c by forwarding, the model can guarantee there will be no coherence violation between d and e, even before d is entirely satisfied.

8.2 RISC-V operationally

Since RISC-V closely follows non-MCA ARMv8, the Flat operational model of Section 6.3 is a good basis for RISC-V as well. Due to the semantic choices concerning the stronger behaviour of store conditional instructions described above, and due to the addition of new barriers, however, it had to be adapted to accurately handle RISC-V. The RISC-V Flat model also contains a semantics for RISC-V's atomic operations (where the ARMv8 Flat model does not yet handle the corresponding ARM operations). After these changes and additions this model is experimentally equivalent to the axiomatic model on a test suite of around 7000 non-mixed-size litmus tests — partly hand-written, mostly systematically generated by diy — to the RISC-V-axiomatic model. For the mixed-size case, there is the aforementioned discrepancy between axiomatic and operational.

One would like to check this equivalence formally, and ideally have a proof of equivalence between operational and axiomatic. The proof of Chapter 7 is not yet adapted to cover RISC-V. As of now, it is not clear how difficult it would be adapting the proof to RISC-V, due to some of the aforementioned subtle differences between ARMv8 and RISC-V. While the proof showing that the operational model allows all the behaviours allowed by the Flat-axiomatic model (adapted to the changes of RISC-V), and the proof that the Operational model satisfies the axiomatic model's axioms should follow along the same lines as the corresponding proofs for ARM, the proof relating Flat-axiomatic and the RISC-V-axiomatic model is less clear, because it hinges on many details of the concurrency models: since RISC-V has preserved dependencies out of store-exclusives and certain fences not present in ARMv8, this has to be checked in detail.

The following gives an overview over the changes and additions to the Flat model in order to also handle RISC-V. The textual description of the full definition can be found in Section E.2 of the appendix, and in Appendix B.3 of the RISC-V ISA manual [121].[1]

Load reserve/store conditional In the case of ARMv8, the operational memory model guarantees that the register write indicating the success of a store exclusive instruction does not generate dependencies due to the ordering of the intra-instruction steps. There, a store exclusive instruction has the following steps:

1. fetch the instruction
2. determine success or failure of store exclusive
3. write the status register indicating success or failure
4. do the register reads necessary to determine the address
5. announce the address/footprint of the store's write
6. do the register reads necessary to determine the data/value of the store instruction
7. create the request to write memory at the previously announced address
8. commit the store (subject to commitment conditions)
9. propagate the write (subject to propagation conditions including atomicity checks for the exclusive write)

[1] This is joint work with Shaked Flur, Peter Sewell, Luc Maranget, Susmit Sarkar, and the Memory Model Task Group, chaired by Daniel Lustig.

10. complete the store
11. finish the store.

Hence, determining success and doing the corresponding register write are the first actions of the instruction after fetching, and following instruction can immediately rely on the success without waiting for the store exclusive's other actions.

For RISC-V's stronger store exclusive semantics the model is adapted to delay the register write in the successful case. Since in the case of failure the register write is not supposed to introduce ordering, the first action after fetching is still determining whether the instruction should fail. If it fails, it writes the register. If not, it proceeds with the actions from (3). Then, at the point of write propagation, the storage subsystem determines whether the store exclusive can in fact become successful. If it fails it writes the register. If not, it propagates the write, and only afterwards writes the register indicating success. Thus, in the RISC-V Flat model, any instruction relying on the success of the store exclusive can only read this register write once the store exclusive has propagated the write into memory.

Section E.1 of the appendix describes further work on the operational model to support atomic memory operations and certain RISC-V barriers that do not have an ARMv8 counterpart.

The Flat model adapted and extended in the ways described above, and with similar adaptations for other barriers, experimentally matches the RISC-V-axiomatic model. The model details can be found in Section E.2 of the appendix. It is included in Appendix B.3 of the RISC-V ISA manual [121]. As of late September 2018, the RISC-V memory model has officially been ratified. The ratification includes the Flat RISC-V operational model and the two variants of the formal axiomatic concurrency model, but as non-normative explanatory material.

Chapter 9

Promising-ARMv8/RISC-V

The Flat model of Chapters 6 to 8 takes advantage of the multicopy atomicity of ARMv8 (and RISC-V) to simplify the Flowing and POP models, expressing the concurrency behaviour purely in terms of the threads' actions. But despite this simplification, the model is still fairly complex. The model intentionally expresses the possible concurrency behaviours in terms of abstractions of real hardware, enabling a clearer relation to micro-architectural implementations. Consequently, understanding the concurrency behaviours of a given program, however, also requires thinking in terms of such abstractions. In Flat, instructions execute *out-of-order*, *speculatively* (the model has explicit branch speculation), and *non-atomically* (with multiple transitions per instruction), and the rules for the ordering of these transitions are intricate. In order to handle branch mispredictions, and coherence violations from speculatively executed instructions, the Flat model sometimes discards or restarts already-executed instructions.

This chapter describes a more abstract operational concurrency model that gives up on the previous operational model's goal to abstract from micro-architectural implementations, with the aim to explain the concurrency behaviour in a way that is simpler and more understandable from a programmer's perspective. This model, Promising-ARM/RISC-V, builds on the work of Kang et al. [71] on the Promising Semantics for C/C++11 concurrency. Using the Promising Semantics' concepts of *views* and *promises*, Promising-ARM/RISC-V explains the concurrency behaviour in a mostly in-order model:[1]

- Loads execute in order; early load satisfaction is captured by allowing reading from older writes in the *write history*.
- Writes can be promised, capturing the out-of-order execution of stores. Every such promise is justified thread-locally and not subject to restarts or discards.
- A concept of views provides coherence guarantees and the ordering guarantees resulting from barriers and dependencies.

The resulting model combines some of the abstractness of the axiomatic models with an operational intuition that *emphasises the execution of threads in program order*, at the cost of a less clear relation to hardware. This model has an abstract state and executes instructions *atomically* and *without branch speculation*, and, with the exception of *write promises*, executes instructions *in order*. As a result, the model does not need instruction restarts or discards for coherence violations or branch mispredictions and offers a simpler way of thinking about the concurrency behaviours. The model is formalised, separately in Lem and Coq.

This model can be made executable, and this chapter also describes a version of this model integrated with ISA models for ARMv8 and RISC-V into rmem. With some simple optimisations it

[1] This chapter describes joint work with Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur, and the text is mostly taken from a joint paper, currently under submission [103].

becomes significantly faster than the Flat model and ARM's axiomatic herd model when tested on a number of standard concurrent datastructure and spinlock examples.

Caveats The model does not currently handle mixed-size accesses. Like the Flat model, the model handles load/store exclusive operations but does not currently handle atomic memory operations/AMOs; it only covers the same ISA fragments as the Flat model, does not deal with system-aspects of the concurrency behaviour, interrupts, or self-modifying code. The ISA model currently also lacks the weaker load acquire LDAPR. In the case of ARMv8, the model can deadlock due to load/store exclusive instructions in a similar way as the Flat model.

For presentation purposes the next section introduces a small language that is used in the definitions of the model in this text and a Coq formalisation. The executable version integrates the Sail ISA models, the Coq model does not.

9.1 Language

Figure 9.1 defines the syntax of this small language. A program is the parallel composition of

p	$::=$	$s_1 \,\|\|\, \ldots \,\|\|\, s_n$	*program*
$s \in \mathrm{St}$	$::=$	$\texttt{skip}$	*statement*
	$\mid$	$s_1; s_2$	*sequential composition*
	$\mid$	$\texttt{if}\ (e)\ s_1\ s_2$	*conditional*
	$\mid$	$\texttt{while}\ (e)\ s$	*loop*
	$\mid$	$r := e$	*register assignment*
	$\mid$	$r := \texttt{load}_{xcl,rk}\ [e]$	*load*
	$\mid$	$r_{\mathrm{succ}} := \texttt{store}_{xcl,wk}\ [e_1]\ e_2$	*store*
	$\mid$	$\texttt{dmb sy} \mid \texttt{dmb st} \mid \texttt{dmb ld} \mid \texttt{isb}$	*ARM barriers*
	$\mid$	$\texttt{fence}_{K_1,K_2} \mid \texttt{fence.tso} \mid \texttt{fence.i}$	*RISC-V barriers*
$r \in \mathrm{Reg}$	$=$	$\mathbb{N}$	*register*
$op \in \mathrm{O}$	$::=$	$+ \mid - \mid \ldots$	*arithmetic ops.*
$e \in \mathrm{Expr}$	$::=$	$v \mid r \mid (e_1\ op\ e_2)$	*pure expression*
$xcl \in \mathbb{B}$	$::=$	$\mathsf{true} \mid \mathsf{false}$	*exclusive or not*
$rk \in \mathrm{RK}$	$::=$	$\mathsf{pln} \mid \mathsf{wacq} \mid \mathsf{acq}$	*read kind*
$wk \in \mathrm{WK}$	$::=$	$\mathsf{pln} \mid \mathsf{wrel} \mid \mathsf{rel}$	*write kind*
$K \in \mathrm{FK}$	$::=$	$\mathtt{R} \mid \mathtt{W} \mid \mathtt{RW}$	*RISC-V fence kind*

Figure 9.1: The language, with ARMv8/RISC-V-like loads, stores, and barriers

statements. Statements include loads, stores, barriers, register assignment, sequential composition, conditionals, and loops. Statements operate on thread-local registers; for simplicity this text assumes infinitely many. A load or store is annotated with (1) a boolean indicating whether it is an exclusive access, and (2) with a read or write kind, respectively indicating whether it is a

plain access or has special acquire or release ordering. A store exclusive writes a bit to a register indicating whether it succeeds or fails. For uniformity of the syntax and the rules, non-exclusive stores also write the success bit to an otherwise unused register, which is omitted in the syntax. Following the ARM ISA, success is indicated by 0 (here called $\mathtt{v}_{\mathrm{succ}}$), and failure by 1 ($\mathtt{v}_{\mathrm{fail}}$). Only store exclusives can fail; non-exclusive stores always succeed. Whenever a load or store command is not annotated with a memory kind, this means a plain load or store; whenever it is not annotated to be exclusive, this means a non-exclusive load or store. So $r := \mathtt{load}\,[e]$ is a plain, non-exclusive load, and $\mathtt{store}\,[e_1]\ e_2$ is a plain, non-exclusive store.[2]

In the model, terms of the shape '$(\mathtt{if}\ (e)\ s_1\ s_2); s_3$' are equivalent to '$\mathtt{if}\ (e)\ (s_1; s_3)\ (s_2; s_3)$': the instructions in s_3 are regarded as control-dependent on expression e. This treatment of control dependencies matches dependencies in machine code, where control flow is not delimited, and is important for the memory ordering resulting from control dependencies.

9.2 Promising-ARM/RISC-V, informally

Promising-ARM/RISC-V expresses the semantics of ARMv8 and RISC-V using four main building blocks, using the concepts from the Promising Semantics for C/C++11 of Kang et al. [71], but adapting them to the specifics of ARMv8 and RISC-V.

1. Memory is the full history of all writes: a list of writes. To account for the effects of the early execution of loads in ARMv8 and RISC-V while still executing programs in order, Promising-ARM/RISC-V allows loads to read from "old" writes in memory. The list records the writes in the order they were propagated in.
2. To handle the early execution of stores, threads can *promise* writes at any time as long as they can later be *fulfilled* by a store instruction executed in program order. A promised write is a write propagated to memory before the execution of the write's thread has reached a store instruction producing this write. The requirement to fulfil the promise constrains the thread's future execution in such a way that it reaches such a store that fulfils the promised writes.
3. To provide the architectural ordering and coherence guarantees, Promising-ARM/RISC-V uses *views* to constrain both reads and writes. Allowing loads to read from arbitrarily old writes will violate the architectural guarantees, for example as given by barriers. Views place a lower bound on how far back in the history of writes a load can read from. Views similarly constrain how "early" certain writes can be promised.
4. To ensure that all promises can be fulfilled, the promises of a thread are *certified* when they are introduced and at any step the thread takes, requiring in any of the thread's steps that the thread remains on an execution path where promise fulfilment can be achieved.

These ideas will be explained in more detail in the following. The full type definitions will be given in Figure 9.2, and the formal definition of the model in Figure 9.3.

[2] RISC-V has a load-reserve strong acquire-release and store-conditional strong acquire-release, in which a single instruction has both strong acquire and strong release ordering combined. For simplicity this text and the Coq formalisation omits these instructions, but the executable model handles them.

For presentation purposes the description uses ARMv8 terminology for barriers: `dmb sy` for full barriers, etc. The following text starts with simple programs using only plain loads and stores and full barriers. It first explains the out-of-order execution of loads and how views constrain them in the view semantics. It then illustrates how the promising semantics extends it to account for the out-of-order execution of stores. Once this core of the model is introduced, it extends the model to other barriers and exclusives, before finally describing how certification avoids executions with unfulfilled promises. For ARMv8 Section F.1 in appendix Chapter F presents an extended certification with a special treatment for load/store exclusive instructions.

9.2.1 View semantics

The view semantics underlying Promising-ARM/RISC-V explains the effects of the out-of-order execution of reads — while executing programs in order — by recording the full write propagation history and allowing reads to read from older writes, and not just the last same-address write [74]. The model state $\langle \vec{T}, M \rangle$ comprises the threads $\vec{T}$, and the memory M, where $\vec{T}$ maps each thread identifier *tid* to the corresponding thread. Each such thread consists of a statement (of type St), a register state, and more components that will be gradually introduced as the text proceeds.

Memory Memory is a list of writes, in the order they were propagated in. A write (message) w, written $\langle x := v \rangle_{tid}$, records the location $w.\mathsf{loc}$ (here x), value $w.\mathsf{val}$ (v), and originating thread identifier $w.\mathsf{tid}$ (*tid*). Initially, memory is the empty list $[]$, which is treated as holding an initial value 0 for all locations. Executing a store generates a write that is appended at the end of memory.

Consider the following `MP` example test, with instruction names a, b, c, d, and e, and comments added for presentation.

$$\begin{array}{l||l} (a)\ \texttt{store}\ [x]\ 37; & (d)\ r_1 := \texttt{load}\ [y];\ //\ 42 \\ (b)\ \texttt{dmb sy}; & (e)\ r_2 := \texttt{load}\ [x]\ //\ 0 \\ (c)\ \texttt{store}\ [y]\ 42 & \end{array}$$

$$r_1 = 42 \wedge r_2 = 0 \text{ allowed}$$

In this test, Thread 0 writes 37 to memory location x, and, after a strong `dmb sy` barrier, writes 42 to y; Thread 1 reads y and then x. In order to focus on capturing the out-of-order execution of loads, the example program has the barrier b between the stores a and c, which forbids their re-ordering. The execution of interest here is one where Thread 1 first reads $y = 42$, and then the initial value $x = 0$. This behaviour is allowed in ARMv8/RISC-V because the (independent) loads on Thread 1 are allowed to execute out of order.

In Promising-ARM/RISC-V, running a, b, c leads to the following steps (b not changing memory):

$$\begin{aligned} \langle \vec{T}, [] \rangle &\overset{(a)}{\Rightarrow} \langle \vec{T}', [\langle x := 37 \rangle_0] \rangle \\ &\overset{(b)}{\Rightarrow} \langle \vec{T}'', [\langle x := 37 \rangle_0] \rangle \\ &\overset{(c)}{\Rightarrow} \langle \vec{T}''', [\langle x := 37 \rangle_0; \langle y := 42 \rangle_0] \rangle \end{aligned}$$

Now d can read $y = 42$. Subsequently, since loads can read not only from the last same-address write, but also older writes in memory or the initial state, e is allowed to read the initial $x = 0$. However, allowing to read from arbitrarily old writes in the write history would violate the architectural ordering and coherence guarantees. The following explains how these are provided using the concept of views.

Views Placing a full barrier `dmb sy` between the loads of Thread 1 orders them and prevents the behaviour where f reads 0 after d reads 42.

$$\begin{array}{l||l} (a)\ \texttt{store}\,[x]\,37; & (d)\ r_1 := \texttt{load}\,[y];\ //\ 42 \\ (b)\ \texttt{dmb sy}; & (e)\ \texttt{dmb sy}; \\ (c)\ \texttt{store}\,[y]\,42 & (f)\ r_2 := \texttt{load}\,[x]\ //\ 0 \end{array}$$

$$r_1 = 42 \wedge r_2 = 0 \text{ forbidden}$$

The model captures this ordering using *views*: (In the following, rules will be labelled "R…" for later reference.)

A *timestamp* $t \in \mathbb{T} = \mathbb{N}$ is a natural number index of a write in the message history or 0, where list indices for memory start from 1 and timestamp 0 indicates the initial writes. A view $v \in \mathbb{V} = \mathbb{T}$ is simply a timestamp and indicates that the write at position v and its predecessors in the message list including the initial writes have been "seen".

In the C11 Promising model views are per location and timestamps for different locations are incomparable, in order to achieve a non-multicopy-atomic semantics. Assume two writes w and w' to two different locations x and y respectively. In a non-multicopy-atomic semantics w and w' can propagate to different threads in different order, as demonstrated by the IRIW+addrs test, for instance. In a multicopy-atomic model, however, there is a total order on the propagation of writes: if w propagates to some thread other than its own before w' does, any thread that "sees" w' must also see w. Consequently, in a multicopy-atomic model timestamps for different locations become comparable, and defining a view to be a single "global" timestamp captures the multicopy atomic propagation of writes.

R1 A view constrains loads: a thread can read from the most recent and older writes, but no older than the view allows — it must not read from writes overwritten by newer "seen" same-address writes.

When executing a load or store i, its *pre-view* is computed. The pre-view forces the execution of i to respect certain constraints, given by ordering from barriers, release/acquire instructions, and dependencies, as well as coherence requirements. In the case of a load, the pre-view constrains what writes it can read from — how far back in the message history it can read. For a store it constrains how "early" its write can be promised.

After executing i, its *post-view* is computed. The post-view of a load or store captures which writes in memory had to have been seen by the thread of i for its execution: for example, if i's post-view is 3, then this means the first three writes in memory had to be visible to its thread in order to execute i in this way. The post-view will then affect certain views in the thread state to constrain program-order-later dependent instructions: for any such instruction j, the pre-view of

j is the maximal post-view of any instruction j depends on, as will become clearer in the following examples.

R2 The post-view of a store is the timestamp of its write message, which is always strictly greater than its pre-view. The post-view of a load is the maximum of its pre-view and a *read-view*. In the first few examples, the read-view is simply the timestamp of the write the load reads from; this will later be refined to handle thread-local *forwarding*. The following will gradually introduce how the pre-view of loads and stores are computed.

Memory barriers Returning to the example, the effects of memory barriers are modelled using views:

R3 Each thread state maintains views $\mathsf{v}_{\mathsf{rOld}}, \mathsf{v}_{\mathsf{wOld}}, \mathsf{v}_{\mathsf{rNew}}, \mathsf{v}_{\mathsf{wNew}} : \mathbb{V}$. Initially, all views are 0.

R4 $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$, respectively, are the maximal post-view of all loads and stores executed so far by the thread.

R5 $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$, respectively, are included in the pre-view of all future loads and stores.

R6 `dmb sy` updates both $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to the maximum of $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$: all future loads and stores program-after the barrier are constrained by the post-views of those program-order-before the barrier. Intuitively, `dmb sy` prevents re-ordering of any load or store before it and that after it, which is the strongest form of barrier. Other fences update these two views in a similar but weaker way.

In the example, after executing a, b, c, the memory is $[1\colon \langle x := 37\rangle_0; 2\colon \langle y := 42\rangle_0]$, (timestamps 1 and 2 added for presentation). The subscript 0 indicates both writes are by Thread 0. Now, if Thread 1 reads 42 and executes the `dmb sy`, it takes the following transitions (just showing the state of Thread 1):

$$\langle \mathsf{v}_{\mathsf{rOld}} = 0, \mathsf{v}_{\mathsf{rNew}} = 0, \ldots\rangle \overset{(d)}{\Rightarrow} \langle \mathsf{v}_{\mathsf{rOld}} = 2, \mathsf{v}_{\mathsf{rNew}} = 0, \ldots\rangle \overset{(e)}{\Rightarrow} \langle \mathsf{v}_{\mathsf{rOld}} = 2, \mathsf{v}_{\mathsf{rNew}} = 2, \ldots\rangle$$

When executing f in the resulting state, f is constrained by view $\mathsf{v}_{\mathsf{rNew}} = 2$. Since all messages are seen with view 2, f must read the last message to location x, which is $\langle x := 37\rangle_0$. (In the previous example without the `dmb sy` the load of y also increased $\mathsf{v}_{\mathsf{rOld}}$ to 2; without the `dmb sy`, however, when executing the load of x, $\mathsf{v}_{\mathsf{rNew}}$ is still 0, and the load of x hence unconstrained and able to read the initial $x = 0$.)

In practice, concurrent ARMv8/RISC-V programs also rely on the ordering resulting from dependencies and on coherence guarantees. The following text discusses how the model handles these.

Address dependencies The next example replaces the `dmb sy` between the loads of Thread 1 with an address dependency from the first load (d) to the second (e). The ordering from the syntactic dependency means that if d reads 42, then e must read 37.

$$\begin{array}{l||l} (a)\ \texttt{store}\,[x]\,37; & (d)\ r_1 := \texttt{load}\,[y];\ //\ 42 \\ (b)\ \texttt{dmb sy}; & (e)\ r_2 := \texttt{load}\,[x + (r_1 - r_1)]\ //\ 0 \\ (c)\ \texttt{store}\,[y]\,42 & \end{array}$$

$$r_1 = 42 \wedge r_2 = 0 \text{ forbidden}$$

Promising-ARM/RISC-V accounts for address dependencies using *register views*:

R7 The register state regs : Reg → (Val × $\mathbb{V}$) of a thread not only maps each register of the thread to a value, but also to an associated view (of type $\mathbb{V}$). The following uses the notation $v@\nu$ for a value-view pair.

R8 When a register is written to by any instruction, it also updates the associated view to specify which writes have to have been seen in order to produce this value. Specifically, for any arithmetic instruction, the view of the output register is the maximum of the views of its input registers. For a load, the output register view is its post-view (that is, the maximum of its pre-view and the read message's timestamp).

R9 Finally, the pre-view of a load or store instruction is the maximal view of its input registers (for loads those involved in the expression computing the address, for stores also that of the data) and the $\mathsf{v}_{\mathsf{rNew}}$ or $\mathsf{v}_{\mathsf{wNew}}$ view, respectively. This will be refined later to handle more dependencies and features such as *weaker barriers*, *release/acquire*, and *exclusives*.

Assuming the previous order a, b, c, when d reads $y = 42$, Thread 1 executes as follows:

$$\langle \mathsf{regs} = \{r_1 \mapsto 0@0, \dots\}, \mathsf{v}_{\mathsf{rOld}} = 0, \mathsf{v}_{\mathsf{rNew}} = 0\rangle$$
$$\overset{(d)}{\Rightarrow} \langle \mathsf{regs} = \{r_1 \mapsto 42@2, \dots\}, \mathsf{v}_{\mathsf{rOld}} = 2, \mathsf{v}_{\mathsf{rNew}} = 0\rangle$$

Now, while $\mathsf{v}_{\mathsf{rNew}} = 0$, the pre-view of e is 2, because r_1 is one of its input registers. Therefore e is constrained by view 2, and thus cannot read the initial value $x = 0$.

Coherence Consider the following example, which adds a later, independent, load f from x on Thread 1. While f is not ordered with d, the execution where d reads $y = 42$, e reads $x = 37$ and f reads $x = 0$ is forbidden, since it violates the principle of coherence: a is coherence-after the implicit initial $x = 0$ in memory. So if e has read $x = 42$, the program-order-later read f must not read the coherence-superseded $x = 0$.

$$\begin{array}{l||l} (a)\ \texttt{store}\,[x]\,37; & (d)\ r_1 := \texttt{load}\,[y];\ \text{// } 42 \\ (b)\ \texttt{dmb sy}; & (e)\ r_2 := \texttt{load}\,[x + (r_1 - r_1)];\ \text{// } 37 \\ (c)\ \texttt{store}\,[y]\,42 & (f)\ r_3 := \texttt{load}\,[x]\ \text{// } 0 \end{array}$$
$$r_1 = 42 \wedge r_2 = 37 \wedge r_3 = 0 \text{ forbidden}$$

To account for the architectural coherence requirements:

R10 Each thread state maintains the coherence view coh : Loc → $\mathbb{V}$. coh maps each location x to the maximal post-view of all loads and stores on x executed so far by that thread.

R11 A load or store on x is constrained not only by its pre-view, but also coherence view $\mathsf{coh}(x)$.

(The Promising-ARM/RISC-V model's coherence view has the same intuition as the *current view* in the Promising C11 model [71] — capturing the writes a thread currently knows about, for each location. The C11 model, however, uses this view not only to enforce coherence but also to implement the semantics of release/acquire accesses and certain fences, which are implemented differently here.)

Since d reads $y = 42$ at timestamp 2, the register view of r_1 is 2, and so is the post-view of e. Thus, after e, the thread state is $\langle \mathsf{coh} = \{x \mapsto 2, \dots\}, \dots\rangle$. Then, although the pre-view of f is 0, f is also constrained by $\mathsf{coh}(x) = 2$, and thus cannot read the initial $x = 0$.

Store forwarding However, while loads from the same location have to respect coherence, they do not have to effectively happen in order. Consider the example below, where Thread 0 is unchanged, but where Thread 1 now contains an earlier read d from y, followed by a write e of 51 to y, before the same basic block of a read f from y followed by a read g from x with an address dependency on f. Assume d reads $y = 42$, and f reads $y = 51$. While g is ordered after f by the address dependency, g is still allowed to read the initial $x = 0$, since f can read from e by forwarding and resolve g's dependency before e and even d.

$$\begin{array}{l||l} (a)\ \texttt{store}\,[x]\,37; & (d)\ r_0 := \texttt{load}\,[y];\ //\ 42 \\ (b)\ \texttt{dmb sy}; & (e)\ \texttt{store}\,[y]\,51; \\ (c)\ \texttt{store}\,[y]\,42 & (f)\ r_1 := \texttt{load}\,[y];\ //\ 51 \\ & (g)\ r_2 := \texttt{load}\,[x + (r_1 - r_1)]\ //\ 0 \end{array}$$

$$r_0 = 42 \wedge r_1 = 37 \wedge r_2 = 0 \text{ allowed}$$

For d to read $y = 42$, Promising-ARM/RISC-V must execute in the order a, b, c, d and then e by coherence, leading to memory $[1{:}\ \langle x := 37\rangle_0; 2{:}\ \langle y := 42\rangle_0; 3{:}\ \langle y := 51\rangle_1]$. If f now were to read $y = 51$ at timestamp 3 then its post-view would become 3, and so would the view of register r_1; this would not allow g to read the initial $x = 0$, since it would be constrained by pre-view 3 due to r_1.

To deal with this behaviour, each thread state records information about the thread's own writes, and the model's definition of the read-view specially handles the case in which a load reads from a write from the same thread to allow it to obtain a smaller post-view than the write's timestamp:

R12 Each thread state has a *forward bank* $\mathsf{fwdb} : \mathsf{Loc} \to \langle \mathsf{time} : \mathbb{T}, \mathsf{view} : \mathbb{V}, \mathsf{xcl} : \mathbb{B}\rangle$ holding, for each location x, information about the last write to x propagated by this thread. Specifically:

R13 Every time a thread executes a store to x, it updates the forward bank entry $\mathsf{fwdb}(x)$ to record the timestamp of the written message (time), the maximal view of the store's input registers (view), and whether it was a write exclusive (xcl). Since a store's input registers are used to compute the address and data of the store, view captures its address and data dependencies.

R14 Initially, fwdb is set to $\langle \mathsf{time} = 0, \mathsf{view} = 0, \mathsf{xcl} = \mathsf{false}\rangle$.

R15 The read-view of a load on some location x is refined as follows: if the read message's timestamp is the same as $\mathsf{fwdb}(x).\mathsf{time}$ (*i.e.* the load reads the last write at x by its thread), its read-view is the associated forward view $\mathsf{fwdb}(s).\mathsf{view}$. Otherwise, it is the read message's timestamp, as before. Since the post-view of a load is the maximum of its pre-view and read-view, this means that when a load reads from a forwarded write, the post-view contains the address and data dependencies of the write instead of its timestamp. The role of the xcl bit will be explained in Section 9.2.4.

In the example above, d reads $y = 42$ at timestamp 2, updating $\mathsf{coh}(y)$ to 2; e writes $y = 51$ at timestamp 3, updating $\mathsf{coh}(y)$ to 3 and $\mathsf{fwdb}(y)$ to $\langle \mathsf{time} = 3, \mathsf{view} = 0, \mathsf{xcl} = \mathsf{false}\rangle$, since e has no input register; f reads $y = 51$ at timestamp 3, with pre-view 0, read-view 0 and post-view 0, since the forward view of the write $y = 51$ is $\mathsf{fwdb}(y).\mathsf{view} = 0$, thereby setting the view of r_1 to

0; finally, g reads the initial $x = 0$ with pre-view 0, since its sole input register, r_1, has view 0.

It is important to note that, as seen in this example, in general the coherence view $\mathsf{coh}(x)$ on a location x is never merged into any other views such as pre-views, post-views and register-views, so that its effect is limited to loads and stores on location x only.

9.2.2 Promising semantics

So far the model has only explained the out-of-order execution of loads and how it is constrained with views. The out-of-order execution of stores in ARMv8 and RISC-V is handled in Promising-ARM/RISC-V by adding the notion of *promises* on top of the view semantics presented before. In the following program, Thread 0 reads from x, and writes the value it reads to y; Thread 1 reads from y, and writes 42 to x. In the model as described so far, it would not be possible for both a and c to read values different from 0: at least one of a and c would have to have executed first in the initial memory, and therefore would have to read 0.

$$\begin{array}{l||l} (a)\ r_1 := \texttt{load}\ [x];\ //\ 42 & (c)\ r_2 := \texttt{load}\ [y];\ //\ 42 \\ (b)\ \texttt{store}\ [y]\ r_1 & (d)\ \texttt{store}\ [x]\ 42 \end{array}$$
$$r_1 = r_2 = 42 \text{ allowed}$$

But, since d is independent of c, it is allowed to execute early, and so both a and c can read 42, which corresponds to an execution order d, a, b, c.

Promises To handle the out-of-order execution of writes, the model uses the concepts of promises and promise fulfilment.

- **R16** Each thread state maintains a set of timestamps $\mathsf{prom} : \mathsf{set}\ \mathbb{T}$, called its *promise set*, which records the timestamps of the outstanding promised writes of the thread.
- **R17** A thread with ID *tid* is allowed to *promise* a write $x = v$, which appends $\langle x := v \rangle_{tid}$ to memory and adds the timestamp t of the write message $\langle x := v \rangle_{tid}$ to prom, but does not otherwise change the thread state. As far as other threads are concerned, this write is no different from any other write in memory (prom is thread-local information).
- **R18** The thread is required to *fulfil* this promise $x = v$ at timestamp t at a later stage by executing a store instruction, removing the promise from prom. Specifically, the store should generate a write $x = v$ with the extra condition that its pre-view and the coherence view $\mathsf{coh}(x)$ are strictly smaller than the promise timestamp t.
- **R19** The execution of a write is split into a promise and its fulfilment. A normal write that is not executed early is accounted for by promising it just before a store fulfils it. Note that the timestamp of a write is always bigger than its pre-view because it is appended at the end with a fresh timestamp and immediately fulfilled.

The pre-view of a store essentially constrains promises by constraining the fulfilment: a promise cannot be made "too early", because it cannot be fulfilled if its timestamp is not strictly larger than its pre-view. With this rule added to the underlying view semantics, the following shows how this notion of promise captures the out-of-order execution of stores in ARMv8/RISC-V.

Out-of-order execution of writes The behaviour in the example above that motivated promises is explained as follows. Thread 1 first promises the write $x = 42$ at timestamp 1, resulting in promise set $\mathsf{prom} = \{1\}$ and memory $[1\colon \langle x := 42\rangle_1]$. In Thread 0, a can read $x = 42$ from memory at timestamp 1 and write $y = 42$ (by a normal write), resulting in memory $[1\colon \langle x := 42\rangle_1, 2\colon \langle y := 42\rangle_0]$. Then, in Thread 1, c can read $y = 42$ from memory, and d can fulfil the promise $x = 42$ at timestamp 1, yielding $\mathsf{prom} = \{\}$; d's pre-view and $\mathsf{coh}(x)$ are 0, so strictly smaller than the promise timestamp 1, as required.

Memory barriers Placing a barrier on Thread 1 prevents this execution.

$$\begin{array}{l||l} (a)\ r_1 := \texttt{load}\ [x];\ //\ 42 & (c)\ r_2 := \texttt{load}\ [y];\ //\ 42 \\ (b)\ \texttt{store}\ [y]\ r_1 & (d)\ \texttt{dmb sy}; \\ & (e)\ \texttt{store}\ [x]\ 42 \end{array}$$

$r_1 = r_2 = 42$ forbidden

Promising-ARM/RISC-V handles this using views. As before, consider the state after Thread 1 promising $x = 42$ and Thread 0 executing a, b, resulting in memory $[1\colon \langle x := 42\rangle_1, 2\colon \langle y := 42\rangle_0]$. Here, c is not allowed to read $y = 42$, because it would not be able to fulfil the promise at timestamp 1. Suppose c does read $y = 42$ at timestamp 2. Then Thread 1 has $\mathsf{v}_{\mathsf{rOld}} = 2$ after c and $\mathsf{v}_{\mathsf{wNew}} = 2$ by $\texttt{dmb sy}$ after d. Then the pre-view of e is 2 due to $\mathsf{v}_{\mathsf{wNew}}$, which is not smaller than the promise timestamp 1. If, instead, c reads the initial $y = 0$, e can fulfil the promise.

Coherence Also, replacing d by $r_3 := \texttt{load}\ [x + (r_2 - r_2)]$ constrains e by coherence and forbids the same behaviour. Suppose executing up to c as before. Since c reads $y = 42$ and thus r_2 holds 42@2, d is constrained by pre-view 2 and must read $x = 42$ at timestamp 1. Now although r_2 and r_3 are not used by e, e still cannot fulfil its promise of $x = 42$ at timestamp 1 since d updated $\mathsf{coh}(x)$ to its post-view 2 and e is constrained by $\mathsf{coh}(x) = 2 \not< 1$.

Address and data dependencies Replacing the barrier in Thread 1 by a dependency from the load to the store — whereby the address or data computation of the store involves the return value of the load — also prevents the behaviour.

$$\begin{array}{l||l} (a)\ r_1 := \texttt{load}\ [x];\ //\ 42 & (c)\ r_2 := \texttt{load}\ [y];\ //\ 42 \\ (b)\ \texttt{store}\ [y]\ r_1; & (d)\ \texttt{store}\ [x + (r_2 - r_2)]\ 42 \end{array}$$

$r_1 = r_2 = 42$ forbidden

Similar to before, consider the execution in which Thread 1 promises $x = 42$ at timestamp 1; a reads $x = 42$; b writes $y = 42$ at timestamp 2; and c reads $y = 42$, thereby setting r_2's register view to 2. Here d's pre-view includes r_2's register view 2, and so d cannot fulfil the promise at timestamp 1. Changing d to '$\texttt{store}\ [x]\ (42 + (r_2 - r_2))$' leads to the same behaviour.

Control and address-po dependencies While loads are allowed to be executed speculatively past conditional branches, stores are not. Stores wait for control dependencies to be resolved and, similarly, for the address of all program-order-earlier memory accesses to be determined (*address-po* dependency). Changing d in the previous example to a conditional branch depending

on c's return value also prevents promising e early: the behaviour in which both a and c read 42 is forbidden in the example below, due to the control flow dependency of the store on c.

$$\begin{array}{l||l} (a)\ r_1 := \texttt{load}\ [x];\ //\ 42 & (c)\ r_2 := \texttt{load}\ [y];\ //\ 42 \\ (b)\ \texttt{store}\ [y]\ r_1 & (d)\ \texttt{if}\ ((r_2 - r_2) = 0) \\ & (e)\quad \texttt{store}\ [x]\ 42 \end{array}$$
$$r_1 = r_2 = 42 \text{ forbidden}$$

To capture control and address-po dependencies, the model introduces another view, $\mathsf{v}_{\mathsf{CAP}}$.

R20 Each thread state has an additional view $\mathsf{v}_{\mathsf{CAP}} : \mathbb{V}$, initially set to 0.

R21 Every time a thread executes a conditional branch, the maximal view of the branch's input registers is merged into $\mathsf{v}_{\mathsf{CAP}}$. Similarly, when a load or store is executed, the maximal view of the input registers used to compute the address is merged into $\mathsf{v}_{\mathsf{CAP}}$.

R22 Finally, the pre-view of a store instruction is refined to include $\mathsf{v}_{\mathsf{CAP}}$ (*i.e.* the pre-view is the maximal view of the input registers and $\mathsf{v}_{\mathsf{wNew}}$ and $\mathsf{v}_{\mathsf{CAP}}$).

Assume again an execution in which $x = 42$ is promised at timestamp 1 by Thread 1, a reads $x = 42$, b writes $y = 42$ at timestamp 2, and c reads $y = 42$ thereby setting r_2's view to 2. Then d merges r_2's view (*i.e.* 2) into $\mathsf{v}_{\mathsf{CAP}}$ since r_2 is used to compute the branch condition. In case d is $\texttt{store}\ [z + (r_2 - r_2)]\ 0$, register r_2's view is also merged into $\mathsf{v}_{\mathsf{CAP}}$ since r_2 is used to compute the address. Then e's pre-view includes $\mathsf{v}_{\mathsf{CAP}} = 2$, and thus e cannot fulfil the promise at timestamp 1.

Replacing d by an address-dependent load or store to an otherwise unused memory location z (e.g. $\texttt{store}\ [z + (r_2 - r_2)]\ 0$) introduces the same ordering and also forbids the behaviour.

9.2.3 Release/acquire accesses and weak barriers

The additional barriers of ARMv8 and RISC-V are handled similarly to `dmb sy`. The RISC-V equivalent of `dmb sy`, `dmb ld`, and `dmb st` are $\texttt{fence}_{\mathsf{RW,RW}}$, $\texttt{fence}_{\mathsf{R,RW}}$, and $\texttt{fence}_{\mathsf{W,W}}$, respectively. `isb` has no equivalent in RISC-V: `fence.i` does not consider control and address-po dependencies. Since the model does not handle self-modifying code, `fence.i` is a no-op in this model. RISC-V has some additional barriers, such as $\texttt{fence}_{\mathsf{W,R}}$ and `fence.tso`, which work analogously to the ARMv8 barriers presented so far (see Section 9.3).

Release/acquire Recall the earlier MP example. Turning b into a release write orders b after a. Making c an acquire load orders d after c. The ordering provided by the release and the acquire makes the behaviour in which c reads 42 and d 0 forbidden.

$$\begin{array}{l||l} (a)\ \texttt{store}\ [x]\ 37; & (c)\ r_1 := \texttt{load}_{\mathsf{acq}}\ [y];\ //\ 42 \\ (b)\ \texttt{store}_{\mathsf{rel}}\ [y]\ 42 & (d)\ r_2 := \texttt{load}\ [x]\ //\ 0 \end{array}$$
$$r_1 = 42 \wedge r_2 = 0 \text{ forbidden}$$

Assume Thread 0 promises $y = 42$ before $x = 37$, at timestamp 1. Executing a places $x = 37$ at timestamp 2, and sets $\mathsf{v}_{\mathsf{wOld}} = 2$.

R23 A store release includes in its pre-view the view of all previous memory accesses, captured by $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$.

Therefore, after a, the pre-view of b is 2, and it cannot fulfil the promise at timestamp 1. So, in the example, when c reads $y = 42$, memory must instead be $[1{:}\ \langle x := 37\rangle_0, 2{:}\ \langle y := 42\rangle_0]$ thereby setting c's post-view to 2.

R24 The load acquire, symmetrically to the store release, merges its post-view into $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$, affecting the pre-view of all future loads and stores.

Therefore, c sets $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to 2. Since in this state d is constrained by timestamp $\mathsf{v}_{\mathsf{rNew}} = 2$ and the initial $x = 0$ is superseded by the write at timestamp $1 \leq 2$, the behaviour where d reads $x = 0$ is forbidden.

In addition, ARMv8/RISC-V enforces ordering from strong store releases to program-order-later strong load acquires. (All ARMv8 release stores are "strong" store releases, only RISC-V distinguishes between weak and strong.) To model this ordering:

R25 The thread state maintains a view $\mathsf{v}_{\mathsf{Rel}} : \mathbb{V}$ containing the maximal post-view of all strong releases executed so far.

R26 The pre-view of any later strong load acquire includes $\mathsf{v}_{\mathsf{Rel}}$, enforcing the memory ordering.

The rules for barriers follow the same principle:

R27 `dmb st` updates $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{wOld}}$.

R28 `dmb ld` updates $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{rOld}}$.

R29 `isb` updates $\mathsf{v}_{\mathsf{rNew}}$ to include $\mathsf{v}_{\mathsf{CAP}}$.

9.2.4 Load/store exclusive instructions

The previously discussed instructions can only introduce intra-thread ordering. Exclusive instructions (called load reserve/store conditional in RISC-V) make it possible to provide inter-thread atomicity guarantees. If a load exclusive a and a store exclusive b are paired (*i.e.* there are no other load/store exclusive instructions program-order-between a and b) and the store exclusive b is successful, then it is guaranteed that no writes from other threads to the same address went into memory after the write a read from and before the write of b.

$$\begin{array}{l|l} (a)\ r_1 := \mathtt{load}_{\mathsf{ex}}\ [x];\ //\ 37 & (c)\ \mathtt{store}\ [x]\ 37; \\ (b)\ r_2 := \mathtt{store}_{\mathsf{ex}}\ [x]\ 42 & (d)\ \mathtt{store}\ [x]\ 51; \\ & (e)\ r_3 := \mathtt{load}\ [x]\ //\ 42 \end{array}$$

$$r_1 = 37 \wedge r_2 = \mathsf{v}_{\mathsf{succ}} \wedge r_3 = 42 \text{ forbidden}$$

In this example, if a reads $x = 37$ from c, and b succeeds, then the write $x = 51$ by d is not allowed to come between the writes of c and b, and memory is not allowed to be $[1{:}\ \langle x := 37\rangle_1, 2{:}\ \langle x := 51\rangle_1, 3{:}\ \langle x := 42\rangle_0]$. (However, writes to different addresses are allowed to enter memory between the writes of c and b; and, as discussed in Section 6.3.7, if Thread 0 had other (non-exclusive) stores to x in-between a and b, b would also be allowed to succeed with those po-intervening writes sequenced between the writes of c and b in memory.) A store exclusive is only allowed to be paired with the most recent program-order-earlier load exclusive (whether to the same location or not), and only if there has been no interposing (successful or unsuccessful) store exclusive (independent of their locations).

To capture the pairing of load exclusives and store exclusives:

R30 Each thread maintains an *exclusives bank* xclb : option $\langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V} \rangle$, initially set to *none*, containing information about the last load exclusive when there has been no other store exclusive in that thread since then. Specifically,

R31 xclb is set to $\langle \mathsf{time} = t; \mathsf{view} = \nu \rangle$ (here and in the following, *some* is omitted for simplicity) whenever a load exclusive reads from timestamp t with post-view ν.

R32 xclb is set to *none* whenever a store exclusive (successful or not) is executed.

Consider an execution of the previous example, where c writes $x = 37$ at timestamp 1 and a reads the write $x = 37$ and sets xclb to $\langle \mathsf{time} = 1, \mathsf{view} = 1 \rangle$. Now if d writes $x = 51$ at timestamp 2, b cannot write to x exclusively and must fail by the following rule:

R33 A store exclusive to location z at timestamp t succeeds only if xclb is not *none* and additionally: in case the message at xclb.time is also z (so the load exclusive was to the same location) every message to z in memory between xclb.time and t is written by this thread.

R34 When the store exclusive succeeds it writes to a register indicating its success. The associated view in the success case is its post-view in RISC-V, and 0 in ARMv8. This means in RISC-V if another write depends on the success of a store exclusive, this write can only be promised after that of the store exclusive. In contrast, in ARMv8 this ordering is not preserved. (Since in ARM dependencies from store exclusives do not create ordering, the Promising ARM model can deadlock, in a similar way as the Flat model, as will be discussed in Section 9.2.5. Section F.1 in appendix Chapter F discusses a possible solution.)

Specifically, in Thread 0, xclb.time is 1 and d should write $x = 42$ at timestamp 3, but then the write $x = 51$ in the middle is written by Thread 1, which violates the above rule. Thus in order for b to be successful, b should be executed before d, resulting in memory $[1\colon \langle x := 37\rangle_1, 2\colon \langle x := 42\rangle_0, 3\colon \langle x := 51\rangle_1]$ after d. Then e is constrained by $\mathsf{coh}(x) = 3$, which is due to d, and thus should read 51.

In addition to this atomicity guarantee, exclusives provide some ordering guarantees. Whereas, for example, plain loads can read from plain stores by thread-internal forwarding, potentially acquiring a smaller post-view than the store they read from, the architectures guarantee that certain loads — load acquires in ARMv8, all loads in RISC-V — cannot read from a store exclusive by thread-internal forwarding. (This corresponds to Flat's forwarding transition explicitly excluding the forwarding of exclusive writes to load acquires in ARMv8, see Section 6.3.8.) To capture this:

R35 Recall that the forward bank fwdb : Loc → option $[\mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}; \mathsf{xcl} : \mathbb{B}]$ records in the xcl field whether the write in the forward bank is an exclusive write. The model then prevents a load acquire in ARM, and any load in RISC-V, from a location z from obtaining the smaller forward view fwdb(z).view if fwdb(z).xcl is set.

RISC-V additionally guarantees ordering of the store exclusive with the paired load-exclusive even if the load and the store are to different addresses.[3] To this end:

R36 Recall that the exclusives bank xclb : option $\langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V} \rangle$ records the post-view of the load exclusive in the view field. In RISC-V, this view xclb.view is included in the paired store exclusive's pre-view.

[3] In the case of ARMv8 the architecture specifies "constrained unpredictable" behaviour; this is still being clarified.

9.2.5 Certification

Our description so far has focussed on the thread steps and has assumed *consistent traces*, traces in which all promises are fulfilled. Indeed, the semantics given by traces of these thread steps, restricted to consistent traces, precisely models the legal behaviours of ARMv8/RISC-V (*i.e.*, is equivalent to the axiomatic model): (combining Theorems 4 and 5).

However, we have not yet discussed how the model ensures threads only take consistent steps (steps of consistent executions). Rather than merely discarding traces with unfulfilled promises at the end of the execution, directly preventing inconsistent thread steps is desirable for two reasons: (1) The model works incrementally in terms of thread-local conditions, thereby also improving interactive exploration. (2) It removes unnecessary non-determinism resulting from promises that eventually are unfulfilled, for executability and exhaustive exploration.

We now show how such inconsistent thread steps can be prevented. To this end, we first define what it means for a thread to *execute sequentially*. It means that the thread executes alone (no other threads executing) and every new promise is immediately followed by its fulfilment (effectively doing all writes in program order) [71]. The model then prevents inconsistent thread steps using a simple, thread-local definition of *certification*, allowing any given thread step only if it leads to a *certified thread configuration*.

R37 A *thread configuration* $\langle T, M\rangle$, consisting of a thread state T and memory state M is *certified* if there exists a sequential execution from $\langle T, M\rangle$ to another thread configuration $\langle T', M'\rangle$ such that T' has no outstanding promises.

Restricting thread steps to steps certified as above, is *sound*, not preventing any consistent executions (See Theorem 5). For RISC-V, this definition is also *precise*, preventing any inconsistent executions (Theorem 6). In ARMv8, however, it is not precise: since in ARMv8 syntactic dependencies out of the store exclusive are not preserved the model can deadlock if a thread makes an assumption about the success of a store exclusive that later turns out to conflict with another thread's write. This will be detailed later.

Thread-local certification The above definition of certification, which is based only on the current thread state and the memory, is sufficient to preserve global consistency of machine states, without preventing executions that lead to consistent machine states (and which therefore should be allowed). Since the threads communicate by reading and writing memory one might imagine certification of a thread having to take this interaction with other threads into account. Informally speaking, the reason this is not the case is that "one thread cannot break another thread's promises" [compare 71, Section 3], and that "one thread cannot help another thread fulfil its promises" [compare 71, Section 1].

"One thread cannot break another thread's promises" Consider the following program.

$$\begin{array}{l} (a)\ r_1 := \texttt{load}\,[x]; \\ (b)\ \texttt{store}\,[x]\,42; \\ (c)\ \texttt{store}\,[y]\,(r_1 + 37) \end{array} \Bigg\| \begin{array}{l} (d)\ \texttt{store}\,[x]\,51; \\ (e)\ \texttt{store}\,[y]\,63 \end{array}$$

In the initial state, Thread 0 is allowed to promise $y = 37$ at timestamp 1 (leading to memory

$[1\colon \langle y := 37\rangle_0]$ and adding 1 to prom) since it is certified after the promise: there is an execution of Thread 0 alone under this memory in which a reads $x = 0$, b writes $x = 42$ at timestamp 2 and c fulfils the promise $y = 37$ at timestamp 1.

To see why certification of Thread 0 cannot be broken by other threads, suppose, for example, that Thread 1 executes d and e, leading to memory $[1\colon \langle y := 37\rangle_0, 2\colon \langle x := 51\rangle_1, 3\colon \langle y := 63\rangle_1]$. With this memory Thread 0's state is still certifiable as before:

- Since all views in Thread 0's state are unaffected by Thread 1's execution, a can still read $x = 0$ at timestamp 0 in the same way as in the previous certification.
- While b has to write $x = 42$ at timestamp 4, a different timestamp from that in the previous certification, c can still fulfil the promise $y = 37$ at timestamp 1 because c's pre-view and $\mathsf{coh}(y)$ are still 0.

Since the views precisely track the dependencies of a store (with the exception of the case of ARMv8 store exclusives), the only writes a promise can depend on must already be in memory at the time of promising. (Otherwise, by view constraints in the initial certification, the promise fulfilment would have failed, disallowing the promise.) In the example, since b is not in memory at the time of c's promise, Thread 0's ability to fulfil c's promise is independent of the how b executes, and so of how b's views change during the certification due to the interference by other threads.

"One thread cannot help another thread fulfil its promises" Now the opposite: due to the way dependencies introduce memory ordering in ARMv8 and RISC-V a thread cannot be "helped" in fulfilling its promises. Consider another example where in Thread 0 a loads x, b writes to y what a read, while in Thread 1 c reads y, and if it reads 42, e writes 42 to y.

$$\begin{array}{l|l} (a)\ r_1 := \texttt{load}\,[x]; & (c)\ r_2 := \texttt{load}\,[y]; \\ (b)\ \texttt{store}\,[y]\ r_1 & (d)\ \texttt{if}\ (r_2 = 42) \\ & (e)\quad \texttt{store}\,[x]\ 42; \end{array}$$

Consider a promise $x = 42$ from Thread 1 in the initial state. Certification does not allow this promise: e can only execute and produce this write if c reads $y = 42$. But in the initial memory there is no such write $y = 42$.

This example program might be taken to suggest the above certification definition was insufficient, since the following *hypothetical* execution could allow Thread 1 to fulfil the promise $x = 42$ with the help of Thread 0, leading to outcome $r_1 = r_2 = 42$: first promise $x = 42$ in Thread 1, then read $x = 42$ with a and write $y = 42$ with b in Thread 0, and finally in Thread 1 read $y = 42$ with c satisfying d's branch condition and allowing e to fulfil the promise — and thereby Thread 0 "helping" Thread 1 fulfil its promise, and implying a more sophisticated certification may be necessary. Since, however, in ARMv8 and RISC-V dependencies create memory ordering, behaviours of this kind are not possible. In this particular hypothetical execution, Thread 1's promise of $x = 42$ in the initial state leads to memory $[1\colon \langle x := 42\rangle_1]$ and $\mathsf{prom} = \{1\}$; executing b afterwards as above to memory $[1\colon \langle x := 42\rangle_1; 2\colon \langle y := 42\rangle_0]$; now when c reads $y = 42$ from this second write it updates r_2's view to its post-view 2; d's branch condition depends on r_2 and

so d updates $\mathsf{v}_{\mathsf{CAP}}$ to 2; since in the next step e's pre-view is constrained by $\mathsf{v}_{\mathsf{CAP}} = 2$ Thread 1 cannot fulfil promise 1: $\langle x := 42 \rangle_1$.

More generally, when some thread i promises a new write p, the fulfilment of p cannot depend on another thread: any write to memory w by another thread after the promise of p must have a timestamp greater than p. Since p's fulfilment cannot depend on any write with timestamp greater than p, no such write w can "help" in fulfilling p.

Load/store exclusive issues In this context, load/store exclusive instructions are special [compare 71, Section 3]. When considering non-exclusive instructions, the behaviour of a thread is monotonic in the contents of memory in the following sense [compare 71, Section 3]: in any given thread state T and memory state M, the set of writes the thread can produce executing sequentially from thread state T under memory M is a subset of the writes it can produce executing sequentially from thread state T under some memory M' that extends M with more writes; this is not true when considering load/store exclusive instructions.

Consider, for instance Thread 0 in the following program Kang et al. [compare 71, Section 3].

$$\begin{array}{l} (a)\ r_1 := \mathtt{load}_{\mathsf{ex}}\,[x]; \\ (b)\ r_2 := \mathtt{store}_{\mathsf{ex}}\,[x]\ 1; \\ (c)\ \mathtt{store}\,[y]\ (63 + r_1 + r_2) \end{array} \Bigg\| \begin{array}{l} (d)\ \mathtt{store}\,[x]\ 5; \end{array}$$

In the initial state, Thread 0 could read 0 with a, do a successful write-exclusive with b (setting r_2 to 0), and then write $y = 63$ with c. Hence, one of the writes it can produce is $y = 63$.

If, however, d executed and extended the memory state with $x = 5$ that execution would no longer be possible. The load a would still be able to read the initial $x = 0$, but then b would not be able to succeed, since it would have to place its write after d's in memory — breaking the atomicity guarantee of store exclusives. So should Thread 0 be allowed to promise c's write $y = 63$ in the initial state? In order to match the architecturally intended semantics, the answer differs for ARMv8 and RISC-V.

The issue in this example arises from the syntactic dependency from b to c: for c to write 63, b must succeed. In ARMv8 this syntactic dependency does not create memory ordering, and the architecture allows executing c's write before b's. Hence, Promising-ARM allows promising c's write before b's, to match ARM's axiomatic model. As a result of this, however, the model can deadlock: assume Thread 0 promises c's write $y = 63$ before doing b's write; then in the following state, Thread 1 can execute d and Thread 0 will not be able to fulfil its promise of $y = 63$ — Thread 1 breaks Thread 0's promise, and the model gets stuck with an unfulfilled promise. Section F.1 of the appendix explains these issues in more detail and explores a possible extension of the Promising-ARM model with locks and more elaborate certification to prevent such model deadlocks. (This extension is currently not covered by the formalisations.)

In RISC-V, on the other hand, the dependency creates ordering from b to c: when b executes, it sets the register view of r_2 to its post-view. This means, in the initial state, c cannot be promised; it can only be promised after promising the write of b. As a result of RISC-V's stronger ordering, the Promising-RISC-V model does not have Promising-ARM's deadlock problems.

In the RISC-V case, Promising behaves similarly to the Promising C11 model in forbidding the early promise of c's write. The Promising C11 model's language does not have exclusive instructions but the similar atomic updates, such as compare-and-swap and fetch-and-add [71, Section 3]. Assume replacing a and b in the above example with a fetch-and-add instruction (ab) $r_1 := \mathsf{FAA}\ x\ 1$ that reads x, saves the returned value in r_1, writes that value incremented by 1 to x, and provides the same atomicity guarantees as load/store exclusive pairs in ARMv8 and RISC-V [71, Section 3] (and always succeeds). Then Promising C11 also prevents the promise of c before ab. The way in which it forbids this, however, is different. Instead of forbidding it on the basis of the syntactic dependency tracking, it forbids the early promise of c by future memory quantification: in the Promising C11 model, a thread is only allowed to make a promise if it is able to fulfil this promise *in any future memory*, so any memory extending the current memory with arbitrary writes [71, Section 3]; in some future memory there might be other writes to x (after the initial one) and the fetch-and-add ab cannot fetch 0, because it would then fail to provide its atomicity guarantee; hence, c's write cannot be promised until after the promise of ab's write. The Promising C11 model allows writes to be promised in a way that enforces adjacency of the promised write to its preceding same-address write (using time intervals), which then guarantees no write can become coherence-ordered-between the two writes [71, Section 3]. After promising ab in this way, future memory quantification allows the promise of c.

Algorithm

The above definition of certification provides a simple executable check for whether a thread configuration (TC) is certified. However, the executable tool of Section 9.5 has to be able to compute for any given thread state *which* promises should be allowed: which promises lead to such certified configurations. For the sake of the executable model, we give an equivalent definition, called find_and_certify that we proved correct in Coq (Theorem 7), and which is the basis for the algorithm used by the executable model.

Given a thread ID *tid* and a TC $\langle T, M\rangle$, we define find_and_certify *tid* $\langle T, M\rangle$ (loc, val) to hold for a location *loc* and value *val* if thread *tid* can execute sequentially from this TC to a TC in which all its promises are fulfilled, and if during this execution it can produce a write $loc := val$ such that the pre-view of this write and coherence-view (at the write location) at the time of writing are less than or equal to the maximal timestamp in the memory M.

More precisely, find_and_certify *tid* $\langle T, M\rangle$ (loc, val) holds if there exists a pre-view ν_{pre} such that:

- *tid* can execute sequentially from TC $\langle T, M\rangle$ to some TC $\langle T_2, M_2\rangle$;
- *tid* can step from TC $\langle T_2, M_2\rangle$ to some TC $\langle T_3, M_3\rangle$ by promising a write at location *loc* of value *val* with pre-view ν_{pre};
- *tid* can step from TC $\langle T_3, M_3\rangle$ to TC $\langle T_3', M_3\rangle$ by (immediately) fulfilling this promise; and
- *tid* can execute sequentially from TC $\langle T_3', M_3\rangle$ to some TC $\langle T_4, M_4\rangle$, such that:
- the promise set of T_4 is empty;
- the coherence view of T_2 at location *loc* is less than or equal to the maximal timestamp in memory M; and

- v_{pre} is less than or equal to the maximal timestamp in memory M.

The intuition behind this is the following: in order for a promise p to be allowed in TC $\langle T, M\rangle$, the TC reached after making that promise, $\langle T', M'\rangle$ (where T' is the same as T with p added as a promise), must be certified: there has to exist a sequential execution from $\langle T', M\rangle$ to a TC with empty promise set. During this sequential execution, in order to fulfil the promise p, *tid* has to produce a write matching p, and at that point pre-view and coherence view have to be strictly less than the timestamp of p in memory. The timestamp p will receive if promised in $\langle T, M\rangle$ is the maximal timestamp in memory M incremented by one. Hence, pre-view and coherence view being strictly less than p's timestamp in memory during the certification run from $\langle T', M'\rangle$ is equivalent to that pre-view and coherence view being less or equal to the maximal timestamp in memory M. So, roughly, find_and_certify defines the set of allowed promises in $\langle T, M\rangle$ as the set of promises p for which, if p is added as a promise, there exists a certification of the state T' reached after making that promise.

The algorithm for enumerating which promises a thread *tid* in configuration $\langle T, M\rangle$ is allowed to do then works as follows:

1. Enumerate all possible traces of *tid* executing *sequentially* (this thread executing alone under current memory). For programs with infinite loops the user can bound the depth.
2. Discard the traces in which the final state of *tid* has unfulfilled promises.
3. For any remaining trace t: any (new) write done during t is a legal promise step if its store's *pre-view and coherence-view (at its location)* are less than or equal to the maximal timestamp of the current memory M (the memory before the start of the certification).

To illustrate the algorithm, consider the following (partial) program.

$$\begin{array}{l} (a)\ r_1 := \texttt{load}\ [w]; \\ (b)\ \texttt{store}\ [x]\ 1; \\ (c)\ \texttt{store}_{\text{rel}}\ [y]\ 1; \\ (d)\ \texttt{store}\ [z]\ r_1 \end{array} \Bigg\| \ \ldots$$

Assume that the memory is $[1{:}\ \langle w := 1\rangle_1, 2{:}\ \langle z := 1\rangle_0]$, that the promise set of Thread 0 is $\mathsf{prom} = \{2\}$, and that Thread 0 has not yet executed a. The certification algorithm first enumerates all sequential executions of Thread 0 under this memory. Here there are three:

1. a reads 1 from w; b writes $x = 1$ at timestamp 3; c writes $y = 1$ at timestamp 4; and d fulfils the promise 2: $\langle z := 1\rangle_0$.
2. a reads 1 from w; b writes $x = 1$ at timestamp 3; c writes $y = 1$ at timestamp 4; and d writes $z = 1$ at timestamp 5.
3. a reads 0 from the initial state; b writes $x = 1$ at timestamp 3; c writes $y = 1$ at timestamp 4; and d writes $z = 0$ at timestamp 5.

In the second and third case Thread 0 does not fulfil the promise at timestamp 2, so the algorithm discards these executions. Now we inspect the writes done by Thread 0 during the single remaining trace, the first one:

- b writes $x = 1$ at timestamp 3, with pre-view 0 and coherence-view 0, setting $\mathsf{v}_{\mathsf{rOld}}$ to its post-view, 3.

- c writes $y = 1$ at timestamp 4, with pre-view 3 and coherence-view 0: as a store release, c's pre-view includes b's post-view, via $\mathsf{v_{wOld}}$.

Therefore:

1. Promising the write $x = 1$ at timestamp 3 is allowed: it is a write done by Thread 0 during a sequential execution fulfilling all promises, and with a pre-view and coherence view less than or equal to the current maximal timestamp in memory, 2.
2. Promising the write $y = 1$, however, is not allowed, since c's pre-view is $3 \not\leq 2$.

9.3 The model, formally

Fig. 9.2 summarises the types used by the model that were introduced in Section 9.2. For simplicity, values and addresses are mathematical integers. Promising-ARM and Promising-RISC-V use the same definitions, with an architecture flag a switching between ARM and RISC-V behaviour. This only affects the treatment of store exclusive instructions, in the store and load rules. However, not all instructions exist in both architectures: RISC-V has more barriers, and a weak store release.

$$a \in \text{Arch} ::= \mathtt{ARM} \mid \mathtt{RISC\text{-}V} \qquad l \in \text{Loc} \triangleq \text{Val} \quad v \in \text{Val} \triangleq \mathbb{Z} \qquad tid \in \text{TId} \triangleq \mathbb{N} \quad t \in \mathbb{T} \triangleq \mathbb{N} \quad \nu \in \mathbb{V} \triangleq \mathbb{T}$$

$$w \in \text{Msg} \triangleq \langle \mathsf{loc} : \text{Loc}; \mathsf{val} : \text{Val}; \mathsf{tid} : \text{TId} \rangle \qquad \langle x := v \rangle_{tid} \triangleq \langle \mathsf{loc} = x; \mathsf{val} = v; \mathsf{tid} = tid \rangle$$

$$M \in \text{Memory} \triangleq \text{list Msg}$$

$$ts \in \text{TState} \triangleq \left\langle \begin{array}{l} \mathsf{prom} : \text{set } \mathbb{T}; \quad \mathsf{regs} : \text{Reg} \to \text{Val} \times \mathbb{V}; \\ \mathsf{v_{rOld}}, \mathsf{v_{wOld}}, \mathsf{v_{rNew}}, \mathsf{v_{wNew}}, \mathsf{v_{CAP}}, \mathsf{v_{Rel}} : \mathbb{V}; \\ \mathsf{fwdb} : \text{Loc} \to \langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}; \mathsf{xcl} : \mathbb{B} \rangle; \\ \mathsf{xclb} : \text{option } \langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V} \rangle \end{array} \right\rangle$$

$$T \in \text{Thread} \triangleq \text{St} \times \text{TState}$$

$$\vec{T} \in \text{TPool} \triangleq \text{TId} \to \text{Thread}$$

$$\langle \vec{T}, M \rangle \in \text{Machine} \triangleq \text{TPool} \times \text{Memory}$$

Figure 9.2: Types in the semantics

Fig. 9.3 gives the formal definition of the steps of the semantics, cross-referenced with the relevant rules in Section 9.2, starting with some auxiliary definitions.

expression interpretation. The interpretation function for expressions (second and third line) takes an expression and a register state m, and returns the expression's value and view. Constants have view 0; registers are looked up in m; the view for an arithmetic expression merges the views of the arguments (**R8**).

read function. read(M, l, t) gives the result of reading location l at timestamp t in memory M. For $t = 0$, this is the initial value v_{init}, here 0; otherwise either the value of the message in M at timestamp t if its location is l, otherwise *none*.

read-view. read-view(a, rk, f, t) returns either the timestamp t of the read message or the forward view of the message f in the forward bank, subject to certain constraints on the architecture a and read kind rk (**R12**, **R13**, **R14**, **R15**, **R35**).

atomic. atomic(M, l, tid, t_r, t_w) checks whether an exclusive write to l at timestamp t_w by thread tid can become successful, and so atomic with respect to its earlier exclusive read with read message at timestamp t_r in the current memory M (**R30**, **R31**, **R33**).

Now we define thread-local steps, which do not change memory.

$c\ ?\ v_1 : v_2 \triangleq \text{if } c \text{ then } v_1 \text{ else } v_2 \qquad c\ ?\ v \triangleq c\ ?\ v : 0 \qquad v_1 \sqcup v_2 \triangleq \max(v_1, v_2) \qquad v@v \triangleq \langle v, v\rangle : \text{Val} \times \mathbb{V}$

$[\![(-)_1]\!]_{(-)_2} : \mathit{Expr} \to (\text{Reg} \to \text{Val} \times \mathbb{V}) \to \text{Val} \times \mathbb{V}$

$[\![v]\!]_m \triangleq v@0 \qquad [\![r]\!]_m \triangleq m(r) \qquad [\![e_1\ \text{op}\ e_2]\!]_m \triangleq (v_1 [\![\text{op}]\!] v_2)@(v_1 \sqcup v_2) \text{ with } [\![e_1]\!]_m = v_1@v_1, [\![e_2]\!]_m = v_2@v_2$

$\text{read}(M, l, t) : \text{option Val} \triangleq \text{if } t = 0 \text{ then } \mathsf{v}_{\text{init}} \text{ else if } M(t).\text{loc} = l \text{ then } M(t).\text{val} \text{ else } \mathit{none}$

$\text{read-view}(a, rk, f, t) \triangleq \text{if } (f.\text{time} = t \wedge (f.\text{xcl} \Rightarrow (a = \mathtt{ARM} \wedge rk \sqsubseteq \text{pln}))) \text{ then } f.\text{view} \text{ else } t$

$\text{atomic}(M, l, tid, t_\text{r}, t_\text{w}) \triangleq M(t_\text{r}).\text{loc} = l \implies \forall t'.\ (t_\text{r} < t' < t_\text{w} \wedge M(t').\text{loc} = l) \implies M(t').\text{tid} = tid$

$\boxed{T, M \xrightarrow{[t]}_{a,tid} T'}$

(EXCLUSIVE-FAILURE)

$$\frac{xcl = \text{true} \quad ts' = ts\left[\text{regs}(r_\text{succ}) \mapsto \mathsf{v}_\text{fail}@0, \text{xclb} \mapsto \mathit{none}\right]}{\langle r_\text{succ} := \mathtt{store}_{xcl,wk}[e_1]\, e_2, ts\rangle, M \to_{a,tid} \langle \mathtt{skip}, ts'\rangle}$$

(READ)

$$\frac{\begin{array}{l} l@v_\text{addr} = [\![e]\!]_{ts.\text{regs}} \\ \text{read}(M, l, t) = v \\ v_\text{pre} = v_\text{addr} \sqcup ts.\mathsf{v}_\text{rNew} \sqcup (rk \sqsupseteq \text{acq}\ ?\ ts.\mathsf{v}_\text{Rel}) \\ \forall t'.\ t < t' \le (v_\text{pre} \sqcup ts.\text{coh}(l)) \implies M(t').\text{loc} \neq l \\ v_\text{post} = v_\text{pre} \sqcup \text{read-view}(a, rk, ts.\text{fwdb}(l), t) \\ ts' = ts \begin{bmatrix} \text{regs}(r) \mapsto v@v_\text{post}, \\ \text{coh}(l) \mapsto ts.\text{coh}(l) \sqcup v_\text{post}, \\ \mathsf{v}_\text{rOld} \mapsto ts.\mathsf{v}_\text{rOld} \sqcup v_\text{post}, \\ \mathsf{v}_\text{rNew} \mapsto ts.\mathsf{v}_\text{rNew} \sqcup (rk \sqsupseteq \text{wacq}\ ?\ v_\text{post}), \\ \mathsf{v}_\text{wNew} \mapsto ts.\mathsf{v}_\text{wNew} \sqcup (rk \sqsupseteq \text{wacq}\ ?\ v_\text{post}), \\ \mathsf{v}_\text{CAP} \mapsto ts.\mathsf{v}_\text{CAP} \sqcup v_\text{addr}, \\ \text{xclb} \mapsto xcl\ ?\ \langle \text{time} = t; \text{view} = v_\text{post}\rangle : ts.\text{xclb} \end{bmatrix} \end{array}}{\langle r := \mathtt{load}_{xcl,rk}\,[e], ts\rangle, M \to_{a,tid} \langle \mathtt{skip}, ts'\rangle}$$

(FULFIL)

$$\frac{\begin{array}{l} [\![e_1]\!]_{ts.\text{regs}} = l@v_\text{addr} \qquad [\![e_2]\!]_{ts.\text{regs}} = v@v_\text{data} \\ xcl \implies ts.\text{xclb} \neq \mathit{none} \wedge \text{atomic}(M, l, tid, ts.\text{xclb.time}, t) \\ t \in ts.\text{prom} \qquad M(t) = \langle l := v\rangle_{tid} \\ v_\text{pre} = v_\text{addr} \sqcup v_\text{data} \sqcup ts.\mathsf{v}_\text{wNew} \sqcup ts.\mathsf{v}_\text{CAP} \sqcup \\ \qquad (wk \sqsupseteq \text{wrel}\ ?\ (ts.\mathsf{v}_\text{rOld} \sqcup ts.\mathsf{v}_\text{wOld})) \sqcup \\ \qquad ((a = \mathtt{RISC\text{-}V} \wedge xcl)\ ?\ ts.\text{xclb.view}) \\ (v_\text{pre} \sqcup ts.\text{coh}(l)) < t \\ v_\text{post} = t \qquad v_\text{succ} = (a = \mathtt{RISC\text{-}V}\ ?\ v_\text{post} : \bot) \\ ts' = ts \begin{bmatrix} \text{prom} \mapsto ts.\text{prom} \setminus \{t\}, \\ \text{regs}(r_\text{succ}) \mapsto xcl\ ?\ \mathsf{v}_\text{succ}@v_\text{succ} : ts.\text{regs}(r_\text{succ}), \\ \text{coh}(l) \mapsto ts.\text{coh}(l) \sqcup v_\text{post}, \\ \mathsf{v}_\text{wOld} \mapsto ts.\mathsf{v}_\text{wOld} \sqcup v_\text{post}, \\ \mathsf{v}_\text{CAP} \mapsto ts.\mathsf{v}_\text{CAP} \sqcup v_\text{addr}, \\ \mathsf{v}_\text{Rel} \mapsto ts.\mathsf{v}_\text{Rel} \sqcup (wk \sqsupseteq \text{rel}\ ?\ v_\text{post}), \\ \text{fwdb}(l) \mapsto \langle \text{time} = t; \text{view} = v_\text{addr} \sqcup v_\text{data}; \text{xcl} = xcl\rangle \\ \text{xclb} \mapsto xcl\ ?\ \mathit{none} : ts.\text{xclb} \end{bmatrix} \end{array}}{\langle r_\text{succ} := \mathtt{store}_{xcl,wk}\,[e_1]\, e_2, ts\rangle, M \xrightarrow{t}_{a,tid} \langle \mathtt{skip}, ts'\rangle}$$

(FENCE)

$$\frac{\begin{array}{l} v_1 = (\mathsf{R} \sqsubseteq K_1\ ?\ ts.\mathsf{v}_\text{rOld}) \sqcup \\ \qquad (\mathsf{W} \sqsubseteq K_1\ ?\ ts.\mathsf{v}_\text{wOld}) \end{array} \qquad ts' = ts \begin{bmatrix} \mathsf{v}_\text{rNew} \mapsto ts.\mathsf{v}_\text{rNew} \sqcup (\mathsf{R} \sqsubseteq K_2\ ?\ v_1), \\ \mathsf{v}_\text{wNew} \mapsto ts.\mathsf{v}_\text{wNew} \sqcup (\mathsf{W} \sqsubseteq K_2\ ?\ v_1) \end{bmatrix}}{\langle \mathtt{fence}_{K_1,K_2}, ts\rangle, M \to_{a,tid} \langle \mathtt{skip}, ts'\rangle}$$

(REGISTER)

$$\frac{ts' = ts\left[\text{regs}(r) \mapsto [\![e]\!]_{ts.\text{regs}}\right]}{\langle r := e, ts\rangle, M \to_{a,tid} \langle \mathtt{skip}, ts'\rangle}$$

(BRANCH)

$$\frac{[\![e]\!]_{ts.\text{regs}} = v@v \qquad ts' = ts[\mathsf{v}_\text{CAP} \mapsto ts.\mathsf{v}_\text{CAP} \sqcup v]}{\langle \mathtt{if}\ (e)\ s_1\ s_2, ts\rangle, M \to_{a,tid} \langle v \neq 0\ ?\ s_1 : s_2, ts'\rangle}$$

(ISB)

$$\frac{ts' = ts\,[\mathsf{v}_\text{rNew} \mapsto ts.\mathsf{v}_\text{rNew} \sqcup ts.\mathsf{v}_\text{CAP}]}{\langle \mathtt{isb}, ts\rangle, M \to_{a,tid} \langle \mathtt{skip}, ts'\rangle}$$

(SKIP)

$$\frac{}{\langle \mathtt{skip}; s, ts\rangle, M \to_{a,tid} \langle s, ts\rangle}$$

(SEQ)

$$\frac{\langle s_1, ts\rangle, M \to_{a,tid} \langle s_1', ts'\rangle}{\langle s_1; s_2, ts\rangle, M \to_{a,tid} \langle s_1'; s_2, ts'\rangle}$$

(WHILE)

$$\frac{s' = \mathtt{if}\ (e)\ (s; \mathtt{while}\ (e)\ s)\ \mathtt{skip}}{\langle \mathtt{while}\ (e)\ s, ts\rangle, M \to_{a,tid} \langle s', ts\rangle}$$

$\boxed{\langle T, M\rangle \xrightarrow{[t]}_{a,tid} \langle T', M'\rangle}$

(EXECUTE)

$$\frac{T, M \to_{a,tid} T'}{\langle T, M\rangle \to_{a,tid} \langle T', M\rangle}$$

(PROMISE)

$$\frac{w.\text{tid} = tid \quad t = |M| + 1 \qquad ts' = ts[\text{prom} \mapsto ts.\text{prom} \cup \{t\}]}{\langle \langle s, ts\rangle, M\rangle \xrightarrow{t}_{a,tid} \langle \langle s, ts'\rangle, M +\!\!+ [w]\rangle}$$

$\boxed{\langle \vec{T}, M\rangle \to_a \langle \vec{T}', M'\rangle}$

(MACHINE-STEP)

$$\frac{\langle \vec{T}[tid], M\rangle \to_{a,tid} \langle T', M'\rangle \qquad \langle T', M'\rangle \text{ certified}}{\langle \vec{T}, M\rangle \to_a \langle \vec{T}\left[tid \mapsto T'\right], M'\rangle}$$

$\langle T, M\rangle \text{ certified} \triangleq \exists T', M'.\ \langle T, M\rangle \xrightarrow{\text{seq}\ *}_{a,tid} \langle T', M'\rangle \wedge T'.\text{prom} = \{\}$

$\boxed{\langle T, M\rangle \xrightarrow{\text{seq}}_{a,tid} \langle T', M'\rangle}$

(SEQ-EXEC)

$$\frac{T, M \to_{a,tid} T'}{\langle T, M\rangle \xrightarrow{\text{seq}}_{a,tid} \langle T', M\rangle}$$

(SEQ-WRITE)

$$\frac{\langle T, M\rangle \xrightarrow{t}_{a,tid} \langle T', M'\rangle \qquad T', M' \xrightarrow{t}_{a,tid} T''}{\langle T, M\rangle \xrightarrow{\text{seq}}_{a,tid} \langle T'', M'\rangle}$$

Figure 9.3: Thread-local steps, thread steps, and machine steps

Thread-local steps $T, M \xrightarrow{[t]}_{a,tid} T'$

EXCLUSIVE-FAILURE. A store exclusive that has not been executed is always allowed to fail. It sets r_{succ} to v_{fail} (here 1) to signal failure, with 0 timestamp, and sets xclb to *none* (**R32**).

FULFIL. This transition is annotated with the timestamp t of the promise that is being fulfilled. (Other thread-local steps do not have the timestamp annotation.) The definition starts with the pre-condition (from top to bottom). First the rules evaluates the address and data expressions. Rule **R33** explains the condition for exclusive writes. Since writes always promise first and then fulfil, this step requires the write to have been promised.

Rules **R9**, **R5**, **R20**, **R23**, **R36** describe the components included in its pre-view. The pre-view and coherence view have to be less than t (**R18**); the post-view is the timestamp t (**R2**). **R34** explains the view ν_{succ} placed on the register write indicating the success. The post-condition removes the promise (**R18**); writes v_{succ} (here 0) to the "success register" (**R34**); and updates the coherence view (to include t, **R10**), certain views (**R4**, **R21**, **R25**); the forward bank (**R13**, **R35**), and the exclusives bank (**R30**, **R32**).

READ also starts with the pre-condition (from top to bottom). First evaluate the address l; in order to read v it must be $v = \text{read}(M, l, t)$ as described above. The pre-view calculation is described in **R9**, **R5**, **R26**. The pre-view (**R1**) and the coherence view (**R11**) constrain the read. The post-view is defined in **R2**, **R15**. The post-condition updates the register with value and post-view (**R8**); the coherence view with the post-view as in rule **R10**; certain views as in **R4**, **R24**, **R21**; and the exclusives bank as in **R31**.

FENCE. This defines a single rule for all non-`isb` ARMv8 and RISC-V fences in a format matching RISC-V's `fence` instruction. $\texttt{fence}_{K_1,K_2}$ has two arguments: K_1 indicates whether the fence creates ordering with respect to program-order-preceding reads (R), writes (W), or both (RW); similarly K_2 indicates which program-order-later instructions are ordered with it (R, W, or RW). It then updates $\mathsf{v}_{\mathsf{rNew}}$ and/or $\mathsf{v}_{\mathsf{wNew}}$ (depending on K_2), to include $\mathsf{v}_{\mathsf{rOld}}$ and/or $\mathsf{v}_{\mathsf{wOld}}$ (depending on K_1) according to the intuition given in **R4**, **R5**. Define ARMv8's full barrier $\texttt{dmb sy} = \texttt{fence}_{RW,RW}$, its load barrier $\texttt{dmb ld} = \texttt{fence}_{R,RW}$, its store barrier $\texttt{dmb st} = \texttt{fence}_{W,W}$, and moreover RISC-V's "TSO fence" as $\texttt{fence.tso} = \texttt{fence}_{R,R}; \texttt{fence}_{RW,W}$. With these definitions, the behaviour of the ARM barriers is as explained with rules **R4**, **R5** **R6**, **R27**, **R28**.

REGISTER. A register assignment updates the register with the expressions and view from the evaluation of its expression (**R8**).

BRANCH. The pre-condition evaluates the condition expression, branches as determined by this value, and updates $\mathsf{v}_{\mathsf{CAP}}$ (**R21**).

ISB. Executes an `isb` by merging $\mathsf{v}_{\mathsf{CAP}}$ into $\mathsf{v}_{\mathsf{rNew}}$ (**R29**).

SKIP, **SEQ**, and **WHILE**. Mostly as expected. `while` is expressed using a branch.

Thread steps $\langle T, M\rangle \xrightarrow{[t]}_{a,tid} \langle T', M'\rangle$

EXECUTE lifts a thread-local step that does not change memory to a thread step. **PROMISE** allows promising any write message, appending this write to memory and recording its timestamp in

prom. As thread-local steps, thread steps can be annotated with a timestamp t; this is used for the steps for promising and for promise fulfilment. While thread steps allow unconstrained promises, machine steps only allow certified promises. Note that "normal writes" are modelled as promises immediately followed by fulfilment.

Sequential steps $\langle T, M\rangle \xrightarrow{\text{seq}}_{a,tid} \langle T', M'\rangle$
Sequential steps can be one of two kinds: either a thread local step **(SEQ-EXEC)**, or a promise immediately followed by its fulfilment **(SEQ-WRITE)**.

Machine steps $\langle \vec{T}, M\rangle \rightarrow_a \langle \vec{T}', M'\rangle$
Lifts *certified* thread steps (**R37**).

9.4 Proof

The Promising-ARM/RISC-V model using the simple certification of Section 9.2 and Section 9.3 is equivalent to the axiomatic models for ARMv8 and RISC-V. For ARMv8 the proof assumes a simplification, validated by the hand-proof between Flat-axiomatic and ARMv8-axiomatic, that defines the relation obs = rfe | fr | co (rather than obs = rfe | fre | coe); this allows dropping the edge rmw from aob, the edge (ctrl | data); coi from dob and po; [L]; coi from bob that are subsumed by combinations of other edges in ob now. Additionally, this model decomposes all non-`isb` barriers into the pseudo-barriers dmb `rr`, dmb `rw`, dmb `wr`, dmb `ww`, where dmb `rr` orders program-order-earlier reads with program-order-later reads, dmb `rw` program-order-earlier reads with program-order-later writes, etc. In this scheme, dmb `ld` is then the sequence dmb `rr`; dmb `rw`. The ISA manual for RISC-V [121] has two formal axiomatic models produced by the RISC-V Memory Model Task Group, an axiomatic model [121, Appendix B.1] specified in the Alloy framework [67], and one [121, Appendix B.2] specified in herd that follows the same style as that of ARMv8. For RISC-V the proof uses a model based on the latter, but adapted in an equivalent way to it can be unified with the ARMv8 model. The resulting axiomatic model, called Axiomatic, is shown below, with RISC-V specific "switches" (here WL is for the weak release writes occurring only in RISC-V); as before we assume control flow is not delimited and so ctrl; po ⊆ ctrl:

```
let obs = rfe | fr | co
let dob = addr | data
        | (addr | data); rfi
        | (ctrl | addr; po); [W]
        | (ctrl | addr; po); [isb]; po; [R]
let aob = [range(rmw)]; rfi; (if arch = RISC-V then [R] else [A|Q])
let bob = [R]; po; [dmb.rr]; po; [R]
        | [R]; po; [dmb.rw]; po; [W]
        | [W]; po; [dmb.wr]; po; [R]
        | [W]; po; [dmb.ww]; po; [W]
        | [L]; po; [A]
```

```
        | [A|Q]; po
        | po; [L|WL]
        | (if arch = RISC-V then rmw)
let ob = obs | dob | aob | bob
acyclic po—loc | fr | co | rf as internal
acyclic ob as external
empty rmw & (fre; coe) as atomic
```

In Coq, we formally prove equivalence to Axiomatic, deadlock freedom for RISC-V, and correctness of find_and_certify. For the following theorems, the hand-proofs were done in joint work with the aforementioned authors, the Coq development is due to Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur.

Theorem 4. *For a program p, $\vec{R}$ is a final register state of a legal candidate execution of p in Axiomatic if and only if it is that of a valid execution of p in Promising-ARM/RISC-V.*

Theorem 5. *Moreover, Promising-ARM/RISC-V is equivalent to Promising-ARM/RISC-V without certification.*

Theorem 6 (Deadlock freedom)**.** *For any machine state in Promising-RISC-V where every thread state of it is certified, there exists an execution to a machine state with no promises.*

Theorem 7 (Correctness of find_and_certify)**.** *Assume the thread configuration $\langle T, M\rangle$ is certified, and promising p leads to $\langle T', M'\rangle$. Then $\langle T', M'\rangle$ is certified if and only if $p \in$ find_and_certify$\langle T, M\rangle$.*

9.5 Exhaustive exploration

Section 9.2 showed an algorithmic definition for enumerating the legal promise steps, leading to an executable version of Promising-ARM/RISC-V. This section discusses this executable model integrated into the rmem tool with the user-mode ARMv8 and RISC-V Sail ISA models mentioned in Chapter 2.

Executable model The executable model closely follows the Coq definitions where possible, but differs from it in three ways. Firstly, the executable model does not use OCaml code automatically produced by Coq, since interfacing with the existing rmem and Sail infrastructure would be difficult. Secondly, while the Coq model formalises the small imperative language, the executable model integrates definitions for user-mode ARMv8 and RISC-V instructions, meaning it needs logic for computing the views of loads and stores of the ISA definitions. (The Sail model does not yet include ARM's weaker load acquire LDAPR introduced in ARMv8.3. The Coq model does cover its concurrency behaviour.) Third, there are minor differences in the definitions for the sake of executability. We ensured the executable model also experimentally agrees with the axiomatic models on the suites of around 6,500 litmus tests for ARMv8 and 7,000 for RISC-V (150 RISC-V tests we cannot run because they have AMO instructions).

With only a few simple optimisations, this model significantly outperforms the Flat model and the ARM axiomatic model specified in herd. The main performance optimisation is based on two simple observations:

Observation 1: For every valid Promising-ARM trace *tr* in which a non-promise transition t is followed by a promise transition t', the trace that swaps t and t' is a valid equivalent trace. The informal reasoning is that the actions of the transitions commute, and doing t' before t at most "allows more behaviour".

Assume in the trace *tr*, transition t causes the model to step from $\langle \vec{T}, M \rangle$ to $\langle \vec{T}', M' \rangle$, and t' from $\langle \vec{T}', M' \rangle$ to $\langle \vec{T}'', M'' \rangle$. Then $M' = M$, since by assumption t does not promise a new write; and $M'' = M +\!\!+ [w]$ for the write w promised by t'. Now there are two cases to consider: either t and t' are by the same thread or not.

- Assume t and t' are from different threads. Then:
  - The transition actions on the state commute. Since the transitions are from different threads, their thread state updates are "disjoint". Since t does not change the memory state, the actions on the memory also commute.
  - t' can be done before t. Since t and t' are by different threads, t does not affect the thread state of t', and it is $M' = M$. Since the set of allowed promises of a thread are a function of its thread state and the memory, the transition t' is enabled before t.
  - t is possible after t'. The transition t' does not change the thread state of t. By assumption t is not a promise, so t could be a memory read transition or a purely thread-local transition that does not interact with memory. If t is a read, then in the incremented memory its thread can still read from the same write as before t'. If it is a purely thread-local transition, then t is unaffected by the incremented memory. Informally speaking, the thread state of t can still be certified after t' since by assumption *tr* is a valid trace, leading to a state in which all threads have fulfilled their promises. (In ARMv8, in principle, t' could break a promise by the thread of t—due to the load/store exclusive issues discussed earlier. This cannot be the case here, since by assumption *tr* is a trace to a final state with all promises fulfilled.)
- Assume t and t' are from the same thread. Then:
  - The transition actions on the state commute. Since t is a non-promise transition it does not change the promise set of its thread or the memory state. Transition t' only changes the thread's promise set and the memory set, so the pieces of state updated by the two transitions are disjoint.
  - t' can be done before t. In order for this to hold it is only required that the promise transition t' can be certified before transition t: there has to be a sequential execution from the thread configuration reached when taking t' before t, to one with empty promise set. By assumption t' is certified after transition t. But then the sequential execution that witnesses the certfiability of the thread configuration reached by taking transition t' after t, prefixed with t, is a witness for the certifiability of the thread configuration reached after taking transition t' before t.

– t can be done after t'. The only change t' makes to the thread state is adding a promise; hence, if we can show that t can be certified in the thread configuration after t', then t' is a possible transition after t. But the fact that t can be certified after t' follows from the assumption that in *tr* the transition t' is possible after t. Since t' is allowed after t, we know that the thread configuration reached after taking t followed by t' is certified. Since the actions of t and t' commute, the thread configuration reached when doing t' followed by t is the same, and so also certified.

We proved the following resulting key property of the model in Coq, for the small imperative language of Section 9.1 handled by the Coq formalisation.

Theorem 8. *For every Promising trace tr, there exists a trace tr′ with same final state such that tr′ can be split into a sequence of promise transitions followed by only non-promise transitions.*

Observation 2: For every Promising-ARM trace any two consecutive non-promise transitions t and t' from different threads commute. t and t' do not change memory, only their respective thread states. Since both transitions' pre and post-conditions depend only on memory and their thread states a trace that re-orders t and t' is valid, too.

The model uses these ideas to enumerate the set of possible final states in the following way:

1. The model starts in "promise-mode": in this mode it allows only promise transitions, no other thread execution steps. In addition, the model allows non-deterministically leaving promise-mode with a stop-promising transition.
2. Once there are no more promising transitions or the stop-promising transition is taken the model enters "non-promise-mode". In this mode the model allows no more promises, so the memory state is now fixed to the one obtained from promise-mode. In non-promise mode, the model computes the set of allowed thread states for each thread independently: since memory is fixed, the threads do not interact with each other any longer. Hence, the model can compute the set of final model states by first computing, for each thread separately, the set of final thread states of this thread under the given memory, and then taking the cartesian product of these sets of thread states: every combination of final thread states for the individual threads together with the memory state returned from promise-mode is a valid final state.

This reduces the model non-determinism to first enumerating possible "final memories" and then exploring the final thread states that are possible as a result of this memory.

In the description so far, the exploration, however, has an additional source of unnecessary non-determinism: non-deterministically allowing to quit the promise-mode means the model can stop promising "too early"; the execution of a thread may still involve a store whose write has not been promised yet. To avoid unecessary non-determinism from leaving promise-mode too early, the model should only be allowed to leave promise-mode when it is possible for each thread to execute from its current state to some final thread state with empty promise set in a way that does not require doing any new writes.

The model achieves this by computing extra information when certifying promises: for each "certification run", in addition to returning whether a given thread state is certifiable, it returns

information about whether it is possible to certify the thread state with an execution of this thread that does not involve making new writes. Supplied with such information returned from the certification of each thread, the model then allows taking the stop-promising transition only if, for each thread, there exists a certifying execution that does not propagate new writes.

The tool uses a second optimisation to improve performance of non-promise-mode: after entering non-promise-mode, the tool runs the remaining thread actions without "normal" certification. Instead it uses a cheaper approximation to eagerly detect inconsistency (checking only that the next instruction step does not increase coh, v_{wNew}, or v_{CAP} to a value larger than or equal to an existing promise's timestamp) and discards deadlocking traces in which promises are not fulfilled. While in promise-mode, dropping certification would render the exploration computationally infeasible, in non-promise-mode, since threads are no longer interleaved, this improves the search complexity, since the certification of any thread's initial state under a final memory state already requires the effort of exploring the thread's possible final states.

9.6 Evaluation

To evaluate the exploration tool, we test several standard datastructure and lock implementations, for ARMv8. The code of the examples can be found in the supplementary material of the Promising-ARM/RISC-V paper, currently under submission [103]. We implement the examples in C++, Rust, or assembly. To the code for the actual data structures we add test code: some threads running in parallel, each executing the datastructure's operations while logging some extra information that we use to check the correctness of the results. In the case of C++ or Rust, we compile the code using a standard GCC or RUSTC cross compiler, and then run a script that maps the assembly into the *litmus* format used by herd and rmem: our tool does not support dynamic thread creation yet; the litmus format allows writing the code directly as a parallel thread composition. Another limitation is that our tool does not yet support dynamic memory allocation, which we "fake" here with a very naive `malloc` as part of the source code. Adding dynamic thread creation and memory allocation will just require additional engineering.

Running our model exhaustively, integrated into rmem, outputs the list of possible final outcomes and allows us to check whether the code has behaved correctly.

Tested examples and results We give an overview of the tests we ran in Table 9.1. We test, from simple to complex: three different spinlock variants, implemented in assembly (SLA), C++ (SLC), and Rust (SLR), where Linux-Spinlock (SLA) is an example taken from a Linux kernel spinlock implementation [79, 64], which was also used in Pulte et al. [104] for demonstrating Flat[4] (the other test from that paper, `spin_unlock_wait`, require mixed-size support); single-producer-single-consumer (PCS) and single-producer-multiple-consumers (PCM) circular queues; a ticket lock (TL); Treiber stack [40], separately implemented in C++ (STC) and Rust (STR); Chase-Lev dequeue (DQ) [77, 70]; and the aforementioned variants of the Michael & Scott queue (QU). In addition, for the last four examples we also try versions optimised for ARMv8. Table 9.2

[4]We change mixed-size loads in the example to same-size and confirmed that performance is unaffected.

Test	Lang	LOC	Ts	Test	Lang	LOC	Ts
SLA	asm.	44	2	TL	C++	120	3
SLC	C++	51	3	STC	C++	366	3
SLR	Rust	84	3	STR	Rust	393	3
PCS	C++	69	2	DQ	C++	247	3
PCM	C++	130	3	QU	C++	473	3

Table 9.1: *LOC* = assembly lines, *Ts* = number of threads

shows the results. All programs in C++ and Rust are compiled with GCC 6.3.0 and RUSTC 1.30 with optimization level 3. All numbers are from a standard desktop machine, Ubuntu 16.04 Intel Core i7-7700 at 3.60GHz, 8GB memory.

The names mean the following. For the spinlock tests: spinlock-*n* means *n* loop unrollings on all threads; PCM-*n*-*n*-*n*: Thread 1 producing *n* times, Threads 2 and 3 consuming *n* times; PCS-*n*-*n* the same for Thread 1 producing, Thread 2 consuming; TL-*n*: threads spin *n* times to acquire lock; STC/R-*abc*-*def*-*ghi*: Thread 1 pushing *a* times, popping *b* times, and again pushing *c* times, and analogously for Thread 2 with *def* and Thread 3 with *ghi*; DQ/(opt)-*abc*-*d*-*e*: Thread 1 pushes *a* times, pops *b* times, and pushes *c* times, Thread 2 steals *d* times, and Thread 3 steals *e* times; QU/(opt)-*abc*-*def*-*ghi*: Thread 1 enqueues *a* times, dequeues *b* times, and enqueues *c* times again, analogously with *def* for Thread 2 and *ghi* for Thread 3.

We tried running the examples on herd, but all but the spin and ticket locks (SLC,SLR,TL) require instructions unsupported by herd. For SLC and TL, we ran the tests on herd:

- SLC-1: 14.72 sec, (Promising: 3.21 sec)
- SLC-2: stack overflow in 123.51 sec, (Promising: 4.69 sec)
- TL-1: 31.04 sec, (Promising: 10.16 sec)
- TL-2: 2370.23 sec, (Promising: 13.72 sec)

The results show that the Promising model scales much better than Flat for the tested examples. However, it may be possible to further improve performance significantly over the current results by studying existing model checking techniques. For instance, for certain litmus tests Promising does not perform as well as Flat: tests with a large number of writes whose interleaving does not matter. In such tests, Promising explores all interleavings of the writes, and each interleaving leads to a different memory state, even if the order of some writes does not affect the possible final outcomes. Here we believe partial-order reduction techniques [49] can help improve performance.

Where data is available, the results also show the Promising model performs much better than herd. The comparison here is less clear: on the one hand herd does not accurately capture the full ISA behaviour, meaning it has to do less work during the exhaustive search; on the other hand herd is not designed primarily to be a fast model checker, and it is not clear how the axiomatic approach compares to ours in search complexity. Herd computes the set of allowed behaviours in two steps: first, herd enumerates the set of all possible *candidate executions*, concrete executions given by a program-order unfolding, write values for the reads, and relations capturing the concurrency

behaviour satisfying only minimal well-formedness conditions; then, herd discards executions violating the model's axioms. A naive generation of the set of candidate executions is expensive. However, it may be possible to efficiently generate candidate executions using an approach similar to the promise-first enumeration described here, or adapt the model checker of Kokologiannakis et al. [73] based on axiomatic models to ARMv8/RISC-V.

Test	Promising	Flat
SLA-1	0.27	0.41
SLA-2	0.30	3.38
SLA-3	0.33	21.57
SLA-4	0.39	110.18
SLA-5	0.44	526.76
SLA-6	0.52	2277.72
SLA-7	0.61	9108.53
SLA-8	0.73	ooT
SLA-9	0.86	ooT
SLA-10	1.01	ooT
SLC-1	3.21	8.63
SLC-2	4.69	121.98
SLC-3	6.58	1472.74
SLR-1	2.47	3.70
SLR-2	3.50	17.51
SLR-3	4.88	52.52
PCS-1-1	0.26	0.33
PCS-2-2	0.40	10.33
PCS-3-3	1.36	249.26
PCM-1-1-1	0.30	23.58
PCM-2-2-2	1.70	ooT
PCM-3-3-3	71.12	ooT
TL-1	10.16	456.12
TL-2	13.72	2202.12
TL-3	18.08	ooT
TL(opt)-1	10.28	1180.33
TL(opt)-2	14.54	7115.31
TL(opt)-3	20.13	ooT

Test	Promising	Flat
STC/(opt)-100-010-000	0.36 / 0.36	35.26 / 104.57
STC/(opt)-100-010-010	0.42 / 0.42	2144.52 / 5943.50
STC/(opt)-100-100-010	8.70 / 8.70	ooT / ooT
STC/(opt)-110-011-000	7.64 / 8.13	ooT / ooT
STC/(opt)-110-100-010	21.84 / 22.48	ooT / ooT
STC/(opt)-200-020-000	7.16 / 7.12	ooT / ooT
STC/(opt)-210-011-000	615.41 / 637.98	ooT / ooT
STR-100-010-000	0.35	4.61
STR-100-010-010	0.39	77.21
STR-100-100-010	7.30	8940.03
STR-110-011-000	6.55	ooT
STR-110-100-010	18.09	ooT
STR-200-020-000	5.80	11325.87
STR-210-011-000	522.19	ooT
DQ/(opt)-100-1-0	0.30 / 0.30	2.93 / 2.97
DQ/(opt)-110-1-0	0.44 / 0.44	1042.88 / 1114.39
DQ/(opt)-110-1-1	0.66 / 0.65	ooT / ooT
DQ/(opt)-111-1-1	1.76 / 2.44	ooT / ooT
DQ/(opt)-211-1-1	9.51 / 37.10	ooT / ooT
DQ/(opt)-211-2-1	28.55 / 111.54	ooT / ooT
QU/(opt)-100-000-000	1.34 / 2.95	2983.11 / ooT
QU/(opt)-100-010-000	2.55 / 5.66	ooT / ooT
QU/(opt)-100-010-010	4.53 / 10.00	ooT / ooT
QU/(opt)-100-100-010	712.57 / 4984.94	ooT / ooT
QU/(opt)-110-011-000	589.50 / ooT	ooT / ooT
QU/(opt)-110-100-010	2108.12 / ooT	ooT / ooT
QU/(opt)-200-010-010	531.41 / ooT	ooT / ooT
QU/(opt)-200-020-000	286.99 / 10585.10	ooT / ooT

Table 9.2: Run times in seconds. ooT = more than four hours.

Chapter 10

Related work

This chapter gives an overview over the literature most relevant to this text. The most closely related work studies the concurrency models of the x86, Sparc, Power, ARM, Itanium, and Alpha architectures.[1]

10.1 Hardware concurrency models

Sparc The Sparc architecture defines three multicopy atomic concurrency models, from weak to strong: Relaxed Memory Order (RMO), Partial Store Order (PSO), and Total Store Order (TSO) [1] and includes formal (albeit prose) specifications in axiomatic style for them. RMO relaxes all basic orderings of memory accesses to different addresses — read-read, read-write, write-read, and write-write ordering — while preserving dependencies and preserving coherence, except read-read coherence; PSO strengthens RMO by enforcing read-read and read-write ordering; TSO only allows write-read re-ordering. Notably, the axiomatic model for MCA ARMv8 is broadly similar to that of SPARC RMO, but differs in maintaining read-read coherence while allowing same-address write-read re-ordering (write-forwarding).

The Sparc memory models have been studied in a number of publications providing formalisations or tools. Sindhu et al. [115] give an axiomatic formalisation for Sparc TSO and PSO. Higham et al. [62] give operational and axiomatic formalisations for PSO and TSO. Park and Dill [99] define an operational executable specification of Sparc RMO for use in model checking small concurrent programs. Gharachorloo [56] presents formalisations of RMO, PSO, and TSO in an axiomatic framework. Sun's TSOtool [61] is an efficient executable tool for detecting certain memory model violations in Sparc TSO hardware implementations. Loewenstein et al. [81] describe a tool for detecting violations of the architectural intention, as opposed to just memory model violations, for TSO. Alglave [8] formalises Sparc RMO in the framework of Alglave et al. [14].

Whereas in the case of MCA ARMv8 and Power widely used implementations exhibit many of the relaxed behaviours allowed by the architecture, existing Sparc implementations have chosen to implement TSO [110].

x86 The Intel x86 architecture [3] has a memory model similar to Sparc TSO: x86 TSO. Since TSO is multicopy atomic and relaxes only write-read re-ordering the model is much stronger than that of (MCA and non-MCA) ARMv8 and Power. Therefore many of the questions that the ARMv8 and Power concurrency models are concerned with, e.g. about details of the storage subsystem

[1] Some of the description of related work is adapted from Pulte et al. [104]; part of the summary of earlier work on hardware concurrency follows Sarkar et al. [110].

behaviour and intra-thread dependencies, do not arise for x86 TSO.

The original architecture documentation did not specify the concurrency semantics precisely. The architecture manual has different memory models for different processor families and the current documentation describes architecturally allowed write-read re-ordering. Roy et al. [108] give an algorithm for detecting memory model violations for Intel IA-32 and for Intel Itanium. Burckhardt et al. [38] define two concurrency models for x86 for the purpose of proving the correctness of compiler passes: one based on rewrite rules and the other a denotational model that interprets concurrent programs as event sequences, both more relaxed than TSO. In contrast, Sarkar et al. [109], Owens et al. [98], Sewell et al. [114] study the memory model of x86 processors based on experimental hardware data and vendor-supplied example behaviours and find x86 to have TSO concurrency semantics. The authors define operational and axiomatic concurrency models for x86 TSO with integrated instruction semantics for a small number of instructions. The model is formalised in HOL4 as well as the executable memevents tool that exhaustively enumerates the possible outcomes of a concurrent x86 program. Moreover, the authors prove a result that data-race-free programs have Sequentially Consistent behaviour. Sevcík et al. [113] describe the first verified compiler for a relaxed memory model. The authors extend CompCert and its correctness proof [78] to develop a verified compiler from CompCert's Clight intermediate language extended with threads and shared memory to x86 TSO.

Power Power [2] has a relaxed-memory concurrency model in which all basic memory access re-orderings are allowed and where writes are non-multicopy-atomic. The imprecise description of the concurrency behaviour in Power's architecture documentation has lead to many publications studying the Power concurrency model.

Early studies of Power [45, 56, 6, 4] either do not give a detailed account of the semantics or are based on now-outdated versions of the Power architecture and hardware. Corella et al. [45] give a formal specification in axiomatic style. The model does not seem to handle ordering resulting from dependencies, and according to Sarkar et al. [110] gives too weak semantics to barriers that allows architecturally forbidden behaviour. Adve and Gharachorloo [6] give a comparison of a number of memory models, including SC, TSO, PSO, RMO, Power, and Alpha. The Power specifications given by Gharachorloo [56] in an axiomatic framework does not capture the precise semantics ("approximate", Section 2.4), and only handles the full memory barrier. Adve and Gharachorloo [6] give a high-level overview, describing the basic order relaxations in Power but without detailing the thread ordering details, and not covering all memory barriers. Adir et al. [4] give a formalisation of Power in axiomatic style in terms of a concept of "view orders", based on testing and discussion with architects. The Power concurrency architecture has changed since this publication and now has different barrier semantics [110, Section 9].

Alglave et al. [13] give a work-in-progress axiomatic concurrency model for Power formalised in HOL4. The model's instruction semantics for a small ISA fragment generates candidate executions in the form of event structures; the model is validated using some hardware testing. Alglave et al. [14] present a framework for axiomatic models formalised in Coq that is instantiated with models for SC, TSO, and Power, all specified in terms of a notion of a "global happens before" (ghb) order,

and proves results about regaining stronger concurrency semantics when placing fences between memory accesses. The authors discuss as shortcomings of the former paper that on some tests the model differs from architectural intention in the semantics of barriers [14, Conclusion]. Alglave [9] expand on this and establishes a hierarchy of such ghb models; Alglave and Maranget [11] develop a tool that inserts fences into concurrency TSO or Power programs to regain stronger memory models.

Sarkar et al. [110] define an executable operational concurrency model, ppcmem, for Power based on discussion with IBM architects and hardware testing that aims to explain the concurrency behaviour using mechanisms abstracting from those of real CPU implementations. ppcmem allows interactive exploration of concurrent programs and exhaustively enumerating the allowed behaviours of small programs. The model was later found to forbid some hardware-observed behaviour [17] and had to be adapted. Sarkar et al. [111] and [31] extend this model with Load Reserve/Store Conditional instructions and prove the correctness of a compilation scheme from C/C++ to Power. This compilation scheme was later found to be broken, independently by Trippel et al. [120] and Lahav et al. [75]. Gray et al. [60] extend ppcmem with an instruction semantics model for large parts of the Power user-mode ISA.

Boudol et al. [35] propose a framework for operationally modelling relaxed-memory concurrency models in terms of a buffer recording memory operations in which they are delayed, potentially re-ordered and then "globally performed". The authors present a Power-like model as an example; Alglave et al. [17] note the barrier semantics presented there is stronger than in Sarkar et al. [110]. Maranget et al. [93] give an overview over the concurrency behaviours in the memory models of TSO, Power, and earlier versions of ARM, focusing on example behaviours rather than models. Mador-Haim et al. [87] build on the work of Sarkar et al. [110] and define an axiomatic concurrency model and executable tool that uses SAT solving for enumerating the possible outcomes of concurrent programs. In contrast to most axiomatic models and the previously mentioned ones this axiomatic model follows the operational model [110, 111] in having *multiple events* per memory access: "satisfy read events, initiate write events, and commit events for both reads and writes" [87]. The authors hand-prove the model equivalent with the operational model of [110, 111], but the two models were later found to experimentally mismatch [17]. Our work in Flur et al. [52] generalises the model of Sarkar et al. [110, 111] to cover memory accesses of mixed sizes.

For the purpose of allowing for the sound use of SC verification tools for relaxed-memory models Alglave et al. [16] define an operational concurrency model that can be instantiated with different concurrency architecture definitions, including x86 TSO, PSO, RMO, and an axiomatic Power model; the operational model framework is proved equivalent to the parametric axiomatic model of Alglave et al. [14]. Whereas the previously mentioned operational Power [110, 111] and later ARMv8 models attempt to abstract from real hardware and compute purely incrementally, this model is different in that it operationalises the axiomatic framework and has explicit deadlock states. Alglave et al. [17] introduce the herd tool, a tool for defining axiomatic memory models and enumerating the possible behaviours of small litmus programs, and give a definition for Power (and ARM, as discussed later) based mostly on extensive testing. This axiomatic model distinguishes

between read-satisfaction and read-commitment, and write-initiation and write-commitment in a similar way as the operational model does, to accurately capture the Power behaviour. In contrast to Mador-Haim et al. [87], however, the axiomatic model does this with only a single event per memory access: interpreting the same event differently in different contexts, e.g. in certain relations interpreting a read event as standing for the read-satisfaction, and in others for the read-commitment. Since the model has to handle the non-multicopy-atomic semantics, the axiomatic model is considerably more complex than the later official axiomatic concurrency model for multicopy atomic ARMv8: it requires three axioms in addition to the standard per-thread coherence axiom; in the non-MCA setting, unlike ARMv8-axiomatic, their happens-before relation cannot include the edges fre (from-reads external) and coe (coherence external) whose composition with thread-local order has to take partial propagation of writes into account. The paper constructs an operationalised model from the axiomatic Power model and uses this as an intermediate model to prove that the axiomatic model allows all executions allowed by the operational model of Sarkar et al. [110]. This thesis' proof of equivalence of the official ARMv8 axiomatic model and the Flowing-based operational model uses an intermediate axiomatic model that in a similar way to Mador-Haim et al. [87] and Alglave et al. [17] distinguishes between multiple events per instruction.

The (aforementioned) operational Power model of Sarkar et al. [110, 111] has been validated by extensive testing and detailed discussion with architects, and is the most feature-complete model — integrating an instruction semantics for large parts of the ISA [60] and covering mixed-size memory accesses [52]. The work on Sequential Consistency for Power programs with mixed-size accesses of Chapter 5 is based on this model. Moreover, the (non-MCA and MCA) ARMv8 concurrency models and the rmem tool that the work described in this thesis is concerned with builds on Sarkar et al.'s ppcmem tool and Power model. Chapter 3 will give a more detailed description of this Power model.

ARM ARMv7 [23] and non-MCA ARMv8 [20] have non-multicopy-atomic relaxed memory models broadly similar to Power. Similarly to the case of Power, in ARM's original architecture documentation the concurrency specification was imprecise, prompting research to establish the precise concurrency semantics.

Chong and Ishtiaq [42] define an axiomatic concurrency model for the non-multicopy-atomic ARMv7 architecture in Coq, dealing with only plains reads and writes and `dmb` barriers. Some of the previously mentioned Power models were initially thought to also be suitable or adaptable to be models for non-MCA ARM (ARMv7 in the earlier papers) [13, 110, 111, 31]. Alglave et al. [17] find their Power model forbids certain behaviour ARMv7 intends to allow and propose an axiomatic model for ARMv7 that is based on their Power model and a reading of the ARMv7 architecture documentation. The model follows the Power model but relaxes its preserved-program ordering originating from same-address memory accesses. Based on the comparison of hardware and model behaviour the authors find hardware errors in the form of read-read coherence violations. The model is primarily validated by experimental hardware testing. The aforementioned relaxed-memory overview of Maranget et al. [93] treat both Power and ARM. The authors find differences

in the observable hardware behaviour between Power and ARMv7 hardware: certain tests show differences in the thread local behaviour of the tested hardware; for some tests Power hardware has observable non-multicopy-atomic behaviour and ARM does not; the paper discusses a test that shows a behaviour that is observable on ARM hardware and that was confirmed to be architecturally intended in ARM is forbidden by the Power model of Sarkar et al. [110].

Due to differences in architectural intention and allowed behaviour between Power and ARM, Flur et al. [51] develop a new specification for the concurrency semantics of non-MCA ARMv8. The Flowing and POP models presented there are based on the Power model of Sarkar et al. [110, 111] but have a different storage subsystem design and differ in many details of the Power model's thread subsystem. Similar to the Power model, Flowing and POP can be used as an executable tool for the interactive and exhaustive exploration of possible outcomes of concurrent programs. The model is validated by detailed discussion with ARM architects and experimental testing. Flur et al. [52] extend the Flowing and POP models to cover the behaviour of ARMv8 programs with mixed-size memory accesses and reports hardware errata in two processor implementations uncovered using the model. Podkopaev et al. [101] prove the correctness of the compilation scheme of the Promising C11 concurrency model to the non-MCA ARMv8 architecture described by POP.

The work on ARM (and RISC-V) concurrency presented here builds on the work on the aforementioned Flowing and POP models of Flur et al. [51, 52]. In particular, the results on mixed-size Sequential Consistency for ARM, described in the latter paper [52], are abstraction results for the POP storage subsystem; the Flat operational concurrency model for the later MCA ARMv8 architecture (of Chapter 6) is an adaptation of Flowing and POP. The details of the Flowing and POP models will be explained in detail in Chapter 3.

Flowing and POP describe a non-multicopy-atomic concurrency semantics, as was intended by ARM for these earlier versions of the ARMv8 architecture. While the existing ARMv8 hardware exhibits many of the relaxed behaviours allowed by these models, available implementations were only observed to have multicopy atomic behaviour, as indicated by the hardware test results for some of the existing processors [93, 17, 51, 52]. ARM have recently revised the ARMv8 architecture [22] and shifted to a multicopy atomic concurrency semantics, partly due to the work on operational concurrency models for the non-MCA ARMv8 architectures of Flur et al. [51] and Flur et al. [52] (as discussed in [104]). The MCA architecture also has, for the first time, a formal reference concurrency model, in axiomatic style, due to Will Deacon of ARM [22, 47]. The work presented here includes the Flat concurrency model, based on Flowing and POP, and the Promising ARM model, for the MCA ARMv8 architecture, and equivalence proofs with respect to ARM's reference axiomatic model.

Intel Itanium Intel Itanium [66] has a relaxed memory model which, in contrast to Power and the earlier non-MCA ARMv8, is specified formally in the vendor documentation in terms of a set of ordering rules Yang et al. [125]. (Intel's formal specification of Itanium [65] seems to longer be available online.) A number of different Itanium formalisations exist. Chatterjee and Gopalakrishnan [41] provide an operational concurrency specification in terms of per-thread

read and write buffers. Joshi et al. [69] give a formalisation of Itanium for the purpose of verifying cache coherence protocols using model checking, in TLA+. Yang et al. [125] formalise Itanium axiomatically in higher order logic and develop a translation of the specification into SAT problems to obtain an executable model; Yang et al. [126] describe the framework used for making axiomatic models executable and additionally apply it to a number of classical concurrency models. Moreover, as mentioned previously, Roy et al. [108] present an algorithm for detecting memory model violations for Intel Itanium. Higham et al. [63] provide two alternative formalisations for Itanium and prove an inclusion result between these two and the Itanium concurrency model as presented in the architecture documentation.

Alpha Alpha [44] has a relaxed memory model that relaxes all basic orderings of memory accesses to different addresses [56, p. 35]. A major difference between Alpha and most other hardware concurrency models, including Power and ARMv8, is that Alpha's memory model does not respect certain data dependencies between loads, allowing for executions in which an address-dependent load returns a value before the load feeding into its address [94]. Attiya and Friedman [28] formalise the concurrency model of Alpha axiomatically and prove results about obtaining simpler programming models when following certain programming disciplines. Gharachorloo [56] gives an axiomatic formalisation of Alpha. Joshi et al. [69] give an operational formalisation of Alpha in TLA+ for the use in verifying a cache coherence protocol for Alpha. The thesis of Alglave [8] also formalises Alpha in the framework of Alglave et al. [14].

Other In early work on formally studying hardware concurrency models, Collier [43] defines a graph-based framework for specifying concurrency models and gives a number of memory model definitions in the framework. Collier's models stand out by being based on experimental hardware testing. Collier also defines the notion of multicopy atomicity. (This definition is stronger than the one assumed here, as will be discussed later.) Adve [5] proposes a framework for defining concurrency models in a programmer-centric way by formally specifying the conditions under which Sequential Consistency can be regained, and gives a number of concurrency model definitions in this framework, including data-race-free (DRF) models: in DRF models programmers provide annotations to distinguish between memory accesses used for synchronisation and those to access other data; the model provides Sequentially Consistent semantics for programs that have no data races on non-synchronisation memory accesses. Gharachorloo [56] also explores programmer-centric concurrency models. There, the programming language provides Sequentially Consistent semantics and the programmer annotates the program's memory accesses with labels in order to provide the underlying implementation with information about whether the memory access may participate in a data race, whether it is used for synchronisation purposes, etc. The author then studies how these models can be efficiently implemented in concurrency models that axiomatically specify the conditions sufficient for such annotated programs to appear Sequentially Consistent. The author also surveys a number of hardware concurrency models, including Sparc TSO, PSO, and RMO, Alpha, Power, and IBM-370, and provides axiomatic specifications for some of them.

Steinke and Nutt [116] propose a formalism for specifying memory models based on notions of views, formalise a number of classical memory models, and prove relations between the models. Boudol and Petri [34] give an operational relaxed memory concurrency model, for no specific architecture, defined in terms of hierarchical FIFO buffers where buffers are per-address in order to allow re-ordering of different-address memory accesses, and prove that data-race-free programs have Sequentially Consistent behaviour. Arvind and Maessen [25] develop a procedure for enumerating possible outcomes of multicopy atomic relaxed-memory programs that is parametric in some ordering choices. Alglave et al. [18] study concurrency behaviour in GPUs and define a model for NVidia GPUs similar to Sparc RMO. Zhang et al. [127] propose an operational concurrency model based on value speculation that aims for simpler definitions than those of Power and ARM. Zhang et al. [128] propose a concurrency model for the RISC-V architecture that forbids load-store re-ordering in order to avoid the complexity of operational models allowing this, in multicopy atomic and non-multicopy-atomic variants, give an axiomatic specification, and an operational model in which instructions execute in order and atomically, and prove the two equivalent. The authors conduct a performance study to argue the proposed memory model does not pose issues for CPU performance compared with more relaxed models.

Classical models and definitions Lamport [76] defines Sequential Consistency. Dubois et al. [48] introduce the Weak Consistency model that distinguishes between two classes of memory accesses: synchronisation accesses and non-synchronisation accesses to shared memory; the model specifies Sequentially Consistent behaviour for synchronisation accesses with respect to each other and only weak requirements from non-synchronisation accesses. Lipton and Sandberg [80] define the PRAM model that requires that writes originating from the same thread have to be observed by any processor in their issuing order. Goodman [58] defines the similar Processor Consistency model. Gharachorloo et al. [57] define the Release Consistency model with Acquire and Release "half-barriers": Acquire loads are ordered before program-order-later memory accesses, Release stores are ordered after all program-order-earlier memory accesses. The Release Consistency model has been incorporated into some later concurrency models, including ARMv8 with Acquire and Release instructions. Ahamad et al. [7] propose the Causal Consistency model that preserves ordering between memory accesses that are related according to a notion of causality.

10.2 Axiomatic/operational model equivalence

In addition, several papers proved the equivalence of operational and axiomatic relaxed memory models in one form or another. Focusing on hardware models: Ahamad et al. [7] axiomatically specify Causal memory and prove that an operational implementation thereof satisfies the axioms (thereby proving a refinement rather than an equivalence). Higham et al. [62] formalise SPARC PSO and TSO and a number of simpler memory models in both axiomatic and operational style and prove equivalence. Owens et al. [98] define and prove equivalent an operational and an axiomatic concurrency model for x86-TSO; similarly Burckhardt and Musuvathi [37, App. A] for TSO. The PSO and (Sparc or x86) TSO models are much stronger than MCA ARMv8. In particular,

in the absence of load-store re-ordering many of the difficulties related to dependencies and preserving per-thread coherence do not arise in these stronger models.

Alglave et al. [16], as mentioned above, define a framework for operational models matching the axiomatic models of Alglave et al. [14] and prove the two equivalent. On the one hand their axiomatic model is more complicated than the reference MCA ARMv8 axiomatic model since they need to handle the non-multicopy-atomic write propagation. On the other hand, in contrast to the Power and ARM operational models discussed in this text [110, 111, 51, 52, 104], and the MCA ARMv8 Flat model and the Promising-ARM/RISC-V model covered by this thesis' equivalence proofs, their operational model is "less operational": it does not work purely incrementally but may deadlock when reaching a state violating certain model guarantees; this greatly simplifies the operational model's coherence mechanism since the operational model neither needs the restarts/rollbacks, nor do model or proof have to deal with the write commitment, store finish, and load finish conditions to prevent coherence violations, which are the main difficulty in the proof of equivalence between the Flat model and the reference ARMv8 axiomatic model (in showing the operational model allows the behaviours allowed by the axiomatic model). Finally, Alglave et al. [17] define axiomatic concurrency models for ARM and Power in herd, and prove that the operational Power model of Sarkar et al. [110] satisfies the conditions of that axiomatic model, including an equivalence proof of the axiomatic Power model and an operationalised version thereof. Again, their proof has to handle the more complicated non-multicopy-atomic semantics, but in contrast to this thesis' proofs their proof is only a refinement proof, not an equivalence proof.

10.3 Test generation and verification tools

Much of the previously mentioned work on Power and ARM concurrency models relies on extensively running litmus concurrency tests on hardware and investigating the hardware-allowed behaviour. Building on an earlier tool by Sarkar et al. [109], Alglave et al. [15] develop the litmus tool that, given a litmus concurrency test, creates executable tests for a number of supported platforms; the tools uses certain heuristics to increase the likelihood of observing non-Sequentially-Consistent behaviours. The tool used to produce many of these tests is the diy tool of Alglave and Maranget [10], Alglave [8]. diy automatically produces litmus tests that check for non-Sequentially-Consistent executions whose execution graph has cycles of certain user-supplied shapes. Mador-Haim et al. [86] generate litmus tests distinguishing between memory models.

Lustig et al. [82] develop a tool for automatically checking processor implementations against concurrency models specified axiomatically. To this end, the tool uses axiomatic specifications of the *micro-architecture* in terms of more fine-grained events than typical architecture-level axiomatic models, and checks against architecture-level models using satisfiability tests, or by comparing the allowed outcomes of both on litmus tests. While Lustig et al. [82] focus on checking the instruction pipeline, Manerkar et al. [88] build on simliar ideas to develop a tool that also allows specifying and checking aspects of the micro-architecture's memory system, including the cache coherence protocol. Both Lustig et al. and Marnerkar et al. focus on TSO.

Lustig et al. [84] apply similar ideas to the specification and checking of concurrency aspects in the hardware/operating system interaction for virtual memory. Manerkar et al. [89] generate SystemVerilog assertions to check whether RTL implemenations satisfy the concurrency model of a more abstract micro-architectural specification. Trippel et al. [120] develop a tool that, supplied with a language-level concurrency model, a compilation scheme for the language for a particular ISA, and the concurrency model of a micro-architectural implementation of this ISA, checks whether the language compilation scheme and the hardware implementation together provide the language memory model guarantees. During the course of this work the authors identified issues in earlier versions of the RISC-V ISA's memory model specification: it failed to provide strong enough guarantees to support the C11 compilation scheme. To address the shortcomings of this early RISC-V memory model the RISC-V Memory Model Task Group, chaired by Daniel Lustig, was formed [26]. Manerkar et al. [90] develop a verification tool that proves an axiomatic concurrency specification of a microarchitectural implementation correct against an axiomatic architectural concurrency specification, which they apply to SC and TSO implementations.

Lustig et al. [83] develop a framework for specifying memory models and a tool for translating programs between different memory models. Wickerson et al. [122] and Bornholt and Torlak [33] develop tools to automatically find litmus tests that distinguish between axiomatically specified memory models, with the latter tool also synthesising axiomatic model specifications when given a "model sketch" and a set of litmus tests. Lustig et al. [85] also synthesise litmus tests from model definitions, in the Alloy framework [67].

10.4 Language concurrency models

There is much related work on relaxed-memory language-level concurrency; the following includes only some of the most related work, on the concurrency models of Java and C/C++11.

Java Gosling et al. [59] define the original Java concurrency model. Pugh [102] finds the Java concurrency model as defined in the original documentation prohibits certain compiler optimisations as implemented by an existing compiler and common prohibits programming idioms, discusses the issues in implementing Java above a relaxed memory processor architecture, and proposes fixes to the concurrency model. Manson and Pugh [91] propose a new operational Java concurrency model; Yang et al. [124] develop a formalisation based on this in their framework for operational concurrency models and model checking. Manson et al. [92] define a new axiomatic concurrency model for Java; the model is non-standard in that the validity of an execution may depend on the existence of a similar valid justifying execution. Aspinall and Sevcík [27] formalise the Java memory model in Isabelle. The authors prove the guarantee given by Java that data-race-free programs only have Sequentially Consistent executions; the authors propose fixes for errors in the memory model. Cenciarelli et al. [39] propose an alternative formalisation that integrates a semantics for the sequential behaviour of a Java fragment and note that the updated Java memory model still does not allow for certain desirable compiler optimisations. Sevcík and Aspinall [112] study optimisations under the Java memory model. The authors prove the correctness of certain

program transformations under Java, and find that some other common optimisations violate the Java memory model. Torlak et al. [119] develop a tool for executable axiomatic memory model specifications based on SAT solving and study formalisations of the Java memory model in the framework.

C/C++11 Batty et al. [30] clarify and formally specify the concurrency behaviour of the C/C++11 programming language and provide a formal specification in axiomatic style. The model is formalised in Isabelle/HOL and in the Cppmem executable tool that can be used to exhaustively check possible outcomes of concurrency programs. Nienhuis et al. [97] describe the difficulties in operationally specifying the behaviour matching that allowed by the axiomatic model. The authors describe a model that operationalises the axiomatic model, incrementally computing the allowed executions preserving the axiomatic model's axioms, and integrates the model with a sequential semantics for C11. Lahav et al. [74] propose a modification of the semantics of C/C++11's release/acquire fragment that provides stronger guarantees to programmers without requiring a change in the compilation scheme for Power and x86. An additional benefit is that it allows for a new operational model for this fragment of C11 proposed by the authors. The model introduces the idea of timestamps and views as used in the later Promising semantics and the Promising-ARM/RISC-V models of Chapter 9.

The C/C++11 concurrency model of Batty et al. [30], following the language documentation's concurrency specification, has the problem that it unintentionally allows certain concurrent executions with causality cycles in which values seem to appear "out of thin air". Batty et al. [32] identify difficulties in forbidding the undesirable thin-air behaviour while still allowing for typical compiler optimisation and an efficient compilation to relaxed memory processor architectures such as Power and ARM. Pichon-Pharabod and Sewell [100], Jeffrey and Riely [68], and Kang et al. [71] propose alternative concurrency semantics for C/C++11 to address this problem. The latter, which the work in Chapter 9 is based on, describes the allowed concurrency behaviour with an operational concurrency model in terms of notions of *views* (constraining the possible read values) and *promises* (capturing the behaviour of early propagation of writes). Section 10.5 compares the Promising C11 model with the Promising-ARM/RISC-V model of this thesis.

Linux, while implemented mostly in C, assumes different concurrency semantics from the one given by C; Alglave et al. [19] study Linux concurrency and define a model of the concurrency semantics assumed by Linux. Kokologiannakis et al. [73] develop a model checking algorithm and tool RCMC for a strengthened C11 semantics which in contrast to other tools is based on an axiomatic model's execution graph rather than operational model thread interleavings, and prove correctness and optimality results. Crary and Sullivan [46] propose an alternative view on concurrency programming. Whereas typical concurrent programs create memory model ordering using memory accesses that provide stronger ordering, or with dependencies implicit in the code, the authors propose a calculus whose memory model is weaker than existing architectures in order to be efficiently implementable and that allows programmers to explicitly annotate programs in the calculus with edges between commands to express the specific ordering the program relies on being preserved.

10.5 Promising-ARM/RISC-V compared to Promising C11

The Promising-ARM/RISC-V model borrows the main ideas from the Promising C11 semantics of Kang et al. [71]:

- a monotonic memory that records all writes seen,
- capturing the behaviour of the early execution of reads while executing them in program order by allowing them to read from old writes in memory,
- the concept of views to explain ordering constraints on reads and writes,
- explaining the out-of-order execution of writes using the notion of promises, and enforcing consistent executions using the concept of certification.

Due to the differences in the architecturally allowed behaviours of ARMv8 and RISC-V compared to the behaviours a concurrency model for a C-like programming language must allow, the two models differ in various other ways.

Syntactic vs semantic dependencies A major difference between Promising C11 and Promising-ARM/RISC-V is the notion of dependencies. A concurrency semantics for a C-like programming language has to be liberal enough to allow for the effects of compiler optimisations. Since compiler optimisations can break syntactic dependencies present in the source code of the program, a concurrency semantics for such a programming language cannot reasonably restrict the concurrency behaviour to respect all syntactic dependencies (see Batty et al. [32]). Hence, in order to be liberal enough to allow for such compiler optimistations, while at the same time preventing out-of-thin-air behaviours arising in a model that ignores any form of dependencies, the Promising C11 model has a *semantic* notion of dependencies. These are captured using the concept of promises: a thread is allowed to do an out-of-order write, a promise, if it can guarantee it will be able to later fulfil it; the certification of the ability to fulfil its promise is defined semantically, in terms of the possible executions of the thread under current memory (or future extensions of this memory).

The Promising-ARM/RISC-V model borrows the C11 model's notion of promise to explain the early execution of writes, and uses the same kind of semantic definition of certifiability in order to ensure consistent thread behaviours. Since in ARMv8 and RISC-V, however, *syntactic dependencies* do introduce memory ordering, the model tracks such syntactic dependencies using views attached to the registers in the thread state, and includes the dependency views into the view of a write when promising; when later a store fulfils the promise the dependencies constrain the store (via the view) to obey the ordering originating from the syntactic dependencies. So while the ARM/RISC-V model borrows the C11 model's certification mechanism, promises in the C11 model and promises in the ARM/RISC-V model are very different.

One consequence of this is also a difference relating to the certification of promises. Both models require a thread to certify its promises and re-certify all promises after each step, in order to prevent executions in which promises remain unfulfilled and to make the model execute incrementally [71]. The Promising C11 model also relies on the certification and re-certification to enforce semantic dependencies and prevent out-of-thin-air behaviours, which would be allowed without certification. In the ARM/RISC-V model, however, this is not the case: the model is

equivalent to the model without certification (Theorem 5).

Multicopy atomicity vs non-multicopy-atomicity Another important difference between the two models is multicopy-atomicity: ARMv8 and RISC-V have a multicopy atomic semantics, C11 does not. ARMv8/RISC-V's multicopy atomicity means there is a total order on all writes that the interaction between threads respects, enabling a model where the memory state is simply a list of write messages, and where writes consist only of address, value, and originating thread identifier. Since this memory state totally order all writes, regardless of the address, timestamps for different addresses are comparable. Hence, Promising-ARM/RISC-V has a simple notion of view: a view is a timestamp, an index into the list of writes that is memory.

Since C11 has a non-multicopy-atomic semantics it needs a more elaborate semantics for memory and views. C11's memory is a set of writes; each write in memory has a timestamp, but timestamps across locations are incomparable: in a non-multicopy-atomic model, just because one thread has "seen" a write with a certain timestamp to some address x, this does not mean it must also see another write with lower timestamp to a different address y. In fact, the semantics even allows promising a write at a timestamp that is lower than the currently maximal timestamp of a write *to the same address* in memory [see 71, 2+2W example]. Hence, timestamps are per location and a view is a timemap, mapping an address to a timestamp of some write in memory to that address. Moreover, to record enough information to implement stronger memory accesses and fences, for example release writes, writes in memory also contain a view (a timemap) that, when the write is read from, is included in the reading thread's views under certain conditions [71].

ARM/RISC-V's multicopy-atomicity is not, however, completely handled by this linearly ordered memory state: the architectures allow a write to become "visible" to its *originating thread* before becoming visible to others. This relaxation is handled using the thread state's forward bank, which keeps a record of the thread's last write for each location and the forward view of this write, allowing a read to obtain a smaller view when reading from a write by its own thread.

C11 atomic updates vs load/store exclusive instructions The language handled by the Promising C11 model has atomic updates/read-modify-write operations, such as fetch-and-add [71, Section 3]. ARMv8 has load/store exclusive instructions, RISC-V the analogous load reserve/store conditional. (ARMv8 and RISC-V also have atomic operations, but we do not currently handle them.)

As discussed by Kang et al. [71, Section 3] and in Section 9.2.5, these operations are special for promise certification, and Section 9.2.5 discusses the differences concerning these operations in the models: in the C11 model consistent executions are guaranteed using time intervals and future memory quantification; in ARMv8 the model can deadlock due to dependencies out of store exclusive instructions that the architecture does not preserve; in RISC-V tracking the syntactic dependencies via register views prevents such deadlocks since the architecture enforces the dependencies out of store conditionals. For ARMv8, we tried to solve the deadlock issues using some form of future memory quantification based on that of the C11 model but not found any solution that prevents all such deadlocks while not also preventing executions that

are architecturally allowed. Instead, Section F.1 presents a prototype extension of the model with more elaborate dependency tracking, locks, and a certification mechanism that takes the locks into account. This, however, adds significant complexity to an otherwise (we think) simpler model, and is not very satisfying.

Executability and the promise-first optimisation In ARM/RISC-V, the linear memory leads to a model in which timestamps are deterministic: when a write is promised it receives the next free timestamp in memory. In the Promising C11 model this is not the case: when promised, a write can be placed at any timestamp in memory that satisfies the view constraints and is timestamp-disjoint from other writes in memory. While in ARMv8/RISC-V the certification of a thread state only requires checking whether the thread has an execution under current memory to a thread state with empty promise set, in the C11 case certification involves quantifying over arbitrary future memories (extending the current one with more writes). The non-determinism in the timestamps and the future memory quantification (both due to the semantics of C11) make it harder to develop an executable model, such as the ARM/RISC-V one of Section 9.5, for C11.

The key optimisation used by the executable ARM/RISC-V model is the promise-first execution. It is not obvious whether this optimisation can be made to work in the C11 model; the C11 model has two features that may be in the way. First, the semantics explicitly requires the set of promises to be empty at the point of doing a release fence [71, Section 4]. This requirement would have to be lifted, as it prevents making any promises past a release fence. According to the authors, this requirement is for convenience rather than essential for the semantics [71, p. 6]. If it is not already equivalent, a semantics dropping this requirement can potentially be made equivalent to the C11 Promising semantics as-is by imposing additional constraints on the views (which is how all barriers are implemented in the ARM/RISC-V model). Second, to implement SC fences in Promising C11 the memory state has an additional component, the global timemap S, which is updated by SC fences. This means SC fences are “effectful” with respect to the memory state, and their ordering seems to matter [71, Section 4.2]. It is unclear how this would interact with promises in a promise-first execution of the model: in the ARM/RISC-V model, all non-write transitions are “pure” with respect to the memory state.

Chapter 11

Conclusion

This thesis tries to improve the understanding of the relaxed memory models of ARMv8, RISC-V, and (to a lesser degree) POWER, by developing simplifying abstractions of existing operational models of these architectures, and new operational models for the multicopy atomic ARMv8 and RISC-V architectures.

The shallow embedding of Sail into Lem leads to simpler instruction semantics for the Sail architecture models, for the behaviour of sequential programs, and, integrated into rmem, for that of concurrent programs. It has since been improved to produce better theorem prover output by Thomas Bauereiss who used it in an Isabelle proof about the architecture's virtual memory management in the sequential model.

The non-multicopy-atomic ARMv8 architecture has a complex concurrency semantics, partly due to the complexity of the storage subsystem's definitions concerning the propagation of events between threads. The NOP storage subsystem offers a simplification over Flowing and POP by abstracting from the explicit propagation of events between threads, and offers improved performance in enumerating the possible concurrency outcomes of small test programs by reducing the model's non-determinism. Extending the concurrency models of POWER and the non-multicopy-atomic ARMv8 architecture to cover memory accesses of mixed sizes leads to additional complexity in an already complicated semantics. Chapter 5 proposes definitions for notions of Sequential Consistency in a mixed-size setting for a characterisation of the simpler programming model that can be obtained in fully-barriered or release/acquire ARMv8 and POWER programs.

Due to ARMv8's shift to multicopy atomicity, NOP and part of the mixed-size SC results on ARMv8 are less relevant now. In particular, not only release/acquire but also fully-barriered ARMv8 programs should be sequentially consistent now (in the stronger original sense). But the results may still be interesting with respect to other non-multicopy-atomic concurrency semantics.

The main results presented in this text are concerning the multicopy atomic concurrency semantics of the revised ARMv8 and the new RISC-V architectures. Enabled by the revised ARMv8 concurrency architecture's simplifications, the Flat model gives a simpler specification of the concurrency behaviour of MCA ARMv8 by expressing it purely in terms of thread actions. This model was co-designed with ARM's official axiomatic model, and Chapter 7 studies their relation. The proof of equivalence between Flat and ARM's official axiomatic model for non-mixed-size programs is useful in three ways. Firstly, it gives more confidence in both models. Secondly, it should improve the understanding of the concurrency behaviour and both models by establishing the relation between two quite different formalisations. Thirdly, it means users of such models can choose between the two models, and pick whichever is best suited for their purpose.

Partly informed by our work on concurrency models for ARMv8 and due to our active participation in the RISC-V Memory Model Task Group, RISC-V has adopted a concurrency semantics

closely following that of ARMv8, but diverging from it in order to avoid issues arising from the operational modelling of load/store exclusive (load reserve/store conditional) instructions in ARMv8. The resulting Flat RISC-V model that adapts that of ARMv8 correspondingly avoids the ARMv8 model's deadlock issues, and the RISC-V ISA specification now includes, alongside axiomatic models closely following ARMv8's, the adapted Flat model.

The Promising-ARM/RISC-V models provide alternative, more abstract concurrency models for ARMv8 and RISC-V, that give up Flat's closer relation to hardware in order to provide a simpler programmer's concurrency model, expressing the concurrency semantics in a way that emphasises the thread-local execution of instructions in program order.

As a result, ARMv8 and RISC-V now have three equivalent models: the ARMv8-axiomatic model due to Will Deacon of ARM and the similar RISC-V-axiomatic concurrency models due to the Memory Model Task Group, the Flat ARMv8 and RISC-V models, and the Promising-ARM/RISC-V operational concurrency models.[1] The three equivalent presentations should improve the understanding of these complicated concurrency semantics, explained differently by each model, and each of the three styles of models may prove useful for certain purposes. In particular:

- Due to Flat's micro-architectural operational intuition it should offer a clearer relation to implementations. Hence, it should be useful both as a starting point for extending the concurrency semantics to cover more aspects of the architecture — such as virtual memory, self-modifying code, and system-level features — in a way that is informed by microarchitectural implementations, and as a model for validating hardware implementations: for trace comparison and bug-finding, and for proof. Currently the proof of equivalence between the axiomatic and the Flat model only covers ARMv8, and only the non-mixed-size programs supported by the axiomatic model; the proof is a hand-proof only and due to its size is hard to check for mistakes. The axiomatic models should be extended for mixed-size accesses, and the equivalence proof extended to cover mixed-size ARMv8 and RISC-V, and mechanised in a theorem prover.

  Some details of the mixed-size semantics for ARMv8 and RISC-V are still being clarified; extending the axiomatic model for mixed-size accesses depends on these details. The mixed-size models will likely be based on interpreting the current axiomatic models as ranging over byte-wise events — similar to what is seen for the mixed-size Sequential Consistency definition in Chapter 5 — and adding certain ordering and atomicity constraints. Mechanising the equivalence proof between the Flat operational model and the axiomatic models is likely a significant proof effort: while the proof relating Flat-axiomatic and ARMv8-axiomatic (or RISC-V-axiomatic when covering RISC-V) should lend itself well to mechanisation, the proofs involving the operational concurrency model will be complicated by the size and complexity of the Flat operational model.
- Promising-ARM/RISC-V gives up Flat's clearer relation with hardware, but should provide a simpler programmer-focused model. In this respect, it may provide an interesting basis for

[1] For RISC-V the Flat/Axiomatic equivalence is not proved, and there are experimental differences for a particular aspect of mixed-size programs that is still being clarified.

reasoning about concurrent software running above ARMv8 or RISC-V: e.g. Svendsen et al. [117] demonstrate the possibility of developing a program logic above a Promising-style model, for the case of the original C11 semantics. Perhaps more immediately practical, Promising-ARM/RISC-V may offer a good starting point for developing a model checker for concurrent ARMv8 and RISC-V programs: the early performance results for exhaustive exploration look promising, without much effort having been put into model checking performance improvements yet. The model, once extended to cover mixed-size programs, and with additional performance improvements may become a useful tool for concurrent software testing and verification purposes.

Appendix A

Single-instruction tests

Even though in the case of Power and ARMv8 the definition of the instruction behaviour is derived from the pseudocode description in the architecture manuals, the sequential semantics — as implemented by interpreter and shallow embedding — nonetheless has to be validated:[1]

1. The pseudocode description in the architecture manuals has no formal semantics. The Sail interpreter and the shallow embedding give meaning to the pseudocode, but this semantics may not match the intention of the ARMv8 and Power architectures. Moreover, both the Sail interpreter and some of the code for the Sail-to-Sail transformations of the shallow embedding are complex enough that they need testing for bugs in their implementation.
2. For some instructions the pseudocode description needed minor adaptation for the integration with the rmem concurrency model; it has to be checked that these adaptations do not introduce errors into the instructions semantics.
3. There may be errors in the process of extracting Sail definitions from the XML version of the Power reference manual, and in the process of manually transcribing the ARMv8 pseudocode descriptions into Sail definitions.
4. The vendor's pseudocode description may be incomplete or contain errors.

This section describes a tool for generating tests to compare the behaviour of the instruction semantics integrated in rmem with hardware, for Power and ARMv8 (not RISC-V or the other ISA models).

For validating the instruction semantics the tool generates a number of test programs for each instruction supported by rmem, each such program testing a particular aspect of an instruction's behaviour. The test programs are assembly programs that all follow the same procedure: each program first sets up the CPU's general and special purpose registers (*GPRs* and *SPRs*) as well as part of the memory in a suitable way for the instruction being tested; then it saves this CPU and memory state in a log file, runs the instruction instance, and saves the CPU and memory state again. The ELF model integrated with rmem allows running the same test programs on Power and ARM hardware and in rmem — standard ELF binaries produced with GCC. (In practice, a model and a hardware test differ in the details of the snapshot mechanism.) Therefore, using the log files produced by the test runs on hardware and in the model one can compare if the tested instructions cause rmem to take the same state transitions as real CPUs.

As the goal of the test programs is to compare the behaviour of an instruction on hardware and in the model, the tool has to generate the tests in such a way that exposes different variants of each instruction's behaviour: instructions often behave differently when applied to special immediate values or the values held by the registers it reads from, and have modes that can be selected by setting certain SPRs or the instruction's mode bits or mode strings. Therefore, in order

[1] The test generation has been described in Gray et al. [60] for POWER, and in Flur et al. [51] for ARMv8.

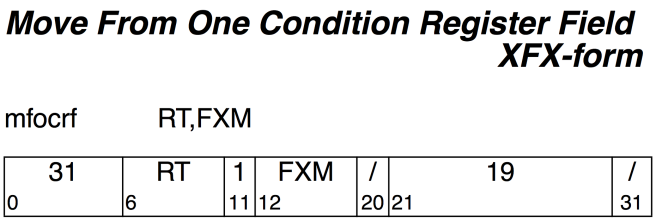


Figure A.1: Decode information for mfocrf

to produce good test programs the test generation tool must choose how to set up the system state before running the instruction (set up those GPRs and SPRs that affect the instruction behaviour in a suitable way) and which parameters (values for the instruction fields) the instruction should be run with.

For Power, the test generation tool does this based mainly on two sources of information: the instruction description in Sail, including its bitvector length and effect information, and the Power ISA's description of how the values encoded in the instruction fields will be used by the instruction. To illustrate this, consider the example of the `mfocrf` ("Move From One Condition Register Field") instruction that copies a field of the CR special purpose register to a GPR. Figure A.1 shows the description of this instruction's encoding taken from the Power ISA manual. This encoding contains the information about `mfocrf`'s instruction fields: their names and the vector length of their encoding. The information about the length of the instruction field's bitvector length is available through the type information in Sail's instruction description and determines the range of the values for the parameters that the instruction can be tested with. In this example, according to the type information, `mfocrf` has two parameters: RT, whose encoding uses five bits, and FXM with bitvector length eight. Moreover, the Power ISA manual contains a section [2, Section 1.6.28] describing for each instruction field how an instruction will use the values encoded in this field and what register or memory values an instruction with this field depends on. In the example of `mfocrf`, the description says that any instruction using the RT field interprets its value as the number of a general purpose register used as a target of the instruction, whereas FXM is a bitmask that selects a condition register field used as a source or target. Combining the two sources of information one learns what the instruction's behaviour depends on and what values are available for the testing. The tool uses this information to test different variants of an instruction's behaviour by trying different settings for all parameters or register and memory values that affect its behaviour; the values are selected partly randomly and partly exhaustively. In the example above the tool selects a number of random values for the instruction parameters within the range allowed by their type information as well as random values for the pre-test setup of the registers `RT` and `CR` and generates tests for all combinations of the selected values and register settings.

In the example of `mfocrf` both instruction fields have a "number value". Selecting random test values for parameters and registers of this kind appears a reasonable alternative to the (computationally infeasible) exhaustive testing of all possible values. Many instructions, however, have single-bit mode parameters or mode strings, or depend on bits of a special purpose register

that switch between two different variants of the instruction. In this case randomly selecting values does not provide good coverage of the instruction's behaviour. Instead, the tool tests these parameters or register values exhaustively. As a result the number of tests generated per instruction depends on the number and kind of the instruction's parameters: of the 7060 tests that are generated for the around 160 Power instructions, 1336 just test the four branch instructions, as their behaviour depends on a higher number of mode bits and special purpose registers for which all combinations are tested.

This approach to test generation allows the tool to work generically for almost all of the instructions currently supported by rmem; only 20 instructions, such as branch instructions or load/store multiple instructions, need special treatment. After using the tool to generate tests for the Power model, the ARMv8 Sail ISA model [51] required the same validation of the instruction semantics and the tool was adapted to support ARM. For ARM, the same general approach to testing is used. Although not much code is shared between the Power and the ARM test generation tools (most of the code deals with details of the architectures) the ARM tool only took around two weeks to finish. For the instructions covered by the hand-written ARMv8 Sail model the tool produces around 8400 tests for the around 220 instructions.

In the cases of both Power and ARMv8 the tests have revealed a number of mismatches between rmem's sequential semantics as described by the interpreter or the shallow embedding and the hardware behaviour. Many of these were due to subtle errors in the Sail interpreter implementation, the Sail to Lem translation, the library functions of the interpreter and the shallow embedding, and the interface between the interpreter and the concurrency model. The testing also found an incompleteness in the Power ISA document, where the textual description mentions special behaviour that is not reflected in the pseudocode [60, Section 7]. All these problems have since been resolved and rmem's interpreter and shallow embedding semantics match that of the hardware for all of the supported Power and ARMv8 instructions in the generated test binaries. A useful addition to this testing would be measuring the code coverage of the instruction semantics definitions by the instruction tests to see whether the selection of instruction parameters and values in the pre-test setup succeeds in exposing the different variants of the instruction (even though full coverage does not guarantee correct semantics). This has not yet been attempted for the Sail instruction testing.

Appendix B

NOP details

B.1 Subset property in POP

The example below[1] is only allowed in POP without subset property. The idea underlying the test is to enforce intermediate states in which the propagation of writes does not correspond to a valid tree topology. In order to do this, the test consists of two cycles: $C1$ with the events a, b, d, g, h, i, a and $C2$ with a, b, c, e, f, h, i, a. Either of these enforce a different topology: in order to allow $C1$, Threads 1 and 2 have to be more closely connected; for $C2$ (which has a WRC shape), Threads 0 and 1 have to be closer. They two cycles are not disjoint, however. In particular, the co edge from b to d connects events from both cycles, and the execution that allows this test brings POP into a state where b is Order-before d, b is propagated to Threads 0 and 1 (not propagated to Thread 2 in order for i to still read from the initial write), while d is propagated to Threads 1 and 2 (so as to resolve the address dependency to h and eventually i as early as possible without propagating a to Thread 2). Although the proof of soundness of NOP with respect to POP uses

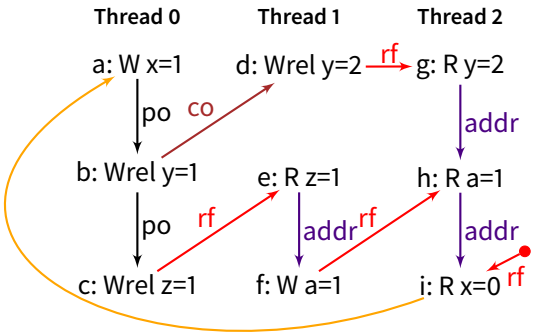


Figure B.1: Subset property test

the fact that in NOP all edges do have the subset property in the corresponding POP state of the simulation, NOP also allows the behaviour from this test.

B.2 Proof of Soundness of NOP

The following presents the proof of the soundness of NOP with respect to POP stated in Chapter 4. For simplicity, the proof assumes traces with no instruction restarts or discards (that would cause events to be deleted from the storage subsystem). Since NOP does not support mixed-size memory accesses, the proof assumes all accesses have the same size. Moreover, for simplicity, the proof assumes reads in the storage subsystem do not read from “initial memory”: i.e. every write in the

[1] Based on discussion with Shaked Flur

storage subsystem originates from a store instruction in the input program.

Let $s_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{n-1}} s_n = s$ be a valid finite POP trace from the initial state. Then define the corresponding NOP trace to be $s'_0 \xrightarrow{t'_0} \ldots \xrightarrow{t'_{n-1}} s'_n = s'$ where s'_0 is the initial state and

$$t'_i = \begin{cases} \emptyset \quad \textit{(stutter)} & t_i = \text{propagate } e \; tid \\ \text{accept } e \; tid & t_i = \text{accept } e \; tid \\ \text{rf-memory } B \; A \; w \; r & \text{satisfy-read } w \; r, \text{fully-propagated } s_i w, \\ \text{rf-segment } w \; r & \text{satisfy-read } w \; r, \neg\text{fully-propagated } s_i \; w \end{cases}$$

where B and A are the order-before/after sets derived from the previous POP state s_i:

$$B = \{w' | (w', w) \in s_i.\text{Order}, \text{address } w = \text{address } w'\}$$
$$A = \{w' | (w, w') \in s_i.\text{Order}, \text{address } w = \text{address } w'\}$$

Lemma 1. *For all $n \in \mathbb{N}$:*

(a) $s'_0 \xrightarrow{t'_0} \ldots \xrightarrow{t'_{n-1}} s'_n = s'$ *is a valid NOP trace.*

(b) $s_n \sim s'_n$

Proof. By induction on the trace length n.

Base case n=0 Let s be the POP initial state and s' the NOP initial state for the input program. Then trivially $s \gtrsim s'$: the Threads is initialised to the same set of thread ids, all other sets are empty (EProp pointwise empty). Since EProp is empty the subset property vacuously holds so that $s \sim s'$.

Step case n → n+1 Assume the induction hypothesis (IH) holds for the POP trace $s_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{n-1}} s_n$, show it also holds for $s_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$. By IH $s'_0 \xrightarrow{t'_0} \ldots \xrightarrow{t'_{n-1}} s'_n$ is a valid NOP trace and $s_n \sim s'_n$. Now there are four cases for t_n:

Propagate event: $t_n = \text{propagate } e \; tid'$. Since $t'_n = \emptyset$, extending the trace to $s'_0 \xrightarrow{t'_0} \ldots \xrightarrow{t'_{n-1}} s'_n \xrightarrow{t'_n} s'_{n+1}$ trivially results in a valid NOP trace (a), by IH. As $t'_n = \emptyset$ it is $s'_{n+1} = s'_n$, and since t_n only increases the EProp and Order sets $s_{n+1} \gtrsim s'_{n+1}$ holds:

$$s'_{n+1}.\text{Events} = s'_n.\text{Order} \subseteq s_n.\text{Order} \subseteq s_{n+1}.\text{Order} \qquad \text{(def and IH)}$$
$$s'_{n+1}.\text{EProp } tid = s'_n.\text{EProp } tid \subseteq s_n.\text{EProp } tid \subseteq s_{n+1}.\text{EProp } tid \quad \text{for all } tid \qquad \text{(def and IH)}$$
(other components unchanged)

So remains showing that the subset property holds for $n+1$ to establish $s_{n+1} \sim s'_{n+1}$. By induction hypothesis have

$$\forall (e', e) \in s'_n.\text{Order}.e \in s_n.\text{EProp } tid \Rightarrow e' \in s_n.\text{EProp } tid$$

Now assume the subset property does not hold for s_{n+1} and s'_{n+1}. Then there exists an edge $(e_1, e_2) \in s'_{n+1}.\text{Order} = s'_n.\text{Order}$ and thread id tid such that $e_2 \in s_{n+1}.\text{EProp } tid$ and

$e_1 \notin s_{n+1}.\mathsf{EProp}\ tid$. Since the subset property holds for s_n and s'_n, and since by definition of the propagate transition only $\mathsf{EProp}\ tid$ changes to include e, it must be $e_2 = e$ and $tid = tid'$. But for t_n to be enabled in s_n, propagate-cand $s_n\ e\ tid'$ must hold, requiring among others

$$\forall e'.(e', e) \in s_n.\mathsf{Order} \Rightarrow e' \in s_n.\mathsf{EProp}\ tid\text{'}.$$

But by $s'_{n+1}.\mathsf{Order} = s'_n.\mathsf{Order} \subseteq s_n.\mathsf{Order}$ it is $(e_1, e) \in s_n.\mathsf{Order}$ and therefore (instantiating $e' = e_1$) it is $e_1 \in s_n.\mathsf{EProp}\ tid' \subseteq s_{n+1}.\mathsf{EProp}\ tid'$, contradicting the assumption.

Accept event: Since NOP can always accept an event, extending the trace with the transition t'_n for $t_n = \mathsf{accept}\ e\ tid'$ to $s'_0 \xrightarrow{t'_0} \ldots \xrightarrow{t'_{n-1}} s'_n \xrightarrow{t'_n} s'_{n+1}$ trivially results in a valid NOP trace (a). For (b), have:

$$\begin{aligned}
s'_{n+1}.\mathsf{Events} &= s'_n.\mathsf{Events} \cup \{e\} && \text{(def)}\\
&= s_n.\mathsf{Events} \cup \{e\} && \text{(IH)}\\
&= s_{n+1}.\mathsf{Events} && \text{(def)}
\end{aligned}$$

$$\begin{aligned}
s'_{n+1}.\mathsf{Order} &= (s'_n.\mathsf{Order} \cup \{(e', e) | e' \in s'_n.\mathsf{EProp}\ tid', \neg\mathsf{reorder}\ e'\ e\})^+ && \text{(def)}\\
&\subseteq (s_n.\mathsf{Order} \cup \{(e', e) | e' \in s'_n.\mathsf{EProp}\ tid', \neg\mathsf{reorder}\ e'\ e\})^+ && \text{(IH, + monotonic)}\\
&\subseteq (s_n.\mathsf{Order} \cup \{(e', e) | e' \in s_n.\mathsf{EProp}\ tid', \neg\mathsf{reorder}\ e'\ e\})^+ && \text{(IH, + monotonic)}\\
&= s_{n+1}.\mathsf{Order} && \text{(def)}
\end{aligned}$$

$$\begin{aligned}
s'_{n+1}.\mathsf{EProp}\ tid' &= s'_n.\mathsf{EProp}\ tid' \cup \{e\} && \text{(def)}\\
&\subseteq s_n.\mathsf{EProp}\ tid' \cup \{e\} && \text{(IH)}\\
&= s_{n+1}.\mathsf{EProp}\ tid' && \text{(def)}\\
s'_{n+1}.\mathsf{EProp}\ tid &\subseteq s_{n+1}.\mathsf{EProp}\ tid && \text{(IH, for other } tid\text{)}
\end{aligned}$$

$$s'_{n+1}.\mathsf{Threads} = s'_n.\mathsf{Threads} = s_n.\mathsf{Threads} = s_{n+1}.\mathsf{Threads} \qquad \text{(def and IH)}$$

So have $s_{n+1} \gtrsim s'_{n+1}$; for $s_{n+1} \sim s'_{n+1}$ remains to show that the subset property holds for s_{n+1}. By induction hypothesis it holds for s_n and s'_n:

$$\forall (e', e) \in s'_n.\mathsf{Order}.e \in s_n.\mathsf{EProp}\ tid \Rightarrow e' \in s_n.\mathsf{EProp}\ tid$$

Now assume the subset property does not hold for s_{n+1} and s'_{n+1}. Then there exists an edge $(e_1, e_2) \in s'_{n+1}.\mathsf{Order}$ and a thread id tid such that $e_2 \in s_{n+1}.\mathsf{EProp}\ tid$ and $e_1 \notin s_{n+1}.\mathsf{EProp}\ tid$. Then there are two cases: (I) $(e_1, e_2) \in s'_n.\mathsf{Order}$ or (II) not.

I As by induction the subset property holds for s_n and s'_n, the transition t_n must have propagated an event e_2 to some thread tid such that $e_1 \notin s_{n+1}.\mathsf{EProp}\ tid$. But because t_n only propagates e this must be $e_2 = e$. But this contradicts the assumption $(e_1, e_2) \in s'_n.\mathsf{Order} \subseteq s_n.\mathsf{Order}$: e_2 was not mentioned by the storage subsystem state s_n.

II Let $New = (\{(e', e) | e' \in s'_n.\mathsf{EProp}\ tid, \neg\mathsf{reorder}\ e'\ e\} \cup s'_n.\mathsf{Order})^+ \setminus s'_n.\mathsf{Order}$. So have $(e_1, e_2) \in New$. Since e is not mentioned by $s'_n.\mathsf{Order}$, all edges $(e_a, e_b) \in New$ have $e_b = e$. As *tid* is the only thread id $\widetilde{tid}$ for which $e \in s'_{n+1}.\mathsf{EProp}\ \widetilde{tid}$ it must be $e_1 \notin s'_{n+1}.\mathsf{EProp}\ tid$, by assumption that the subset property is violated. So have $e_1 \notin s_{n+1}.\mathsf{EProp}\ tid$ and $e_2 = e$.

As *New* only has edges (e_a, e) (edges pointing into e), every such edge in *New* can be decomposed into (e_a, e_c) and (e_c, e) for some e_c such that

$$(e_c, e) \in \{(e', e) | e' \in s'_n.\mathsf{EProp}\ tid, \neg\mathsf{reorder}\ e'\ e\}$$

and such that it is either $e_a = e_c$ (reflexive edge) or (e_a, e_c) in $s'_n.\mathsf{Order}$. By definition then $e_c \in s'_n.\mathsf{EProp}\ tid \subseteq s_n.\mathsf{EProp}\ tid$ holds.

So let $(e_1, e_3), (e_3, e)$ be this decomposition for (e_1, e). Then it is $e_3 \in s_n.\mathsf{EProp}\ tid$ and there are two cases: (IIa) $e_1 = e_3$ or (IIb) $(e_1, e_3) \in s'_n.\mathsf{Order}$.

IIa Then immediately have $e_1 \in s_n.\mathsf{EProp}\ tid$.

IIb Then as by induction hypothesis the subset property holds for n it is also $e_1 \in s_n.\mathsf{EProp}\ tid$.

So in both cases $e_1 \in s_n.\mathsf{EProp}\ tid \subseteq s_{n+1}.\mathsf{EProp}\ tid$, contradiction. Therefore the subset property holds for $n+1$ and $s_{n+1} \sim s'_{n+1}$.

rf-segment t_n = satisfy-read w r and ¬fully-propagated s_n w. For (a) only have to show that rf-segment-cand s'_n w r holds, as the proof of (b) will show that $s'_{n+1}.\mathsf{Order} \subseteq s_{n+1}.\mathsf{Order}$, and by the acyclicity of s_{n+1} follows the acyclicity of s'_{n+1}, and thus that t'_n is enabled in s'_n. So have to show:

1. w is a write intersecting the read r. This follows from the fact that t_n is enabled in s_n and POP's satisfy-read-cand condition.
2. $(r, w) \notin s'_n.\mathsf{Order}$. By definition of satisfy-read-cand in POP, $(w, r) \in s_n.\mathsf{Order}$. Since $s_n.\mathsf{Order}$ acyclic $(r, w) \notin s_n.\mathsf{Order} \subseteq s'_n.\mathsf{Order}$.
3. r has not been satisfied yet. By construction of the NOP trace in s_n and s'_n the set of reads that are satisfied are the same. Since t_n is enabled in s_n, r cannot be satisfied in s_n, and therefore also not in s'_n.
4. r and w are not a read acquire and a write release. This follows from the definition of POP's satisfy-read-cand for t_n.

So the rf-segment-cand condition holds.

For (b) first show $s_{n+1} \succsim s'_{n+1}$. Let $\widetilde{s}'_n$ be the state constructed after step 1 and before step 2 of rf-segment-action. Show $s_n \succsim \widetilde{s}'_n$.

$$\begin{aligned}
\widetilde{s_n}'.\mathsf{Events} = s_n'.\mathsf{Events} &= s_n.\mathsf{Events} && \text{(def, IH)}\\
\widetilde{s_n}'.\mathsf{Threads} = s_n'.\mathsf{Threads} &= s_n.\mathsf{Threads} && \text{(def, IH)}\\
\widetilde{s_n}'.\mathsf{Order} &= (s_n'.\mathsf{Order} \cup \{(w,r)\})^+ && \text{(def)}\\
&\subseteq (s_n.\mathsf{Order} \cup \{(w,r)\})^+ && \text{(IH)}\\
&\subseteq s_n.\mathsf{Order} && ((w,r) \in s_n.\mathsf{Order} \text{ as } t_n \text{ enabled in } s_n)\\
\widetilde{s_n}'.\mathsf{EProp}\ tid = s_n'.\mathsf{EProp}\ tid &\quad \text{for } tid \notin T_{wr} && \text{(def)}\\
&\subseteq s_n.\mathsf{EProp}\ tid \quad \text{for } tid \notin T_{wr} && \text{(IH)}
\end{aligned}$$

Still have to show

$$\widetilde{s_n'}.\mathsf{EProp}\ tid \subseteq s_n.\mathsf{EProp}\ tid \quad \text{for } tid \in T_{wr} = \{tid | \{w,r\} \cap s.\mathsf{EProp}\ tid \neq \emptyset\}.$$

Since $T_{wr} = \{tid | r \in s_n'.\mathsf{EProp}\ tid \vee w \in s_n'.\mathsf{EProp}\ tid\}$, by IH it is $r \in s_n.\mathsf{EProp}\ tid \vee w \in s_n.\mathsf{EProp}\ tid$. By definition of POP's read-cand the events w and r are propagated to the same threads, so have: $\{w,r\} \subseteq s_n.\mathsf{EProp}\ tid$.
By definition $\widetilde{s_n'}.\mathsf{EProp}\ tid = s_n'.\mathsf{EProp}\ tid \cup \{w,r\} \cup B_{wr}$. By IH $s_n'.\mathsf{EProp}\ tid \subseteq s_n.\mathsf{EProp}\ tid$ and have $\{w,r\} \subseteq s_n.\mathsf{EProp}\ tid$, so only have to show $B_{wr} = \{e | (e,w) \in s_n'.\mathsf{Order} \vee (e,r) \in s_n'.\mathsf{Order}\} \subseteq s_n.\mathsf{EProp}\ tid$. Let $e \in B_{wr}$. Then either (a) $(e,w) \in s_n'.\mathsf{Order}$ or (b) $(e,r) \in s_n'.\mathsf{Order}$.

(a) Have $w \in s_n.\mathsf{EProp}\ tid$. So by IH (subset property) have $e \in s_n.\mathsf{EProp}\ tid$.

(b) Have $r \in s_n.\mathsf{EProp}\ tid$. By IH (subset property) then have $e \in s_n.\mathsf{EProp}\ tid$.

All in all, have $\widetilde{s_n'}.\mathsf{EProp}\ tid \subseteq \widetilde{s_n}.\mathsf{EProp}\ tid$ for all tid and therefore $s_n \gtrsim \widetilde{s_n'}$.
Now to show $s_n \sim \widetilde{s_n'}$ still have to show that the subset property holds. To this end, show first prove the following lemma.

Lemma 2. *Let s be a POP state and s' a NOP state. Assume*

$$\forall (e',e) \in s'.Order.e \in s.EProp\ tid \Rightarrow e' \in s.EProp\ tid,$$

and assume an edge $(\widetilde{e'},\widetilde{e})$ with

$$\widetilde{e} \in s.EProp\ tid \Rightarrow \widetilde{e'} \in s.EProp\ tid.$$

Then

$$\forall (e',e) \in (s'.Order \cup \{(\widetilde{e'},\widetilde{e})\})^+.e \in s.EProp\ tid \Rightarrow e' \in s.EProp\ tid.$$

Proof. Only have to show $y \in s.\mathsf{EProp}\ tid \Rightarrow x \in \mathsf{EProp}\ tid$ for edges $(x,y) \notin s'.\mathsf{Order}$ and for $(x,y) \neq (\widetilde{e'},\widetilde{e})$. Assume $y \in s.\mathsf{EProp}\ tid$ and show $x \in s.\mathsf{EProp}\ tid$ by case distinction of the possible forms of (x,y). By definition of the transitive closure, and since $s'.order$ is transitively closed, these are:

- (x, y) with $(x, \widetilde{e'}) \in s'.\mathsf{Order}$ and $(\widetilde{e}, y) \in s'.\mathsf{Order}$. Then from $y \in s.\mathsf{EProp}\ tid$ follows $\widetilde{e} \in s.\mathsf{EProp}\ tid$ since the subset property holds for $s'.\mathsf{Order}$ edges; from $\widetilde{e} \in s.\mathsf{EProp}\ tid$ follows $x \in s.\mathsf{EProp}\ tid$ subset property of $s'.\mathsf{Order}$ again.
- $(x, \widetilde{e})$ with $(x, \widetilde{e'}) \in s'.\mathsf{Order}$. Then from $\widetilde{e} \in \mathsf{EProp}\ tid$ follows $\widetilde{e'} \in \mathsf{EProp}\ tid$ by assumption; and from this $x \in \mathsf{EProp}\ tid$ since $(x, \widetilde{e'}) \in s'.\mathsf{Order}$.
- $(\widetilde{e'}, y)$ with $(\widetilde{e}, y) \in s'.\mathsf{Order}$. Then from $y \in \mathsf{EProp}\ tid$ follows $\widetilde{e} \in s.\mathsf{EProp}\ tid$ as $(\widetilde{e}, y) \in s'.\mathsf{Order}$; and from this by assumption $\widetilde{e'} \in s.\mathsf{EProp}\ tid$.

□

By induction hypothesis have $s_n \sim s'_n$. So according to Lemma 2 to show the subset property holds for $\widetilde{s'_n}$ and s_n only have to show it holds for $\{w, r\}$ as well. But by definition of POP's read-cand it is $w \in s_n.\mathsf{EProp}\ tid \Leftrightarrow r \in s_n.\mathsf{EProp}\ tid$ for all tid. So $s_n \sim \widetilde{s'_n}$.
Now show $s_{n+1} \precsim s'_{n+1}$.

$$
\begin{aligned}
s'_{n+1}.\mathsf{Events} &= \widetilde{s'_n}.\mathsf{Events} \setminus \{r\} && \text{(def)}\\
&= s_n.\mathsf{Events} \setminus \{r\} && (s_n \precsim \widetilde{s'_n})\\
&= s_{n+1}.\mathsf{Events} && \text{(def)}\\
s'_{n+1}.\mathsf{Threads} &= \widetilde{s'_n}.\mathsf{Threads} && \text{(def)}\\
&= s_n.\mathsf{Threads} && (s_n \precsim \widetilde{s'_n})\\
&= s_{n+1}.\mathsf{Threads} && \text{(def)}\\
s'_{n+1}.\mathsf{Order} &= ((\widetilde{s'_n}.\mathsf{Order} \downharpoonright s'_{n+1}.\mathsf{Events}) \cup (B_r \times \{w\}))^+ && \text{(def)}\\
&= ((\widetilde{s'_n}.\mathsf{Order} \downharpoonright s_{n+1}.\mathsf{Events}) \cup (B_r \times \{w\}))^+ && \text{(above)}\\
&= ((\widetilde{s'_n}.\mathsf{Order} \downharpoonright s_{n+1}.\mathsf{Events}) \cup ((\{e | (e, r) \in \widetilde{s'_n}.\mathsf{Order}\} \setminus \{w\}) \times \{w\}))^+ && (\text{def } B_r)\\
&= ((\widetilde{s'_n}.\mathsf{Order} \downharpoonright s_{n+1}.\mathsf{Events}) \cup \{(e, w) | (e, r) \in \widetilde{s'_n}.\mathsf{Order}, e \neq w\})^+\\
&= ((\widetilde{s'_n}.\mathsf{Order} \downharpoonright s_{n+1}.\mathsf{Events})\\
&\quad \cup \{(e, w) | (e, r) \in \widetilde{s'_n}.\mathsf{Order}, e \neq w, (w, e) \notin s_n.\mathsf{Order}\})^+ && \text{(below)}\\
&\subseteq ((s_n.\mathsf{Order} \downharpoonright s_{n+1}.\mathsf{Events})\\
&\quad \cup \{(e, w) | (e, r) \in s_n.\mathsf{Order}, e \neq w, (w, e) \notin s_n.\mathsf{Order}\})^+ && (s_n \precsim \widetilde{s'_n})\\
&= s_{n+1}.\mathsf{Order} && \text{(def)}\\
s'_{n+1}.\mathsf{EProp}\ tid &= \widetilde{s'_n}.\mathsf{EProp}\ tid \setminus \{r\} && \text{(def)}\\
&= s_n.\mathsf{EProp}\ tid \setminus \{r\} && (s_n \precsim \widetilde{s'_n})\\
&= s_{n+1}.\mathsf{EProp}\ tid && \text{(def)}
\end{aligned}
$$

Still have to justify

$$
\begin{aligned}
&((\widetilde{s'_n}.\mathsf{Order} \downharpoonright s_{n+1}.\mathsf{Events}) \cup \{(e, w) | (e, r) \in \widetilde{s'_n}.\mathsf{Order}, e \neq w\})^+\\
&= ((\widetilde{s'_n}.\mathsf{Order} \downharpoonright s_{n+1}.\mathsf{Events}) \cup \{(e, w) | (e, r) \in \widetilde{s'_n}.\mathsf{Order}, e \neq w, (w, e) \notin s_n.\mathsf{Order}\})^+
\end{aligned}
$$

by showing

$$(e, r) \in \widetilde{s'_n}.\mathsf{Order} \wedge e \neq w \Rightarrow (w, e) \notin s_n.\mathsf{Order}.$$

Let $(e,r) \in \widetilde{s'_n}.\mathsf{Order}$ such that $e \neq w$, and assume $(w,e) \in s_n.\mathsf{Order}$. From $(e,r) \in \widetilde{s'_n}.\mathsf{Order}$ have $(e,r) \in s_n.\mathsf{Order}$ by $s_n \gtrsim \widetilde{s'_n}$. Since $s_n.\mathsf{Order}$ is acyclic have $e \neq r$.
By $s_n \sim \widetilde{s'_n}$ from the subset property have that in s_n the event e is propagated to all threads r is propagated to. Since in s_n the read r is propagated at least to one thread, have $\exists tid.\{e,r\} \subseteq s_n.\mathsf{EProp}\ tid$. Because of $(w,e) \in s_n.\mathsf{Order}$ and $(e,r) \in s_n.\mathsf{Order}$ POP's read-cand requires for t_n to be enabled in s_n the following, among others:

$$\exists tid.\{e,r\} \subseteq s_n.\mathsf{EProp}\ tid \Rightarrow \mathsf{fully\text{-}propagated}\ s_n\ e \wedge \ldots$$

So fully-propagated s_n e. But by definition of fully-propagated and $(w,e) \in s_n.\mathsf{Order}$ also fully-propagated s_n w, contradicting the assumption that w is not fully propagated in s_n, so no such e with $(e,r) \in \widetilde{s'_n}.\mathsf{Order} \wedge e \neq w \wedge (w,e) \in s_n.order$ can exist.
Now have $s_{n+1} \gtrsim s'_{n+1}$ and remains to show that the subset property holds for $s_{n+1} \sim s'_{n+1}$. First show $s_n \sim \widetilde{s'_n}$. Already showed $s_n \gtrsim s'_n$, so still have to show the subset property holds for $\widetilde{s'_n}$ and s_n. By definition $\widetilde{s'_n}.\mathsf{Order} = (s'_n.\mathsf{Order} \cup \{(w,r)\})^+$, and since the subset property holds for s'_n and s_n, by Lemma 2 only have to show (w,r) satisfies $r \in s_n.\mathsf{EProp}\ tid \Rightarrow w \in s_n.\mathsf{EProp}\ tid$. But this follows from the definition of POP's read-cand: since t_n is enabled in s_n the events w and r are propagated to the same threads.
So remains showing $s_{n+1} \gtrsim s'_{n+1}$. Define the NOP state s''_{n+1} to be s'_{n+1} with $\mathsf{Order} = \widetilde{s'_n}.order \restriction s'_{n+1}.\mathsf{Events}$. Then

$$\forall (e',e) \in s''_{n+1}.\mathsf{Order}.e \in s_{n+1}.\mathsf{EProp}\ tid \Rightarrow e' \in s_{n+1}.\mathsf{EProp}\ tid :$$

already showed $\forall (e',e) \in \widetilde{s_n}.\mathsf{Order}.e \in s_n.\mathsf{EProp}\ tid \Rightarrow e' \in s_n.\mathsf{EProp}\ tid$, by definition of s_{n+1} it is $s_{n+1}.\mathsf{EProp}\ tid = s_n.\mathsf{EProp}\ tid \setminus \{r\}$ and $s''_{n+1}.\mathsf{Order} \subseteq \widetilde{s'_n}.\mathsf{Order}$ has no edges (r,e).
With this definition of s''_{n+1}, it is $s'_{n+1} = (s''_{n+1}$ with $\mathsf{Order} = (s''_{n+1}.\mathsf{Order} \cup (B_r \times \{w\}))^+)$ for B_r taken from the definition of s_{n+1}. Now by Lemma 2 showing the subset property holds for s'_{n+1} and s_{n+1} only requires showing that the edges $B_r \times \{w\}$ satisfy the subset property: show

$$\forall (e,w) \in B_r \times \{w\}.w \in s_{n+1}.\mathsf{EProp}\ tid \Rightarrow e \in s_{n+1}.\mathsf{EProp}\ tid.$$

So let $(e,w) \in B_r \times \{w\}$ and tid some thread id. By definition of B_r it is $(e,r) \in \widetilde{s'_n}.\mathsf{Order}$ and $e \neq w$, and it must be $e \neq r$ since otherwise $(r,r) \in \widetilde{s'_n}.\mathsf{Order} \subseteq s_n.\mathsf{Order}$ and $s_n.\mathsf{Order}$ is acyclic.

$$\begin{aligned}
w \in s_{n+1}.\mathsf{EProp}\ tid &\Rightarrow w \in s_n.\mathsf{EProp}\ tid && (t_n \text{ does not change the propagation of } w)\\
&\Rightarrow r \in s_n.\mathsf{EProp}\ tid && (\text{by POP's read-cand, since } t_n \text{ enabled in } s_n)\\
&\Rightarrow e \in s_n.\mathsf{EProp}\ tid && ((e,r) \in \widetilde{s'_n}.\mathsf{Order} \text{ and } s_n \sim \widetilde{s'_n})\\
&\Rightarrow e \in s_{n+1}.\mathsf{EProp}\ tid && (\text{def } t_n,\ e \neq r)
\end{aligned}$$

So have $s_{n+1} \sim s'_{n+1}$.
Before moving to the next case, rf-memory, prove the following lemma:

Lemma 3. *Let s be a state in a finite POP trace from the initial state and $e, e' \in s.Events$ be two same-address events with $e \in s.EProp\ tid$ for all tid. Then $(e, e') \in s.Order$ or $(e', e) \in s.Order$.*

Proof. Since e is propagated to all threads, e and e' are propagated to at least one common thread. Let s' be the state before s where for the first time e and e' are propagated to a common thread *tid*, and t' the transition in which s' is reached. Then t' is the propagate-event transition for e or for e'. Since e and e' are to the same address, reorder e e' and reorder e' e do not hold, and t' adds (e, e') or (e', e) into s'.Order. Assume it is (e, e').
Then $(e, e') \in s$.Order. No POP transition except the satisfy-read transition deletes Order edges. The satisfy-read transition preserves (e, e'): assume POP takes the transition $t_x =$ satisfy-read w_x r_x from some state s_x to the state s'_x where $(e, e') \in s_x$.Order. Show $(e, e') \in s'_x$.Order.
By definition of t_x it is:

$$s'_x.\mathsf{Order} = (s_x.\mathsf{Order} \downharpoonright s'_x.\mathsf{Events} \cup \{(e, w_x) | (e, r_x) \in s_x.\mathsf{Order}, e \neq w_x, (w_x, e)) \notin s_x.\mathsf{Order}\})^+$$

It can be neither $e = r_x$ nor $e' = r_x$. Assume it is, then t_x deletes r_x from Events. But by assumption $\{e, e'\} \subseteq s$.Events and s after s'_x in the POP trace, contradiction. So assume $e \neq r_x$ and $e' \neq r_x$. Since t_x removes at most $r_x \notin \{e, e'\}$ from s_x.Events, it is $(e, e') \in s'_x$.Order. Therefore, since no POP transition deletes the edge (e, e') from Order it is $(e, e') \in s$.Order. The case $(e', e) \in s'$.Order is symmetrical. □

rf-memory $t_n =$ satisfy-read w r and fully-propagated s_n w.
As in the previous case, for (a) only have to show that rf-memory-cand holds as the proof of (b) will show that $s'_{n+1}.\mathsf{Order} \subseteq s_{n+1}.\mathsf{Order}$: then from the acyclicity of $s_{n+1}.\mathsf{Order}$ follows that acyclicity of $s'_{n+1}.\mathsf{Order}$ and that t'_n is enabled in s'_n. So have to show:

1. w is write to the same address as r. This follows by definition of POP's read-cand, since t_n enabled in s_n.
2. $(r, w) \notin s'_n.\mathsf{Order}$. By definition of POP's read-cand it is $(w, r) \in s_n.\mathsf{Order}$. By acyclicity of $s_n.\mathsf{Order}$ it is $(r, w) \notin s_n.\mathsf{Order}$. So the proof (below) of $s'_n.\mathsf{Order} \subseteq s_n.\mathsf{Order}$ will imply $(r, w) \notin s'_n.\mathsf{Order}$
3. r is not satisfied in s'_n. By definition of POP's read-cand r is not satisfied in s_n, so by construction of the NOP trace, r is also not satisfied in s'_n.
4. A and B as defined above partition the set of all writes to the same address as w. This follows by Lemma 3: by assumption w is fully propagated in s_n, and therefore it is $(w, w') \in s_n.\mathsf{Order}$ or $(w', w) \in s_n.\mathsf{Order}$ for every same-address write w'; it cannot be both, since $s_n.\mathsf{Order}$ acyclic. Therefore A and B partition the set of writes to the same address as w.

For (b) have to show $s_{n+1} \sim s'_{n+1}$. By definition of t'_n:

$$\begin{aligned}
&\text{let } B = \{w' | (w', w) \in s_n.\mathsf{Order}, \mathsf{address}\ w = \mathsf{address}\ w'\} \\
&\text{let } A = \{w' | (w, w') \in s_n.\mathsf{Order}, \mathsf{address}\ w = \mathsf{address}\ w'\} \\
s'_{n+1}.\mathsf{Order} = &(s'_n.\mathsf{Order} \cup \{(w, r)\} \\
&\cup \{(r, e) | (w, e) \in s'_n.\mathsf{Order}, \mathsf{is\text{-}write}\ e, \mathsf{address}\ e = \mathsf{address}\ r\} \\
&\cup \{(e, w) | (e, r) \in s'_n.\mathsf{Order}, \mathsf{is\text{-}write}\ e, \mathsf{address}\ e = \mathsf{address}\ w, w \neq e\} \\
&\cup B \times \{w\} \\
&\cup \{r\} \times A)^+
\end{aligned}$$

First prove $s'_{n+1}.\mathsf{Order} = (s'_n.\mathsf{Order} \cup \{(w, r)\} \cup B \times \{w\} \cup \{r\} \times A)^+$ by showing

(a) $EW = \{(e, w) | (e, r) \in s'_n.\mathsf{Order}, \mathsf{is\text{-}write}\ e, \mathsf{address}\ e = \mathsf{address}\ w, w \neq e\} \subseteq B \times \{w\}$

(b) $RE = \{(r, e) | (w, e) \in s'_n.\mathsf{Order}, \mathsf{is\text{-}write}\ e, \mathsf{address}\ e = \mathsf{address}\ r\} \subseteq \{r\} \times A$

(a) Let $(e, w) \in EW$. Then e is a same-address write with $(e, r) \in s'_n.\mathsf{Order}$ and $e \neq w$. By IH it is $s_n \sim s'_n$, so $(e, r) \in s_n.\mathsf{Order}$. By assumption w is fully-propagated, so by Lemma 3 it must be $(w, e) \in s_n.\mathsf{Order}$ or $(e, w) \in s_n.\mathsf{Order}$.
It cannot be $(w, e) \in s_n.\mathsf{Order}$: since w is fully propagated it is propagated to a common thread with e, and have $\{(w, e), (e, r)\} \subseteq s_n.\mathsf{Order}$. Then by POP's read-cand e must not be a same-address write, contradiction.
So it must be $(e, w) \in s_n.\mathsf{Order}$. But then $e \in B$, by definition of B, and $(e, w) \in B \times \{w\}$.

(b) Let $(r, e) \in RE$. Then e is a same-address write with $(w, e) \in s'_n.\mathsf{Order}$. By $s_n \sim s'_n$ also have $(w, e) \in s_n.\mathsf{Order}$, and therefore also $e \in A$ and $(r, e) \in \{r\} \times A$.

So have $s'_{n+1}.\mathsf{Order} = (s'_n.\mathsf{Order} \cup \{(w, r)\} \cup B \times \{w\} \cup \{r\} \times A)^+$. By definition of POP's satisfy-read-cand it is $\{(w, r)\} \in s_n.\mathsf{Order}$, by IH it is $s'_n.\mathsf{Order} \subseteq s_n.\mathsf{Order}$, and by definition of t_n it is $s_{n+1}.\mathsf{Order} = s_n.\mathsf{Order}$. So in order to show that $s'_{n+1}.\mathsf{Order} \subseteq s_{n+1}.\mathsf{Order}$ remains to show to things:

(a) $B \times \{w\} \subseteq s_n.\mathsf{Order}$. This is immediately true by definition of B.

(b) $\{r\} \times A \subseteq s_n.\mathsf{Order}$. Let $(r, e) \in \{r\} \times A$. By definition of A the event e is a write to the same address as r and w with $(w, e) \in s_n.\mathsf{Order}$. By definition of POP's satisfy-read-cand r is propagated to the same threads as w and w is fully-propagated. Therefore by Lemma 3 it is $(e, r) \in s_n.\mathsf{Order}$ or $(r, e) \in s_n.\mathsf{Order}$.
Assume $(e, r) \in s_n.\mathsf{Order}$. But then $\{(w, e), (e, r)\} \subseteq s_n.\mathsf{Order}$, and POP's satisfy-read-cand requires e not to be a same address write, contradiction. So it must be $(r, e) \in s_n.\mathsf{Order}$.

Therefore $s_{n+1}.\mathsf{Order} \subseteq s_{n+1}.\mathsf{Order}$. Also have

$$s'_{n+1}.\mathsf{Events} = s'_n.\mathsf{Events} = s_n.\mathsf{Events} = s_{n+1}.\mathsf{Events} \qquad \text{(def, IH, def)}$$

$$s'_{n+1}.\mathsf{Threads} = s'_n.\mathsf{Threads} = s_n.\mathsf{Threads} = s_{n+1}.\mathsf{Threads} \qquad \text{(def, IH, def)}$$

So for $s_{n+1} \sim s'_{n+1}$ remains showing $s'_{n+1}.\mathsf{EProp} \subseteq s_{n+1}.\mathsf{EProp}$ and the subset property. Show the subset property first:

$$\forall (e', e) \in s'_{n+1}.\mathsf{Order}. e \in s_{n+1}.\mathsf{EProp}\ tid \Rightarrow e' \in s_{n+1}.\mathsf{EProp}\ tid.$$

As shown above, have:

$$s'_{n+1}.\mathsf{Order} = (s'_n.\mathsf{Order} \cup \{(w,r)\} \cup B \times \{w\} \cup \{r\} \times A)^+$$

By IH it is $s_n \sim s'_n$, so the subset property holds for s_n and s'_n:

$$\forall (e',e) \in s'_n.\mathsf{Order}.e \in s_n.\mathsf{EProp}\ tid \Rightarrow e' \in s_n.\mathsf{EProp}\ tid.$$

It is $s_{n+1}.\mathsf{EProp} = s_n.\mathsf{EProp}$, so also have

$$\forall (e',e) \in s'_n.\mathsf{Order}.e \in s_{n+1}.\mathsf{EProp}\ tid \Rightarrow e' \in s_{n+1}.\mathsf{EProp}\ tid.$$

So to show the subset property holds for s_{n+1} and s'_{n+1} by Lemma 2 only have to show the edges $\{(w,r)\} \cup B \times \{w\} \cup \{r\} \times A$ satisfy the subset property as well:

$$\forall (e',e) \in \{(w,r)\} \cup B \times \{w\} \cup \{r\} \times A.e \in s_{n+1}.\mathsf{EProp}\ tid \Rightarrow e' \in s_{n+1}.\mathsf{EProp}\ tid.$$

Assume $(e',e) \in \{(w,r)\} \cup B \times \{w\} \cup \{r\} \times A$ and $e \in s_{n+1}.\mathsf{EProp}\ tid$ for some tid. Show $e' \in s_{n+1}.\mathsf{EProp}\ tid$. Then there are three cases:

1. $(e',e) \in \{(w,r)\}$. Since by definition of POP's satisfy-read-cand r is propagated to the same threads as w in s_n, so $e' = w \in s_n.\mathsf{EProp}\ tid$.
2. $(e',e) \in B \times \{w\}$ By assumption w is fully-propagated, requiring all Order-before events to be propagated to all threads as well, so $e' \in s_n.\mathsf{EProp}\ tid$.
3. $(e',e) \in \{r\} \times A$. As by definition of POP's satisfy-read-cand w is fully-propagated and r propagated to the same threads $e' = r \in s_n.\mathsf{EProp}\ tid$ is true for all tid.

In all cases $e' \in s_n.\mathsf{EProp}\ tid = s_{n+1}.\mathsf{EProp}\ tid$, so the subset property holds. Now remains showing $s'_{n+1}.\mathsf{EProp} \subseteq s_{n+1}.\mathsf{EProp}$.

$$\begin{aligned}
s'_{n+1}.\mathsf{EProp}\ tid &= s'_n.\mathsf{EProp}\ tid \cup \{r\} \cup \{e | (e,r) \in s'_{n+1}.\mathsf{Order}\} && \text{(def)} \\
&= s_n.\mathsf{EProp}\ tid \cup \{r\} \cup \{e | (e,r) \in s'_{n+1}.\mathsf{Order}\} && \text{(IH)} \\
&= s_n.\mathsf{EProp}\ tid \cup \{e | (e,r) \in s'_{n+1}.\mathsf{Order}\} && (r \text{ prop. to all threads in } s_n) \\
&= s_{n+1}.\mathsf{EProp}\ tid \cup \{e | (e,r) \in s'_{n+1}.\mathsf{Order}\} && \text{(def)} \\
&= s_{n+1}.\mathsf{EProp}\ tid && \text{(below)}
\end{aligned}$$

The last step is correct because, as shown before, $r \in s_{n+1}.\mathsf{EProp}\ tid$ for all tid and the subset property holds for s_{n+1} and s'_{n+1}. So have $s_{n+1} \sim s'_{n+1}$.

So all in all the NOP transitions corresponding to the POP ones preserve the simulation relation $\sim$, showing that every valid finite POP trace from the initial state can be simulated in NOP. □

Appendix C

Proofs of mixed-size SC properties

The following gives the proofs of the statements concerning the properties of fully-barriered mixed-size ARMv8 programs and mixed-size ARMv8 programs in which all loads are load acquires and all stores are store releases, from Chapter 5.[1]
In the following, the term "barrier", if not specified otherwise, will mean a full barrier: a `dmb sy` barrier for ARMv8 and a `sync` barrier for Power. For simplicity, the proofs assume that reads do not read from "initial memory", so, that each read always reads from writes that originate from stores in the input program. Also, for simplicity assume the fully-barriered programs have no barriers other than full barriers.

In the context of describing the dynamic behaviour of POP, following the definition of the reorder condition, the *footprint* of a read request/event in some POP state refers to the unsatisfied slices of the read request in that state. Accordingly, a read request in some POP state is said to *overlap* with another request in the storage subsystem when its footprint overlaps that of the other request in that state. (The footprint of a write request is the full footprint of the write.)

The following assumes aligned memory accesses, and instructions that perform no more than a single memory access (and so assumes no atomic memory operations, no load/store pair or load/store multiple instructions, etc.). For all aligned memory accesses of such instructions, the thread subsystem generates a single memory request (a single read request for loads, a single write request for stores). The following text, therefore, for convenience, often identifies a memory access instruction with its single memory request (read or write request). The following assumes programs with at least two threads (for convenience). Finally, the following assumes finite executions.

Definition 1. Let *tr* be a POP trace with final state *s*. Then the coherence relation co derived from *tr* is given by the transitive closure of the restriction of *s*.Order to overlapping writes. The byte-wise coherence relation given by *tr* is *s*.Order lifted to the byte-sized subevents (leaving barriers unchanged), restricted to same-byte-address subwrites.

The reads-from relation rf determined by *tr* relates a read event *r* of *tr* with a write event *w* of *tr*, together with a footprint *fp*, if *r* read *fp* from *w* in *tr*. The byte-wise reads-from relation contains (sw, sr) if *sr* is a subread that read from the subwrite *sw* in *tr* (so *sw* and *sr* have the same byte-address).

The program order relation po determined by *tr* relates the reads, writes, and any (`dmb sy` or other) barrier events of *tr* in the same order as their originating instruction instances in the threads' instruction trees in the final state *s*. The byte-wise program order relation lifts this relation

[1] The results of this chapter have been published in Flur et al. [52] and the proof text adapts that found in the supplementary material of same paper.

to byte-sized subevents (lifting reads to their subreads, writes to their subwrites and leaving any (`dmb sy` or other) barriers unchanged).

Definition 2. An operational model trace *tr* is an SC execution if there exists a total order *eo* on all read and write events of *tr* that contains the program order relation determined by *tr* (restricted to reads and writes) and coherence relation determined by *tr*, and where the reads-from relation determined by *tr* agrees with *eo*: for each read r and bit address b in the full footprint of r, if r read the bit at bit address b from w in *tr*, the write w precedes r in *eo* and is the maximal predecessor write of r writing to bit address b in *eo*.

C.1 POP preserves singlecopy-atomicity

Lemma 4. *Assume a trace with no restarts or instruction discards. If* $(e', e) \notin s.\mathit{Order}$ *and* e *is fully propagated in* s *and* $s \rightarrow^* s'$*, then* $(e', e) \notin s'.\mathit{Order}$*.*

Proof. Assume $(e', e) \notin s.\mathsf{Order}$ and e fully propagated in s. Show, for any transition t, if $s \xrightarrow{t} s'$, then it is $(e', e) \notin s'.\mathsf{Order}$, by case analysis on t.

- An accept-event transition t cannot add edges (e', e) pointing into e.
- A propagate-event transition t cannot add edges (e', e) pointing into e. Neither e nor any of the events $e*$ with $(e*, e) \in s.\mathsf{Order}$ can be propagated to a new thread, since by assumption e is fully propagated (and therefore also $e*$); and for other events $e*$ with $(e*, e) \notin s.\mathsf{Order}$, propagating $e*$ to some new thread does not add (direct or transitive) edges (e', e) to Order, since by assumption e and all events Order-before e are propagated to all threads.
- Assume a satisfy-read transition t for a write w and a read r adds an edge (e', e) to Order when going from s to s'. Since the only direct edge a satisfy-read transition can add is (r, w) (swapping w and r in Order), it must be a satisfy-read transition for which w is e or an event $e*$ with $(e*, e) \in s.\mathsf{Order}$. Then by assumption w must be fully propagated, and by definition of the satisfy-read transition also r. But then the satisfy-read transition does not flip r and w in Order and does not add the edge (r, w), contradiction.

So in all cases $(e', e) \notin s'.\mathsf{Order}$. Therefore, if e is still in the storage subsystem in s', e still fully propagated in s', since no transition "un-propagates" events. (Else, by assumption of no restarts or discards, e will not return to the storage subsystem.)

Then inductively also $(e', e) \notin s'.\mathsf{Order}$ for $s \rightarrow^* s'$. □

Lemma 5 (SCA)**.** *Let tr be a POP trace, r a read, and w and w′ two writes overlapping each other. Assume r, w, and w′ are aligned and in tr the coherence order is determined as* $w \xrightarrow{\mathsf{co}} w'$*. If part of the final return value of r (the return value after all restarts r might have) was read from w′ in tr, its final return value in tr cannot contain a part read from w that is covered by w′ (is coherence-hidden behind w′).*

Proof. Assume a trace *tr* where the return value of r was partly read from w' but where the return value also contains a part for footprint *fp* that was read from w where the footprint *fp* of w is coherence-hidden by w' (in the final coherence order). Without loss of generality, assume *tr* has

no restarts or instruction discards. Since r is aligned, the thread subsystem generates a single read request for it, since w and w' are aligned, the thread subsystem generates a single write request for each of them. Now there are two cases: r reads from w by forwarding or in the storage subsystem.

Case r reads from w by forwarding.

Then w' cannot be from the same thread as r and w: assume it is, then it needs to be program ordered with w and r; w is ordered before r so it can forward to r; w has to be ordered before w', because by assumption the coherence order is $(w, w') \in \mathsf{co}$; w' cannot be after r, since then r would not be able to read from w', and w' cannot be between w and r (if w' is available for forwarding at the point of forwarding w to r then fp would be read from w', contradicting the assumption; if not, w' would later restart r, contradicting the assumption of no restarts). So w' not from the same thread as w and r. Let *tid* be the thread of r and w. Now there are again two cases: w' propagates to thread *tid* before w propagates to memory or after.

Case w' propagates to *tid* before w propagates to memory.

Then when w propagates to memory w' is already propagated to *tid* and POP adds the edge (w', w) to Order. This edge remains until the end of the trace. Contradiction to the assumption that $(w, w') \in \mathsf{co}$.

Case w propagates to memory before w' propagates to thread *tid*.

Let $s*$ be the state after w enters the storage subsystem. By definition of the thread subsytem w can only be accepted into the storage subsystem after r is issued to the storage subsystem (due to forwarding w to r). So r has already been issued in $s*$; since by assumption w' propagates to *tid* only after w propagates to memory, in $s*$ the read r is not satisfied yet; and, by definition of the re-order condition, POP added the edge $(r, w) \in s*.\mathsf{Order}$. By assumption, r reads from w'; the edge (r, w) remains until r is entirely satisfied, so at least until r reads from w'. Let s be the state before r reads from w', after $s*$. For r to read from w', by definition of the satisfy-read transition it has to be $(w', r) \in s.\mathsf{Order}$. But then transitively also $(w', w) \in s.\mathsf{Order}$, which will remain until the end of tr, contradicting $(w, w') \in \mathsf{co}$.

Case r does not read from w by forwarding but in memory.

r reading from w and w' in the same transition is only possible if r reads from both writes by forwarding, but by assumption r does not read from w by forwarding. So r must either first read from w and then w', or the other way round. If in tr the read r first reads from w', then r reads from w' the biggest possible footprint, which by assumption includes fp, and so r cannot read fp from w, contradiction to the assumption. So assume r first reads from w. (This also means r does not read from w' by forwarding.)

Let s be the state before r reads from w and s' the state afterwards. Then by definition of the satisfy-read transition there is an edge $(w, r) \in s.\mathsf{Order}$. Now there are two cases: either w fully propagated in s or not.

Case w fully propagated.

Then by definition of POP r also fully propagated. Since by assumption r also reads

from w' in tr, in some state after s there must be an Order-edge (w', r). Then by Lemma 4 it must be $(w', r) \in s.\mathsf{Order}$, and so w' also propagated to all threads. w and w' must be Order-related. Then the edge must be $(w, w') \in s.\mathsf{Order}$. (Assume it is not, then $(w', w) \in s.\mathsf{Order}$, which will remain until the end of tr, contradicting the assumption $(w, w') \in \mathsf{co}$.) By assumption, w and w' cover the footprint fp, and r reads fp from w. Therefore in s the (unsatisfied footprint of the) read r overlaps w and w'. But since r, w, and w' are propagated to all threads, w' is a write overlapping (the unsatisfied footprint of) r Order-between r and w that is propagated to a common thread with r. Contradiction to the assumption that r can read from w in state s.

Case w not fully propagated.

Then r is not fully propagated either.

Now if it is $(w', r) \in s.\mathsf{Order}$, then — since r not fully propagated in s — the satisfy-read transition flips the order of r and w in Order and it is $\{(w', r), (r, w)\} \subseteq s'.\mathsf{Order}$ (the edge (w', r) remains, since by assumption r will still read from w', and so must still overlap w' after reading from w), contradicting the coherence order (w, w'). So it is $(w', r) \notin s.\mathsf{Order}$. It cannot be $(r, w') \in s.\mathsf{Order}$, since in that case r would not be able to later read from w'. So r is not related to w' in $s.\mathsf{Order}$.

Since w and r are not fully propagated the satisfy-read transition flips the order of r and w and it is $(r, w) \in s'.\mathsf{Order}$. Since by assumption there are no restarts or instruction discards, no transition can delete the edge $(r, w) \in \mathsf{Order}$ until r is satisfied. Let s'' be the state, after or equal to s', before r reads from w'. Then by definition of the satisfy-read transition it is $(w', r) \in s''.\mathsf{Order}$. Also have $(r, w) \in s''.\mathsf{Order}$ and therefore transitively $(w', w) \in s''.\mathsf{Order}$. Since w and w' overlap, no transition can delete this edge, and so in any following state it is $(w', w) \in \mathsf{Order}$, so that in the final state the coherence order is determined to $w' \xrightarrow{\mathsf{co}} w$, contradicting the assumption.

□

C.2 Release/Acquire restore SC

Lemma 6. *Two write releases that are propagated to at least one common thread are Order-related with each other.*

Proof. Let w and w' be two such write releases. Let tid' be the first thread they were both propagated to. Without loss of generality assume w propagated to tid' before w'. Let tid_w be the thread of w and $tid_{w'}$ the thread of w'.

Now assume the state s in which w was already propagated to tid' and just before w' was propagated to tid'. Now there are two cases: the transition that propagated w' to tid' was an accept-request transition or a propagate transition.

***w'* propagated to *tid'* with an accept-request transition** Then $tid' = tid_{w'}$ and when w' was accepted into the storage subsystem w was already propagated to $tid_{w'}$ and POP added the edge (w, w') to Order when accepting w', because w and w' do not satisfy the reorder

condition. This edge cannot be deleted by any transition (or restart or discard) and remains in Order in all subsequent states.

***w′* propagated to *tid′* with a propagate transition** Then w' was already in the storage subsystem in s, and w not propagated to $tid_{w'}$ (otherwise tid' would not be the first thread that both w and w' were propagated to). If w and w' were already s.Order-related, then the statement follows, because no transition (or restart or discard) can delete this edge.
Else, when in the next transition w' propagated to tid' POP added the edge (w', w) to Order, since w' and w do not satisfy the reorder condition, since they are not already related, and since w is propagated to tid' but not yet propagated to $tid_{w'}$. This edge cannot be deleted by any transition (or restart or discard) and will remain in all subsequent states.

□

Lemma 7. *A write release that is propagated to all threads is Order-related with all other write releases in the storage subsystem.*

Proof. Let w be a write release propagated to all threads and w' another write release in the storage subsystem. Then w and w' are propagated to a common thread and by Lemma 6 they are Order-related. □

Corollary 1. *In a final POP state s all release writes propagated to the storage subsystem in the execution are totally ordered by s.Order.*

Proof. As s is a final state, all these release writes are propagated to all threads. By Lemma 7 any two release writes are now Order-related. Since Order is acyclic it totally orders the release writes. □

Lemma 8. *A program in which all loads are acquire loads and all stores are release stores does not have restarts or thread-internal forwarding in any execution.*

Proof. Restarts are caused for two reasons: reads being issued out of order with respect to program-order-earlier reads or writes, and thread-internal forwarding of writes to reads. Since acquire reads can only issue if all previous acquire loads have already been entirely satisfied and previous release stores are finished, programs with only acquire loads and release stores do not have out of order issuing of reads. Since read acquires can only satisfy by forwarding when all previous write releases are finished, a program with only acquire loads and release stores does not have thread-internal forwarding. Therefore there are no restarts and there is no forwarding. □

Lemma 9. *Let (e, e') in s.Order and s' such that $s \xrightarrow{t} s'$ for some transition t. If e and e' are release writes, then (e, e') in s'.Order. If not both e and e' are release writes, the edge (e, e') is in s'.Order unless t is a satisfy-read transition or there is an instruction restart or discard.*

Proof. By case analysis on the transition types. □

Lemma 10. *Let tr be a POP trace without instruction restarts or discards, let s be a POP state in tr, and r a read that is fully propagated in s. Let ws be the writes that r will still read from in tr after s. Then all writes $w \in ws$ are propagated to the storage subsystem in s and ordered $(w, r) \in s.\mathsf{Order}$.*

Proof. Let w be one such write in ws. In order for r to read from w in some state s', by definition of the satisfy-read transition (w, r) must be in $s'.\mathsf{Order}$. If r is fully propagated in s before s', by Lemma 4 (w, r) must already be in $s.\mathsf{Order}$, and thus w must be propagated to the storage subsystem in s. □

Lemma 11. *Assume a program in which all loads are load acquires and stores store releases. Then in any POP execution for this program, all reads and writes of the execution are accepted into the storage subsystem in program order.*

Proof. By Lemma 8 there are no restarts. According to the thread semantics a read acquire can only issue or satisfy by forwarding when all po-earlier acquire loads have been entirely satisfied, and all po-earlier release stores are finished and their writes accepted into the storage subsystem. Since there is no thread-internal forwarding each read acquire of the execution therefore enters the storage subsystem, and only after its program-order-preceding reads and writes. Any write release of the execution can only commit and propagate to memory once all po-earlier memory access instructions are finished (so writes are accepted to the storage subsystem, (acquire) reads have been issued and satisfied). □

Theorem 9. *An ARM program in which all loads are acquire loads and all stores are release stores, and whose memory accesses are all aligned, has sequentially consistent behaviour.*

Proof. For simplicity assume the ARM program has no barriers (no `dmb sy` or other). Let $tr = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$ be a POP trace for a program in which all loads are acquire loads and stores are release stores. By Lemma 8 this trace has no restarts. Moreover, without loss of generality, assume the trace has no instruction discards (does not explore instruction tree branches that are the result of incorrect branch speculation). Let $s := s_n$ be the final state. Let E be the set of all read and write events/requests from tr. Now define an initial event order eo as follows:

$$eo := s.\mathsf{Order} \downharpoonright \{e | e \text{ is write}\}$$

Lemma 12. *eo totally orders all write events of tr, and if $w \xrightarrow{\mathsf{po}} w'$, it is $(w, w') \in eo$.*

Proof. Since by Corollary 1 $s.\mathsf{Order}$ totally orders all write events, so does eo. Let $w \xrightarrow{\mathsf{po}} w'$ be two write releases from the same thread. Then (w, w') is in eo: when w' is accepted into the storage subsystem w must have already been accepted into the storage subsystem as well, by Lemma 11. When w' is accepted, the edge (w, w') is added to Order. By Lemma 9 (w, w') is also contained in $s.\mathsf{Order}$. □

Now define eo by algorithm embed (refining the initial eo), where pos is the list of per-thread program orders, each in list form, (a list of lists of events, where each list of events contains the

events of one thread in order matching po derived from *tr*), in some arbitrary thread order. Here rf is the reads-from relation determined by *tr*, ignoring the footprint information.

```
embed' po_t =
  for i in [0 .. length po_t − 1] {
    if po_t[i] is read event {
      let r = po_t[i] in
      eo := eo ∪{(w, r)|w ∈ E, (w, r) ∈ rf}     //A
      eo := eo ∪{(e, r)|e ∈ E, (e, r) ∈ po}     //B
      eo := eo ∪{(r, w)|w ∈ E, w is write, (r, w) ∈ po, (w, r) ∉ eo^+}        //C
      eo := eo ∪{(r, w)|w ∈ E, w is write, w overlaps the full footprint of r, (w, r) ∉ eo^+}        //D
      eo := eo^+
    }
  }

embed = map embed' pos
```

Since POP's Order relation is acyclic in any state and since *eo* is initially set to *s*.Order restricted to writes, the initial *eo* is acyclic. Prove that executing loop i of embed' po_t for any of the threads' program orders po_t during any stage of the execution of embed preserves acyclicity of *eo*. There are two cases: $\mathsf{po}_t[i]$ is a write event or $\mathsf{po}_t[i]$ is a read event. If $\mathsf{po}_t[i]$ is a write event, then since the loop does not add to *eo* this is immediately true. So assume $r = \mathsf{po}_t[i]$ is a read event. Running the loop for i adds r into *eo*. The commands A and B only add edges pointing into r. C only adds edges (r, e) if (e, r) is not already in eo^+ after running A and B. D only adds edges (r, e) if (e, r) is not already in eo^+ after A, B, and C. Therefore *eo* remains acyclic by construction.

Now prove that before executing the loop for index i during the run of embed' for $0 \leq i \leq$ length po_t (taking the run for $i =$ length po_t to be the loop termination) on any of the thread's program orders po_t (not necessarily the first po_t in *pos*, so during any stage of the execution of embed) the following holds (INV): Let r be a read from $\mathsf{po}_t[0..i-1]$. Then:

1. *eo* agrees with program order:
   (a) if (e, r) in po for some write or read event e then (e, r) in *eo*.
   (b) if (r, w) in po for a write w, then (r, w) in *eo*.
2. For any write w in E that overlaps the full footprint of r: either (w, r) or (r, w) in *eo*.
3. Let b be a bit address in the full footprint of r. Let w_b be the write from which r read the bit at bit address b in *tr*. Then $(w_b, r) \in eo$ and w_b is the maximal write preceding r in *eo* that writes to bit address b. (The maximum is well-defined, since all writes of E are totally ordered in *eo*.)

Once the statement above is proved, from this follows that after running embed, *eo* is a partial order of all read and write events from *tr* that contains program order and coherence, and in which each read's *eo*-prefix determines the writes (and footprints) read from in the same way as the rf relation determined by *tr*. Any linear extension of the partial order *eo* on E (linearise all events of E in a way that is compatible with *eo*) is a total order that contains program order (since in *eo* all same-thread events are ordered in program order), and co, and agrees with rf of

the POP trace (since all writes are totally ordered and since in *eo* any read is already ordered with all overlapping writes (overlapping its full footprint) in a way that is compatible with rf). From this follows that *tr* is a valid SC execution.

So have to show INV, by induction on i.

Base case: $i = 0$. 1.–3. are vacuously true.

Step case: $i \to i+1$. Assume INV holds for i, show it also holds for $i+1$. Therefore, show that the execution of loop i preserves INV. If $i+1 = \text{length } \mathsf{po}_t + 1$ then nothing to show. So assume $i < \text{length } \mathsf{po}_t$. If $\mathsf{po}_t[i]$ is a write event, the reads from $\mathsf{po}_t[0..i+1-1]$ are the same as from $\mathsf{po}_t[0..i-1]$, and because *eo* does not change, 1.–3. hold by induction hypothesis. So assume $r = \mathsf{po}_t[i]$ is a read event.

1. (a) By induction hypothesis this is true for all reads in $\mathsf{po}_t[0..i-1]$, only need to show it is also true for $r = \mathsf{po}_t[i]$. Then for r these edges are added by command B.

   (b) By induction hypothesis this is true for all reads in $\mathsf{po}_t[0..i-1]$, only need to show it is also true for $r = \mathsf{po}_t[i]$. Let w be a write such that $r \xrightarrow{\mathsf{po}} w$. Show that (r, w) is added to *eo*. To do that, show that before running C in the loop for i, (w, r) is not in eo^+ and therefore C adds (r, w). Assume (w, r) is in eo^+ before C in loop i.

   Now there are two cases: (w, r) is a transitive edge or not.

   Case (w, r) is a transitive edge.

   Then look at the immediate predecessor e of r on one path from w to r in the transitive reduction of *eo*. So there is $(w, e), (e, r)$, in eo^+ where (e, r) cannot be a transitive edge and $e \neq r$. Now there are two cases: (e, r) was added because e is po-before r or because e is a write that r read from.

   Before running C in the loop i, embed' only added edges pointing into r, so (w, e) must have also already been in *eo* before running loop i.

   Case (e, r) was added because e is po-before r.

   But then $e \xrightarrow{\mathsf{po}} r \xrightarrow{\mathsf{po}} w$, so (e, w) must have been in *eo* before the execution of the loop for i (if e is a write this is by Lemma 12, if it is a read then by induction hypothesis). But then before running loop i there was a cycle $(w, e), (e, w)$ in *eo*. Contradiction: *eo* is acyclic.

   Case (e, r) was added because e is a write that r read from.

   Let s' be the state before r reads from e. Since r cannot read by forwarding this is when e and r are in the storage subsystem. By definition of the satisfy-read transition r and e are both fully propagated in s' and there exists an edge $(e, r) \in s'.\mathsf{Order}$. Now in s' either w is propagated to the storage subsystem or not. If not, then once it propagates to the storage subsystem POP adds the edge (e, w) to Order, since e is propagated to w's thread. But then by Lemma 9 it is also $(e, w) \in s.\mathsf{Order}$ and therefore in (the initial and all later) *eo*, contradiction to the acyclicity of (the final) *eo*, since by assumption have (w, e) in *eo* (before loop i and in all later *eo*).[2]

[2] embed' never deletes edges from *eo*, so once some edge is included in *eo* at some stage of the execution of embed, it remains until the end. When the proof here (and in other places) states a contradiction to the acyclicity of *eo*

So assume in s' the write w is already propagated into the storage subsystem. But this cannot happen: w can only commit and propagate when all program-order-earlier load acquires are finished, so only after s', contradiction.

Case (w, r) is a direct edge.

Then by assumption that $r \xrightarrow{\mathsf{po}} w$, (w, r) must have been added because r read from w. r cannot read from w by forwarding since $r \xrightarrow{\mathsf{po}} w$. And by definition of the thread subsystem w can only propagate into the storage subsystem after r is finished. Contradiction, r cannot read from w.

2. By induction hypothesis this is already true for all reads in $\mathsf{po}_t[0..i-1]$, and since embed' only adds edges to *eo* (does not delete edges) loop i preserves this. Only need to show this for $r = \mathsf{po}_t[i]$. Commands A and B of loop i add certain edges pointing into r, command C some going out of r. For any write event w in E overlapping the full footprint of r that is not directly or transitively related $(w, r) \in eo^+$ after that, command D adds an edge (r, w) to *eo*.
3. By induction hypothesis for all reads in $\mathsf{po}_t[0..i-1]$ this is true before running loop i. Have to show (a) that loop i preserves it for $\mathsf{po}_t[0..i-1]$ and (b) that loop i establishes it for $r = \mathsf{po}_t[i]$.
   (a) Let r' be a read from $\mathsf{po}_t[0..i-1]$. By induction hypothesis, for each bit address b in the footprint of r' the write w_b from which r' read the bit at bit address b in tr is *eo*-before r' and is the closest *eo*-predecessor write of r' writing to bit address b in *eo* before executing loop i. Since by induction hypothesis (2.) r' is *eo*-ordered with all writes in E overlapping the full footprint of r' before running loop i, since *eo* totally orders all writes of E, and since embed' preserves acyclicity of *eo*, the *eo*-prefix of r' of overlapping writes is the same before and after the execution of loop i and (a) follows.
   (b) Have to show that loop i establishes this for $r = \mathsf{po}_t[i]$. So let b be a bit address in the footprint of r and w_b the write from which r read the bit at bit address b. Then command A in loop i adds w_b into the *eo*-prefix of r. Have to show after loop i there is no write w' *eo*-between w_b and r that also writes to bit address b. Assume after executing loop i there is such a write. Let w' be the *eo*-maximal such write with $\{(w_b, w'), (w', r)\} \subseteq eo$ writing to bit address b. (This is well-defined since *eo* totally orders writes.) Since $s.\mathsf{Order}$ totally orders writes and since *eo* acyclic, it must then also be $(w_b, w') \in s.\mathsf{Order}$.

   Let s' be the state when r is fully propagated for the first time. Show it is $(w', r) \in s'.\mathsf{Order}$, which leads to a contradiction: assume $(w', r) \in s'.\mathsf{Order}$. Then in s' the write w' is fully propagated, and therefore by Lemma 4 it must be $(w_b, w') \in s'.\mathsf{Order}$ and w_b also fully propagated. Assume the state s'' before r reads from w_b. Since tr involves no write forwarding this is s' or after s'. Since by assumption r reads the bit at bit address b from w_b, in s'' the (unsatisfied) footprint of read r overlaps w' (the part for bit address b is not satisfied until now), and so it is still $\{(w_b, w'), (w', r)\} \subseteq s''.\mathsf{Order}$.

---

without stating "when", this refers to the acyclicity of the final *eo*. Similarly, when the proof shows the existence of certain *eo* edges, this usually refers the existence in the final *eo*.

(r cannot read from w' before reading from w_b since then it would read the bit at bit address b from w'. Since r is fully propagated the storage subsystem does not swap any edges involving r.) But then in s'' the write w' overlapping the unsatisfied footprint of r is Order-between w_b and r, and propagated to a common thread with r, and r cannot read from w_b, contradiction.

Remains to show $(w', r) \in s'.\mathsf{Order}$. Now there are different cases for why embed' added the edge (w', r):

Case r read from w'.

r cannot read by forwarding. By definition of POP r can only read once it is fully propagated, and so in s' the read r has not read from w' yet. Then by Lemma 10 it is $(w', r) \in s'.\mathsf{Order}$.

Case $w' \xrightarrow{\mathsf{po}} r$.

By Lemma 11 r can only enter the storage subsystem after w' is propagated into the storage subsystem. In the transition when r issues into the storage subsystem POP adds the edge (w', r) to Order that remains at least until the state when r is fully propagated for the first time. So $(w', r) \in s'.\mathsf{Order}$.

Case (w', r) is a transitive edge.

Let p be a path from w' to r in the transitive reduction of eo, and w'' the last write on p. Now either $w' = w''$ or not.

Case $w' = w''$. By choice of w'' there are no writes between $w' = w''$ and r on p. Since (w', r) is a transitive edge there must be at least one other event between w' and r, let e be the first one. e must be a read, and the edge (w', e) a direct edge. Then it is $(w', e), (e, r)$ in eo, and by construction of eo it must be $e \xrightarrow{\mathsf{po}} r$. If $w' \xrightarrow{\mathsf{po}} e$, then $w' \xrightarrow{\mathsf{po}} r$, which is dealt with by the previous case. So assume it is not $w' \xrightarrow{\mathsf{po}} e$. Then it must be that e reads from w'.

r can only issue after e is entirely satisfied. Let $s*$ be the state before r is issued into the storage subsystem, before s'.

Then e has already read from w' in $s*$ and w' is fully propagated. Since w' has propagated to r's thread in $s*$, when r issues POP adds the edge (w', r) to Order, since w' and r overlap. Since r cannot be partially or entirely satisfied until it is fully propagated, this remains until s'.

Case $w' \neq w''$. Then $(w', w'') \in eo$ and therefore $(w', w'') \in s.\mathsf{Order}$. Again, by choice of w'' there are no writes between w'' and r on p. Let e be the first read on the path from w'' to r on p. Now there are four cases:

Case $e = r$ and e read from w''. $e = r$ cannot read by forwarding. By definition of POP r can only read once it is fully propagated, and so in s' the read r has not read from w'' yet. Then by Lemma 10 it is $(w'', r) \in s'.\mathsf{Order}$, and so w'' is fully propagated. Since $(w', w'') \in s.\mathsf{Order}$ by Lemma 4 it must be $(w', w'') \in s'.\mathsf{Order}$ and therefore transitively $(w', r) \in s'.\mathsf{Order}$.

Case $e = r$ and $w'' \xrightarrow{\mathsf{po}} e$. By Lemma 11 r can only enter the storage sub-

system after w'' is propagated into the storage subsystem. In the transition when r issues into the storage subsystem POP adds the edge (w'', r) to Order that remains while r is in the storage subsystem. So $(w'', r) \in s'.\mathsf{Order}$, and w'' fully propagated in s'. Then, as before, since $(w', w'') \in s.\mathsf{Order}$ by Lemma 4 also $(w', w'') \in s'.\mathsf{Order}$ and transitively $(w', r) \in s'.\mathsf{Order}$.

Case $e \xrightarrow{\mathsf{po}} r$ and e read from w''. r can only issue once e is entirely satisfied. Let $s*$ be the state before r is issued into the storage subsystem, before s'. Then e has already read from w'' in $s*$ and w'' is fully propagated. Since it will be $(w', w'') \in s.\mathsf{Order}$, by Lemma 4 it must be $(w', w'') \in s*.\mathsf{Order}$, and w' must be fully propagated in $s*$. Since w' has propagated to r's thread in $s*$, when r issues POP adds the edge (w', r) to Order, since w' and r overlap. Since r cannot be (partially or entirely) satisfied until it is fully propagated this remains until s'.

Case $e \xrightarrow{\mathsf{po}} r$ and $w'' \xrightarrow{\mathsf{po}} e$. Then $w'' \xrightarrow{\mathsf{po}} r$, and the proof is similar to before: By Lemma 11 r can only enter the storage subsystem after w'' is propagated into the storage subsystem. In the transition when r issues into the storage subsystem POP adds the edge (w'', r) to Order that remains while r is in the storage subsystem. So $(w'', r) \in s'.\mathsf{Order}$, and w'' fully propagated in s'. Then, as before: since $(w', w'') \in s.\mathsf{Order}$, by Lemma 4 also have $(w', w'') \in s'.\mathsf{Order}$, and transitively $(w', r) \in s'.\mathsf{Order}$.

□

C.3 Barriers restore BSC+SCA

C.3.1 ARM

By Lemma 5 the SCA part is given in POP even without any (`dmb sy` or other) barriers. So need to show that fully-barriered ARM programs without misaligned accesses are BSC.

First prove the following lemma.

Lemma 13. *A fully-barriered ARM program does not have restarts or thread-internal forwarding.*

Proof. Restarts are caused for two reasons: reads being issued out of order with respect to program-order-earlier reads or writes, and thread-internal forwarding of writes to reads. Since reads can only be issued if all previous barriers are finished, and therefore all po-earlier loads are finished and their reads satisfied and all po-earlier stores are finished and their writes propagated to memory, fully-barriered ARM programs do not issue reads out of order. Thread-local write forwarding can only happen if, when a read is available for being satisfied, the po-earlier writes to forward from are not propagated yet. But since any read can only read by forwarding once po-earlier barriers are finished, and therefore po-earlier writes are propagated, there is no thread-local write forwarding in fully-barriered programs. Therefore there are no restarts. □

So the following assumes traces with no restarts and no thread-internal forwarding. Moreover,

without loss of generality, assume (as before) traces with no instruction discards.

Theorem 10. *The behaviour of fully-barriered ARM programs with no misaligned memory accesses is BSC+SCA.*

Proof. Now use a similar approach as in the Release/Acquire SC proof to construct a total order on the *byte-sized* subevents that corresponds to byte-wise po, rf, and co. Let $tr = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$ be a POP trace without restarts or discards with final state $s := s_n$ for a fully-barriered ARM program without misaligned accesses. Assume for simplicity that the program does not have barriers other than `dmb sy` barriers, and no release/acquire instructions. Split up events from *tr* into byte-sized subevents (leaving barriers unchanged), and let E be the set of all subreads and subwrites of *tr*. Let po be the byte-wise program order, and rf and co the byte-wise reads-from and coherence relations determined by *tr*. Now define the initial *eo*:

$$eo := s.\text{Order restricted to writes, lifted to byte-sized subevents}$$

Now *eo* is an acyclic relation on all subwrites of the trace. Furthermore, if $sw \xrightarrow{\text{po}} sw'$ for a subwrite sw of w and subwrite sw' of w', then (sw, sw') in *eo*.

Proof. Acyclicity of *eo* follows from the acyclicity of s.Order. As $w \xrightarrow{\text{po}} w'$ there is at least one barrier between w and w', let b be the last one. Committing and propagating w' requires b to be finished, committing and finishing b requires w to be finished and therefore propagated. When committing b, an edge (w, b) is added to Order that no transition can remove. Thus the edge is still in Order when w' is propagated to memory, at which point (b, w') is added to Order, and by transitivity $(w, w') \in$ Order. As no transition can delete (b, w') from Order, it is (w, w') in s.Order and therefore (sw, sw') in *eo*. □

Furthermore, if (sw, sw') in co then (sw, sw') in *eo*.

Proof. Let w be the write of sw, w' the write of sw'. Since $sw \xrightarrow{\text{co}} sw'$, (w, w') in s.Order. Therefore (sw, sw') in *eo*. □

Now define *eo* with the algorithm embed (refining the initial *eo*), where *pos* is the list of non-byte-wise per-thread program orders, each in list form (a list of lists of events, not subevents, where each list of events contains the events of a single thread in order matching the non-byte-wise po derived from *tr*), in some arbitrary fixed thread order.

```
embed' po_t =
  po_t := remove any (dmb and other) barriers from po_t;
  for i in [0 .. length po_t − 1] {
    if po_t[i] is read event {
      let r = po_t[i] in
      for (sr subread of r) {eo := eo ∪{(sw, sr)|sw ∈ E, (sw, sr) ∈ rf}}      // A
      for (sr subread of r) {eo := eo ∪{(se, sr)|se ∈ E, se po-before sr}}     // B
      for (sr subread of r) {eo := eo ∪{(sr, sw)|sw ∈ E, sw is subwrite,
```

sr po-before $sw, (sw, sr) \notin eo^+\}\}$ // C
for (sr subread of r) {eo := eo $\cup\{(sr, sw) | sw \in E, sw$ is subwrite,
sr and sw to same byte-address, $(sw, sr) \notin eo^+\}\}$ // D
eo := eo^+;
}
}

embed = map embed' *pos*

Now prove that executing the loop of embed' po_t for any po_t (not necessarily the first one) for index i during any stage of the execution of embed preserves the acyclicity of eo. There are two cases: $\mathsf{po}_t[i]$ is a write event or $\mathsf{po}_t[i]$ is a read event. If $\mathsf{po}_t[i]$ is a write event, then since the loop does not change eo, this holds. So assume $r = \mathsf{po}_t[i]$ is a read event. Running the loop body for i adds the subreads of r into eo. The commands A and B only add edges pointing into subreads of r, not going out of the subreads. For every subread sr C and D add edges (sr, sw) going out of sr only if (sw, sr) is not already in eo's transitive closure. Therefore loop i preserves the acyclicity of eo. Then after running embed the final eo is acyclic.

Lemma 14. *Let w be a write po-before e in a fully-barriered ARM program. Whenever e is in the storage subsystem of a state s there is an edge (w, b) and (b, e) in s.Order for every barrier b program-order-between w and e.*

Proof. Let b be such a barrier between w and e. When e is in the storage subsystem b must already be finished and therefore committed, which in turn requires w to be finished and therefore propagated. When b is committed (w, b) is added into Order, which no transition can remove; when e is accepted into the storage subsystem (b, e) is added into Order, which can only be deleted by deleting e from the storage subsystem. □

Lemma 15. *Let sw be subwrite of a write w (from tr) and sw' subwrite of a write w' (from tr) and assume a path from sw to sw' in the transitive reduction of eo that includes no other subwrites but at least one subread. Then (w, w') in s.Order.*

Proof. By case analysis on the shapes of the path. The only direct edges that connect subwrites to subreads are the ones added by commands A and B; the only direct edges between subreads are the edges added by command B; the only direct edges that connect subreads with subwrites are the edges from C and D. By type, the possible shapes of the path from sw to sw' are therefore of the shapes (identifying sets of direct edges with the commands that added them — e.g. B standing for the edges added by B —, and where ';' is sequential composition):

1. $B; B^*; C$
2. $B; B^*; D$
3. $A; B^*; C$
4. $A; B^*; D$

Show that for all of the shapes $(w, w') \in s$.Order.

1. Then it is $w \xrightarrow{\mathsf{po}} w'$ and therefore by Lemma 14 it is $(w, w') \in s$.Order.

2. Let sr, subread of some read r, be the last subread on the path. Then it is (sw, sr) in eo^+ and $w \xrightarrow{\text{po}} r$. Let b be the last barrier program-order-between w and r. Since (sr, sw') is a direct edge added by D, sr and sw' are to the same address.
   Let s' be the state before sr is satisfied from some subwrite $sw*$ of a write $w*$. So it must be $(sw*, sr) \in eo$. Since (sr, sw') in eo, A did not add (sw', sr) to eo, and so sr does not read from sw' and therefore $sw' \neq sw*$ and $w* \neq w'$. Have $(sw*, sw') \in eo^+$. Since sw' and $sw*$ are to the same byte-address they must be coherence-related, and since eo contains co and is acyclic it must be $sw* \xrightarrow{\text{co}} sw'$.
   Since there is no write forwarding, r reads in memory. By definition of the satisfy-read transition there is an edge $(w*, r)$ in $s'.\mathsf{Order}$ and there is no write overlapping sr $s'.\mathsf{Order}$-between $w*$ and r that is propagated to a common thread with r, and $w*$ and r are propagated to the same threads.
   In s' the write w and the barrier b must be committed to the storage subsystem and it is $\{(w, b), (b, r)\} \subseteq s'.\mathsf{Order}$. Then $w*$ must be ordered with b.
   If it is ordered $(b, w*) \in s'.\mathsf{Order}$, then have $(w, w*) \in s'.\mathsf{Order}$ and also $(w, w*) \in s.\mathsf{Order}$. By coherence also have $(w*, w') \in s.\mathsf{Order}$, and so transitively $(w, w') \in s.\mathsf{Order}$, as required.
   So assume $w*$ is ordered $(w*, b) \in s'.\mathsf{Order}$. Then since b is $s'.\mathsf{Order}$-between $w*$ and r and propagated to a common thread with r, the read r must be fully propagated in s', by definition of the satisfy-read transition. Then $w*$ and b are fully propagated.
   If w' is not in the storage subsystem yet in s', once it commits into the storage subsystem it will be ordered (b, w') since b is propagated to all threads. This edge remains until s, and so transitively then also $(w, w') \in s.\mathsf{Order}$.
   If w' is already in the storage subsystem in s', since b and $w*$ are propagated to all threads, w' must be ordered with them. By coherence it must be $(w*, w') \in s'.\mathsf{Order}$. It cannot be $(w', b) \in s'.\mathsf{Order}$: then by full propagation of r the write w' would be fully propagated; then w' would be a write overlapping the unsatisfied footprint of r ordered between $w*$ and r and propagated to a common thread with r, and r would not be allowed to read from $w*$ in s' by definition of the satisfy-read transition. So it is $(b, w') \in s'.\mathsf{Order}$. Then $\{(w, b), (b, w')\} \subseteq s'.\mathsf{Order}$ and $\{(w, b), (b, w')\} \subseteq s.\mathsf{Order}$. So $(w, w') \in s.\mathsf{Order}$.
3. Let sr, subread of some read r, be the first subread on the path. So sr reads from sw and it is $r \xrightarrow{\text{po}} w'$.
   Let s' be the state before sr is satisfied. Since there is no thread-internal forwarding sr is satisfied in memory. Then by definition of the satisfy-read transition w and r are propagated to the same threads, including r's thread, and there is an edge $(w, r) \in s'.\mathsf{Order}$. r and w' are separated by a barrier in program order, let b be the last one in program order. For b to commit r must be finished and therefore satisfied. For w' to propagate to memory b must be finished and therefore committed. So in s' neither b is committed or finished, nor w' is propagated to the storage subsystem, and w is propagated to r's thread.
   When b commits to the storage subsystem POP adds the edge (w, b) to Order, since w is propagated to b's thread. When w' propagates to the storage subsystem, after that, POP adds the edge (b, w') to Order. Both edges cannot be deleted by any transitions and so it is

$\{(w,b),(b,w')\} \subseteq s.\mathsf{Order}$ and therefore $(w,w') \in s.\mathsf{Order}$.

4. Let sr, subread of some read r, be the first subread on the path, and sr', subread of some read r' be the last subread on the path.
   If $sr = sr'$ then sw and sw' are to the same byte address and w and w' must be ordered by coherence in $s.\mathsf{Order}$. Since co included in eo and since eo acyclic, from $(sw,sw') \in eo$ follows $(w,w') \in s.\mathsf{Order}$.
   So assume $sr \neq sr'$. Then sr reads from sw, $sr \xrightarrow{\mathsf{po}} sr'$ and (sr',sw') added by command D.
   Let s' be the state before sr is satisfied. Then r' is not in the storage subsystem yet (due to the barriers between sr and sr' waiting for sr to finish), and by definition of the satisfy-read transition w is propagated to the thread of r. Let b be the last barrier between r and r'. When b is accepted into the storage subsystem (after s') POP adds (w,b) to Order, which cannot be deleted by any transition; when r' is accepted into the storage subsystem POP adds (b,r') to Order, which — in the absence of restarts or discards — cannot be deleted before entirely satisfying r'.
   Now let s'' be the state before sr' reads from some subwrite $sw*$ of a write $w*$. (Note that it is not necessarily $w \neq w*$.) As (sr',sw') is added by D, sr' and sw' are to the same address and $sw' \neq sw*$ and therefore $w' \neq w*$. Also, since sr' reads from $sw*$ it is $(sw*,sr') \in eo$ and therefore $(sw*,sw') \in eo^+$. Since $sw*$ and sw' are to the same byte-address they must be coherence-related. Since coherence is a subset of eo, this means it must be $sw* \xrightarrow{\mathsf{co}} sw'$.
   Now in the state s'' there is (w,b) and (b,r') in Order for barrier b. By definition of the satisfy-read transition there is an edge $(w*,r')$ in $s''.\mathsf{Order}$.
   Since $w*$ by definition of POP is propagated to the thread of r' and b, $w*$ and b must be $s''.\mathsf{Order}$-related. If it is $(b,w*) \in s''.\mathsf{Order}$ then also $(b,w*) \in s.\mathsf{Order}$ and therefore $(w,w*) \in s.\mathsf{Order}$. Combined with coherence $(w*,w') \in s.\mathsf{Order}$ have $(w,w') \in s.\mathsf{Order}$.
   So assume it is $(w*,b) \in s''.\mathsf{Order}$. Then, since b is Order-between $w*$ and r' and propagated to a common thread with r', the read r' must be fully propagated, and therefore also b, $w*$, and w.
   If w' is not in the storage subsystem yet, then when it is accepted into the storage subsystem, the edge (b,w') is added to Order and therefore $\{(w,b),(b,w')\} \in s.\mathsf{Order}$ and transitively $(w,w') \in s.\mathsf{Order}$.
   If w' is already in the storage subsystem it must be ordered in coherence order $(w*,w') \in s''.\mathsf{Order}$. w' must also be ordered with b. And it cannot be (w',b): since otherwise have $\{(w*,w'),(w',r')\} \subseteq s''.\mathsf{Order}$ and the write w' overlapping the unsatisfied footprint of r' is Order-between r' and $w*$ and propagated to a common thread with r' (r' is propagated to all threads), and r' would not be allowed to read from $w*$ in s''. So it must be $(b,w') \in s''.\mathsf{Order}$.
   Then $\{(w,b),(b,w')\} \subseteq s.\mathsf{Order}$ and transitively $(w,w') \in s.\mathsf{Order}$.

□

Corollary 2. *For any (non-empty) path from a subwrite sw of a write w (from tr) to a subwrite sw′ of some w′ (from tr) in the transitive reduction of eo it must be* (w,w') *in* $s.\mathsf{Order}$.

Proof. Divide the path into subpaths that start from a subwrite and end in a subwrite passing

only subreads: whenever there is a subpath $sw* \to sw*'$ for a subwrite $sw*$ of $w*$ and $sw*'$ of $w*'$ with no subreads in-between it must have already been $(w*, w*')$ in $s.\mathsf{Order}$, since embed' does not add direct edges between subwrites. And whenever there is a path from one subwrite $sw*$ of $w*$ to a subwrite $sw*'$ of $w*'$ with only subreads (but at least one subread) in-between, according to the previous Lemma it is also $(w*, w*')$ in $s.\mathsf{Order}$. □

Now prove INV. Before executing loop i of embed' on any of the threads' program orders po_t (not necessarily the first thread's po_t, so during any stage of the execution of embed) the following holds for $0 \leq i \leq \text{length } \mathsf{po}_t$ (taking the run for $i = \text{length } \mathsf{po}_t$ to be the loop termination): let r be a read from $\mathsf{po}_t[0..i-1]$ and sr a subread of r. Then

1. eo agrees with program order:
   (a) if (se, sr) in po for some subread or subwrite se then (se, sr) in eo.
   (b) if (sr, sw) in po for some subwrite sw, then (sr, sw) in eo.
2. For any subwrite sw in E from a write w from tr where sw is to the same byte-address as sr: either (sw, sr) or (sr, sw) in eo.
3. Let sw be subwrite of a write w (from tr), and assume sw satisfied sr in tr. Then $(sw, sr) \in eo$ and the maximal same-byte-address predecessor subwrite of sr in eo is sw. (The eo-maximum here is well-defined since eo relates all same-byte-address subwrites of E by coherence.)

After proving this, have that embed returns a partial order on the byte-sized subreads and subwrites (already showed acyclicity of eo before) that contains program order (restricted to subreads and subwrites), coherence, and is compatible with reads-from. (By definition, byte-wise coherence is compatible with the si relation.) As this partial order already contains program order, coherence, and every subread is already related to all subwrites to the same address, any linear extension of this partial order on E (linearly order all subevents of E in a way that is compatible with eo) is a witness that tr is a BSC execution. Combined with the result that ARM programs preserve singlecopy-atomicity, have a proof for the BSC+SCA theorem.

Thus only remains to show INV, by induction on i.

Base case: $i = 0$. 1.–3. are vacuously true.

Step case: $i \to i + 1$. Assume INV holds for i, show it also holds for $i + 1$. Therefore, show that the execution of loop i preserves INV. If $i + 1 = \text{length } \mathsf{po}_t + 1$ then nothing to show. So assume $i < \text{length } \mathsf{po}_t$. If $\mathsf{po}_t[i]$ is a write event, then the reads in $\mathsf{po}_t[0..i]$ are the same as in $\mathsf{po}_t[0..i-1]$, and because eo does not change, 1.–3. hold by the induction hypothesis. So assume $r = \mathsf{po}_t[i]$ is a read event.

1. (a) For all reads in $\mathsf{po}_t[0..i-1]$ this is true by the induction hypothesis; for a subread sr of a read $r = \mathsf{po}_t[i]$ command B adds these edges.
   (b) By induction hypothesis this is true for all reads in $\mathsf{po}_t[0..i-1]$, only need to show it is also true for $r = \mathsf{po}_t[i]$. Let sw be subwrite of a write w and sr a subread of r such that $sr \xrightarrow{\mathsf{po}} sw$, so $r \xrightarrow{\mathsf{po}} w$. Show that (sr, sw) is added to eo. To do that, show that before running loop i's command C for sr, (sw, sr) is not in eo^+ and therefore C adds (sr, sw). Assume (sw, sr) is in eo^+ before running command C for sr. Two cases: (sw, sr) is a direct edge or a transitive edge.

Case (sw, sr) is direct edge.

Then it cannot be a po-edge since it is $r \xrightarrow{\mathsf{po}} w$. Therefore assume it is an rf-edge. But this cannot happen: w can only be accepted into the storage subsystem once all barriers between r and w are finished and therefore committed. This in turn requires r to already be satisfied when w propagates to memory.

Case (sw, sr) is transitive edge.

Now let (se, sr) be the last edge in a path from sw to sr in the transitive reduction of eo. Now this was either added as a po edge or as an rf edge.

(se, sr) added as po edge.

Then it is $se \xrightarrow{\mathsf{po}} sr \xrightarrow{\mathsf{po}} sw$ and by induction hypothesis and the fact that all subwrite pairs in program order are eo-related according to po it is (se, sw) in the final eo. But as se is on the path from sw to sr also have (sw, se) in the final eo, which contradicts acyclicity of the final eo. (This uses the fact that embed' never deletes edges from eo.)[3]

(se, sr) added as rf edge.

Then se is a subwrite of some write w' that sr read. Let s' be the state before sr read from se. According to the definition of the satisfy-read transition w' was propagated to the thread of r in s', and by definition of the thread semantics w was not accepted into storage. When the last (in program order) barrier b between r and w was accepted into the storage subsystem, which by definition of the thread subsystem is after s', (w', b) was added to Order, which cannot be deleted by any transition; when w was accepted into storage after that (it must have waited for b to be committed by definition of the thread semantics) (b, w) was added. Therefore it is (w', w) in Order in this and all future states, including s, and therefore also (se, sw) in (the initial and all later) eo. But by assumption (sw, se) in eo^+ before running command C for sr and therefore the final eo, which contradicts the acyclicity of eo.

2. By induction hypothesis this is true for all sr from $\mathsf{po}_t[0..i-1]$, since embed' never deletes edges of eo. So only need to show this is also true for the case that sr is subread of a read $r = \mathsf{po}_t[i]$. Let sw from E be subwrite of a write w, to the same byte-address as sr. Show that loop i relates sw and sr. If (after running commands A, B, and C) at the point of running command D for sr the subevents sr and sw are not related $(sw, sr) \in eo^+$, then command D adds (sr, sw) to eo.
3. Have to show two things: After running loop i, (3.1) for any such sr and sw the subwrite sw is in the eo-prefix of sr, and (3.2) there is no same-address subwrite sw' in-between sw and sr in eo.
   (3.1) In case sr is subread of a read from $\mathsf{po}_t[0..i-1]$ this holds by the induction hypothesis. In case sr is subread of $r = \mathsf{po}_t[i]$ command A adds (sw, sr) to eo.

[3] When showing the existence of certain eo edges or proving a contradiction to the acyclicity of eo in the proof, this will usually refer to the final eo.

(3.2) In case sr is subread of a read from $\mathsf{po}_t[0..i-1]$ this holds by induction hypothesis before running loop i. Loop i preserves this, since by induction hypothesis 2 sr is already eo-related to all subwrites of E to the same address before loop i, and since all same-address subwrites of E are totally ordered by eo. So since eo stays acyclic, loop i cannot change the same-address subwrites in the eo-prefix of sr, or their order.
So let sr be the subread of a read $r = \mathsf{po}_t[i]$, where sr reads from subwrite sw of write w and assume after running loop i there is a same-address subwrite $sw' \neq sw$ of a write w' with edges $sw \to sw' \to sr$ in eo. Let sw' be the eo-maximal such subwrite. (This is well-defined since eo orders all same-address subwrites.) Since eo contains coherence, is acyclic, and orders all same-address subwrites, it must be $sw \xrightarrow{\mathsf{co}} sw'$ and therefore $(w, w') \in s.\mathsf{Order}$. The edge (sw', sr) can either be a direct edge or a transitive edge.

Case (sw', sr) is a direct edge.

Then it must have been added by A or B. So it is $w' \xrightarrow{\mathsf{po}} r$, since by assumption sr reads from sw, not sw'. Let s' be the state before sr reads from sw, let b be the last barrier po-between w' and r. Then there is (w', b) and (b, r) in $s'.\mathsf{Order}$. By definition of the satisfy-read transition it is $(w, r) \in s'.\mathsf{Order}$ and w is propagated to r's thread, so w must be ordered with b.

If it is (b, w) in $s'.\mathsf{Order}$, have $(w', w) \in s'.\mathsf{Order}$, which due to the barrier cannot be deleted. So also $(w', w) \in s.\mathsf{Order}$, contradicting acyclicity of Order.

Therefore it must be $(w, b) \in s'.\mathsf{Order}$. But then b is Order-between w and r and propagated to a common thread with r, and by definition of the satisfy-read transition r, w, and w' must be fully propagated. Since it is $(w, w') \in s.\mathsf{Order}$, by Lemma 4 it must already be $(w, w') \in s'.\mathsf{Order}$. But this contradicts the assumption that sr can read from sw in s', since the write w' overlapping the unsatisfied footprint of r is Order-between w and r and propagated to a common thread with r.

Case (sw', sr) is a transitive edge.

Then pick a path in the transitive reduction of eo from sw' to sr, and let $sw*$ be the last subwrite on the path. Let $sw*$ be subwrite of some write $w*$. Now either $sw' = sw*$ or not

Case $sw' = sw*$.

Then there must only be subreads on the path from sw' to sr in the transitive reduction of eo. Since (sw', sr) is a transitive edge, there is at least one other subread on the path from sw' to sr, let sr', subread of some read r', be the first one. The only direct edges between subreads added by embed' are those added by command B, between subreads in program order. So it is either $sw' \xrightarrow{\mathsf{po}} sr$ or $sw' \xrightarrow{\mathsf{rf}} sr' \xrightarrow{\mathsf{po}} sr$.

In the former case $w' \xrightarrow{\mathsf{po}} r$ there is also a direct (program-order) edge from sw' to sr in eo, which has already been dealt with in the previous case. So assume sr' reads from sw' and $r' \xrightarrow{\mathsf{po}} r$, and so $r' \neq r$.

Let b be the last barrier in program order between r' and r. Then b can only commit if r' is finished and therefore satisfied, and r can only issue into the storage subsystem when b is finished and therefore committed. Let s' be the state when b is committed into the storage subsystem. Since r' is satisfied, w' has propagated to the thread of r' and b, and so POP adds the edge $(w', b) \in s'.\mathsf{Order}$.

Now let s'' be the state after s' when r is issued to the storage subsystem. Then POP adds the edge $(b, r) \in s''.\mathsf{Order}$ and it is still $(w,' b) \in s''.\mathsf{Order}$.

Now assume the state s''' after s'' before r reads sr from sw. So it is $(w, r) \in s'''.\mathsf{Order}$ and w and r are propagated to the same threads by definition of the satisfy-read transition. Therefore both w and w' are propagated to the thread of r' and r and they must be related, since they overlap. Since this edge between w and w' will not be deleted, and remain until the final state, since Order is always acyclic, and since $(w, w') \in s.\mathsf{Order}$ it must also be $(w, w') \in s'''.\mathsf{Order}$.

So have $(w, w') \in s'''.\mathsf{Order}$ but also still have $\{(w', b), (b, r)\} \subseteq s'''.\mathsf{Order}$ (the former cannot be deleted by any transition, the latter only by entirely satisfying r) and therefore transitively the edge $(w', r) \in s'''.\mathsf{Order}$. But then by definition of the satisfy-read transition r cannot satisfy sr from sw in s''', since the write w' overlapping the unsatisfied footprint of r is Order-between w and r and propagated to a common thread with r, contradiction.

Case $sw' \neq sw*$.

It is $sw* \neq sw$ (otherwise contradiction to acyclicity of eo). Therefore there are two cases now for the reason for the edge between $sw*$ and sr: either (A) $sw* \xrightarrow{\mathsf{po}} sr$ ("directly or transitively") or, (B) there is a subread $sr*$ of a read $r*$ so that $sw* \xrightarrow{\mathsf{rf}} sr*$ and $r* \xrightarrow{\mathsf{po}} r$.

(A) Let s' be the state before sr is satisfied. Then there is $(w*, b)$ and (b, r) in $s'.\mathsf{Order}$ for the last barrier b po-between $w*$ and r, and by definition of the satisfy-read transition it is $(w, r) \in s'.\mathsf{Order}$ and w is propagated to r's thread. Thus w must be ordered with b. If it is (b, w) the edge $(w*, w)$ is in $s'.\mathsf{Order}$ and since the edge cannot be deleted due to the barrier also $(w*, w)$ in $s.\mathsf{Order}$ and therefore $(sw*, sw) \in eo$, which contradicts acyclicity of eo.

Therefore assume it is $(w, b) \in s'.\mathsf{Order}$. But then b is Order-between r and w and propagated to a common thread with r, forcing r and therefore $w*$ to be fully propagated in s' by definition of the satisfy-read transition. Since by Corollary 2 it will be (w, w') and $(w', w*)$ in $s.\mathsf{Order}$, by Lemma 4 there must already be (w, w') and $(w', w*)$ in $s'.\mathsf{Order}$. Then w' fully propagated. But this contradicts the assumption that r can read from w, since the write w' is Order-between w and r, overlaps the unsatisfied footprint of r, and is propagated to a common thread with r.

(B) Let s' be the state before sr is satisfied. Then $r*$ must have read from $w*$ and $w*$ was propagated to r's thread before the last barrier b po-between $r*$ and r was committed. When b was committed $(w*, b)$ was added to the Order of this state, which remains in all future states; when r was issued (b, r) was added, which — by assumption that tr has no restarts or discards — cannot be deleted before r is entirely satisfied.

Now in s' it is $(w, r) \in s'.\mathsf{Order}$ and the write w is propagated to the thread of r and must be Order-related with b. As before, if it is ordered (b, w), then it is $(w*, w)$ in $s'.\mathsf{Order}$ and also $s.\mathsf{Order}$ and therefore $(sw*, sw) \in eo$, contradicting the acyclicity of eo where it already is $(sw, sw'), (sw', sw*)$.

Thus assume it is $(w, b) \in s'.\mathsf{Order}$. But then b is Order-between w and r and propagated to a common thread with r. So r and $w*$ must be fully propagated. By Corollary 2 it will eventually be $(w', w*)$ in $s.\mathsf{Order}$. Thus this edge must by Lemma 4 already be in $s'.\mathsf{Order}$, forcing full propagation of w'. Also it will by Corollary 2 be (w, w') in $s.\mathsf{Order}$, which by Lemma 4 means it must already be in $s'.\mathsf{Order}$. But then have (w, w') and by transitivity also the edge (w', r) in $s'.\mathsf{Order}$, and w' is propagated to a common thread with r. But this contradicts the assumption that in s' the read r can read from w, because the write w' is Order-between w and r, propagated to a common thread with r, and overlaps the unsatisfied footprint of r.

□

Appendix D

Flat operational model and ARMv8-axiomatic equivalence proof

The following gives the proof of equivalence between Flat and the ARMv8-axiomatic model[1].

D.1 Flat Operational behaviour included in ARMv8 Axiomatic

Let (po, rf, co, rmw) be a candidate execution. Ignoring weak acquire loads ("Q"), ARMv8 Axiomatic is the following:

```
let ca = fr | co
let obs = rfe | fre | coe
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [R]
        | addr; po; [W]
        | (addr | data); rfi
        | (ctrl | data); coi
let aob = rmw
        | [range(rmw)]; rfi; [A]
let bob = po; [DMB.SY]; po
        | [L]; po; [A]
        | [R]; po; [DMB.LD]; po
        | [A]; po
        | [W]; po; [DMB.ST]; po; [W]
        | po; [L]
        | po; [L]; coi
let rec ob = obs | dob | aob | bob | ob; ob
acyclic po—loc | ca | rf as internal
irreflexive ob as external
empty rmw & (fre; coe) as atomic
```

This can be simplified for the purposes of the proof:

1. Since ca is used only in the definition of the internal axiom it can be inlined there.
2. Assume there is a cycle using an edge from [R];po;[dmb.ld];po. Then there is also one just using [R];po;[dmb.ld];po[R|W]: all aob|dob|bob edges are subset of program order and therefore acyclic; and all edges in obs start from a read or write.
3. In the same way, replace [A];po with [A];po;[R|W],
4. po;[L] with [R|W];po;[L], and
5. po;[L];coi with [R|W];po;[L];coi.
6. The recursion in the definition of ob can be replaced by transitive closure.

With these simplifications, the model is equivalent to the one below:

```
let obs = rfe | fre | coe
```

[1] As before, the proof is a minor adaptation of that found in the supplementary material of Pulte et al. [104].

```
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [R]
        | addr; po; [W]
        | (addr | data); rfi
        | (ctrl | data); coi
let aob = rmw
        | [range(rmw)]; rfi; [A]
let bob = po; [dmb.sy]; po
        | [L]; po; [A]
        | [R]; po; [dmb.ld]; po; [R|W]
        | [A]; po; [R|W]
        | [W]; po; [dmb.st]; po; [W]
        | [R|W]; po; [L]
        | [R|W]; po; [L]; coi
let ob = (obs | aob | dob | bob)+
acyclic po—loc | fr | co | rf as internal
irreflexive ob as external
empty rmw & (fre; coe) as atomic
```

Lemma 16. *Let Tr be a valid finite trace of Flat Operational that induces the relations* po*,* co*,* rf*, and* rmw*. Then there is a valid finite trace Tr' that induces the same* po*,* co*,* rf*, and* rmw *relations but has no restarts and no discarded instruction tree branches ("always speculates the correct successor instruction of a branch").*

□

Note The following often uses some notation for "pattern-matching" on different cases of elements of a relation of events and for introducing names for the related events. For instance, in a case distinction, the case "*Edge*: write W → read R" covers the cases where *Edge* relates a write to a read and, for the following description, introduces the name W for the write and R for the read. Similarly, the case "*Edge*: write W' → write W", covers the situation where *Edge* relates a write to another write and introduces the name W' for the first write and W for the second.

D.1.1 Show ob acyclic

Lemma 17. *Let Tr be a valid finite trace of Flat Operational that has no restarts or discarded instruction tree branches that induces the (finite) candidate execution $x = (\mathsf{po}, \mathsf{co}, \mathsf{rf}, \mathsf{rmw})$. Let Edge be an edge from ARMv8-ax's* ob *relation for x. Then the following property holds for Tr, depending on the type of Edge:*

- *Edge: barrier E → barrier E'. In Tr E is committed before E'.*
- *Edge: barrier E → write E'. In Tr E is committed before E' is propagated.*
- *Edge: write E → barrier E'. In Tr E is propagated before E' is committed.*
- *Edge: write E → write E'. In Tr E is propagated before E'.*
- *Edge: barrier E → read R. In Tr E is committed before R is satisfied.*
- *Edge: write E → read R. In Tr E is propagated before R is satisfied.*
- *Edge: read R → barrier E. In Tr R is satisfied before E is committed.*
- *Edge: read R → write E. In Tr R is satisfied before E is propagated.*
- *Edge: read R → read R'. In Tr R is satisfied before R'.*

Proof. By induction on the definition of ob.

Induction start: *Edge* ∈ obs | aob | dob | bob. Then there are multiple cases:

***Edge* ∈ rfe.**

Then $Edge : W \rightarrow R$ and W and R are from different threads. In Flat Operational for R to read from W, W must be propagated to memory, so the induction statement holds.

***Edge* ∈ coe.**

Then $Edge$: $W \rightarrow W'$, and W and W' in co. By assumption the coherence order induced by Tr is the same as co. By definition, if (W, W') in Flat Operational's coherence order, then W is propagated to memory before W' in Tr. So the induction statement holds.

***Edge* ∈ fre.**

Then $Edge : R \rightarrow W$, R and W are from different threads, and there is W' with $(W', R) \in$ rf and $(W', W) \in$ co. Now there are two cases:

***R* is satisfied by forwarding from *W′*.** Then R is satisfied before W' propagates in Tr and by the proof for $Edge \in$ coe the write W' propagates to memory before W. So R is satisfied before W propagates to memory and the induction statement holds.

***R* satisfied in memory.** Then when R is satisfied W' is already propagated and in Tr the write W' propagates before W. Since R reads from the last write to the location of R and $(W', R) \in$ rf it must be that W' is in memory and W is not yet in memory when R reads. So R satisfies before W is propagated to memory.

***Edge* ∈ rmw.**

Then $Edge : R \rightarrow W$. Then R and W are a successful load/store exclusive pair and by definition of the store commit transition the read R must be finished, and therefore satisfied, for W to commit (before it propagates).

***Edge* ∈ [range(rmw)];rfi;[A].**

Then $Edge : W \rightarrow R$. Then W is a successful store exclusive and R an acquire read. Since by definition of Flat Operational R cannot read from W by forwarding W must propagate to memory before R can satisfy.

***Edge* ∈ addr.**

Then either $Edge : R \rightarrow R'$ or $Edge : R \rightarrow W$.

***Edge* : *R* → *R′*.** In Tr the read R' can only be satisfied if initiated, so after its footprint/address is available, so R must be satisfied before R' is.

***Edge* : *R* → *W*.** In Tr the write W can only propagate if initiated, so after its footprint/address is available, so R must be satisfied before W propagates.

***Edge* ∈ data.**

Then $Edge : R \rightarrow W$. W can only propagate when its data is available, hence when its data-feeding memory reads are satisfied. So R is satisfied before W propagates in Tr.

***Edge* ∈ ctrl;[W].**

Then $Edge : R \rightarrow W$. W can only propagate when committed, and by definition of the store commit transition only commit when all program-order-preceding conditional/computed branches are finished. These can only finish after the memory reads feeding into their

register reads, including R, are finished, and hence satisfied. So R is satisfied before W propagates in Tr.

***Edge* ∈ (ctrl | (addr;po)); [ISB];po;[R].**

Then $Edge : R \rightarrow R'$. By definition of the read-request-condition R' can only satisfy if all po-earlier `isb` barriers are finished and therefore committed. By definition of the barrier-commit transition any such `isb` can only commit if all preceding conditional/computed branch instructions are finished and preceding memory accesses have fully determined memory footprints. Therefore the memory reads feeding into the conditional/computed branches po-before the `isb`, including R, must be finished, and hence satisfied, and all memory accesses po-preceding the `isb` must have computed their footprint/address and their footprint/address-feeding memory reads, including R, must be finished, and hence satisfied. Therefore R must be satisfied before R' can be satisfied in Tr.

***Edge* ∈ addr;po;[W].**

Then $Edge : R \rightarrow W$. The write W can only propagate when committed, and commit only when the footprint/address-feeding memory reads of all po-earlier memory accesses, including R, are finished, and hence satisfied. Therefore R must be satisfied in Tr before W propagates.

***Edge* ∈ (addr|data);rfi.**

Then $Edge : R \rightarrow R'$. For R' to satisfy, the write it reads from must have both footprint/address and data available, so any memory read feeding into the footprint/address or data register reads of this write, including R, has to be satisfied. Therefore R must be satisfied before R' can satisfy.

***Edge* ∈ (ctrl|data);coi.**

Then $Edge : R \rightarrow W$ and there is W' such that $(R, W') \in (\mathsf{ctrl}|\mathsf{data})$ and $(W', W) \in \mathsf{coi}$. By definition of how Flat Operational induces coherence W' must propagate to memory before W does. Before W' propagates it must commit, and hence it must have computed its data and the data must be fully determined, and preceding conditional/computed branches must be finished. Therefore all memory reads feeding into the data registers reads of W' and all memory reads feeding into the register reads by conditional/computed branch instructions po-before W' must be finished, and hence satisfied. Therefore R must be satisfied before W' can propagate, before W can propagate in Tr.

***Edge* ∈ po;[dmb.sy];po.**

Let *DMB* be the `dmb`. Now there are different cases for the type of the event on the right of the edge. (*RBW* stands for a read, write, or barrier).

Case *Edge* : *RBW* → *R*. R can only satisfy when *DMB* is finished, hence committed. *DMB* can only commit when all po-earlier memory access instructions and barriers are finished, and hence satisfied (for reads)/committed (for barriers)/propagated (for writes).

Case *Edge* : *RBW* → *B*. B can only commit when *DMB* is finished, hence committed. *DMB* can only commit when all po-earlier memory access instructions and barriers are finished, and hence satisfied (for reads)/committed (for barriers)/propagated (for

writes).

Case *Edge* : *RBW* → *W*. *W* can only propagate when committed, and it can only commit when *DMB* is finished, hence committed. *DMB* can only commit when all po-earlier memory access instructions and barriers are finished, and hence satisfied (for reads)/committed (for barriers)/propagated (for writes).

***Edge* ∈ [L];po;[A].**

Then *Edge* : *W* → *R*. By definition of the read-request-condition, the acquire read *R* can only satisfy when all po-earlier release writes are finished and hence propagated. So *W* propagates before *R* satisfies in *Tr*.

***Edge* ∈ [R];po;[dmb.ld];po;[R|W].**

Let *B* be the `dmb ld`. There are two cases for the type of the event on the right of the edge.

Case *Edge* : *R* → *W*. Then *B* can only commit after *R* is finished, and hence satisfied. The write *W* can only commit and hence propagate after *B* is finished and hence committed. So *R* is satisfied before *W* propagates in *Tr*.

Case *Edge* : *R* → *R′*. Then *B* can only commit after *R* is finished, and hence satisfied. By definition of the read-request-condition R' can only satisfy if *B* is finished and therefore committed. So *R* is satisfied before R' in *Tr*.

***Edge* ∈ [A];po;[R|W].**

Now there are two cases for the type of the event on the right of the edge.

Case *Edge* : *R* → *R′*. Then by definition of the read-request-condition *R* must be finished or entirely satisfied before R' can satisfy.

Case *Edge* : *R* → *W*. Then by definition of the store-commit transition *W* can only commit and hence propagate after *R* is finished and hence satisfied in *Tr*.

***Edge* ∈ [W];po;[dmb.st];po;[W].**

Then *Edge* : $W \rightarrow W'$. Let *B* be the `dmb st`. Then by definition of the barrier-commitment transition the barrier *B* can only commit when *W* is finished and hence propagated and W' can only commit and hence propagate when *B* is finished and hence committed. So *W* propagates before W' in *Tr*.

***Edge* ∈ [R|W];po;[L].**

There are two cases for the type of the event on the left of the edge.

Case *Edge* : *R* → *W*. Then by definition of the store-commit transition *W* can only commit and hence propagate when *R* is finished and hence satisfied in *Tr*.

Case *Edge* : *W* → *W′*. Then by definition of the store-commit transition W' can only commit and hence propagate when *W* is finished and hence propagated in *Tr*.

***Edge* ∈ [R|W];po;[L];coi.**

Let *L* be the write release. There are two cases for the type of the event on the left of the edge.

Case *Edge* : *R* → *W*. As shown above *L* can only propagate after *R* is satisfied, and by definition of the write-propagate transition *W* can only propagate when *L* is propagated. So *R* must be satisfied before *W* can propagate in *Tr*.

Case *Edge* : *W* → *W′*. As shown above *L* can only propagate after *W* is propagated, and

by definition of the write-propagate transition W' can only propagate when L is propagated. So W must propagate before W' in Tr.

Induction step: $Edge = Edge'; Edge'' \in$ ob;ob. This holds by transitivity of implication. □

Corollary 3. *Let Tr be a finite trace of Flat Operational that has no restarts or discarded instruction tree branches that induces the relations* rf, co, rmw, *and* po. *Then ARMv8-axiomatic's* ob *relation is acyclic for the (finite) candidate execution* (po, rf, co, rmw).

Proof. Assume a cycle in ARMv8-axiomatic's ob. Let *Edge* be the cycle. So $Edge : R \to R$ or $Edge : B \to B$ or $Edge : W \to W$.

Case $Edge : R \to R$. Then from Lemma 17 follows that in *Tr* R is satisfied before R is satisfied. But since *Tr* has no restarts or discarded branches, R can only be satisfied once, contradiction.

Case $Edge : B \to B$. Then from Lemma 17 follows that in *Tr* B is committed before B is committed. But every barrier is only committed once, contradiction.

Case $Edge : W \to W$. Then from Lemma 17 follows that in *Tr* W is propagated before W is propagated. But every write is only propagated once, contradiction.

□

D.1.2 Show po-loc | fr | co | rf acyclic

Lemma 18. *Let Tr be a valid finite trace of Flat Operational that has no restarts or discarded instruction tree branches that induces* $x = (\mathsf{po}, \mathsf{co}, \mathsf{rf}, \mathsf{rmw})$. *Then* po-loc | fr | co | rf *acyclic.*

Proof. Assume a cycle in po-loc | fr | co | rf as induced by the trace *Tr* and let C be a cycle that is derived using a minimal number of po-loc edges, and among the cycles with the same number of po-loc edges minimises the total number of edges. Then C is derived with exactly one po-loc edge.

Assume C derived with no po-loc edge. Then $C \in (\mathsf{rf} \mid \mathsf{co} \mid \mathsf{fr})+$. But then $C \in (\mathsf{rf} \mid \mathsf{co})$: assume it is not, so it includes at least one fr edge. (By type rf^{-1};co itself is acyclic.) $C = C'; (E_1, E_2); (E_2, E_3)$ with $(E_2, E_3) \in \mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$. So there exists W such that $(E_2, W) \in \mathsf{rf}^{-1}$ and $(W, E_3) \in \mathsf{co}$. Since E_2 is a read, by type $(E_1, E_2) \in \mathsf{rf}$, and it is $E_1 = W$. But then $C'; (W, E_3)$ is a shorter cycle with no po-loc edges.

So $C \in (\mathsf{rf} \mid \mathsf{co})+$. But then by type $C \in \mathsf{co}+$: No edge in $(\mathsf{rf} \mid \mathsf{co})+$ starts from a read, so rf cannot participate in the cycle C. By definition of Flat Operational's co relation it is $(W, W') \in \mathsf{co}$ only if W propagates to memory before W' in *Tr*. But since every write only propagates once, co+ is acyclic, contradiction.

So assume C derived with at least one po-loc edge. Show it was not derived with multiple po-loc edges. Assume C was derived with multiple po-loc edges. Then it must be $C = C'; (E_1, E_2); C''; (E_3, E_4)$ with $(E_1, E_2) \in$ po-loc and $(E_3, E_4) \in$ po-loc and without loss of generality assuming C'' does not contain a po-loc edge. Since po-loc; po-loc $\subseteq$ po-loc assume also C'' non-empty. So $C'' \subseteq (\mathsf{co}|\mathsf{rf}|\mathsf{fr})+$. Now there are different cases for the type of $(E_1, E_2) \in$ po-loc:

$(E_1, E_2) : W \to W'$. The writes E_1 and E_2 must be coherence related, and it must be $(E_1, E_2) \in \mathsf{co}$ (otherwise it is $(E_2, E_1) \in \mathsf{co}$ and $(E_2, E_1); (E_1, E_2)$ is a cycle in co; po-loc with fewer po-loc edges). But then C can be derived without using (E_1, E_2) as po-loc but using (E_1, E_2) as co, contradicting the minimality of C's po-loc edges.

$(E_1, E_2) : W \to R$. If it is $(E_1, E_2) \in \mathsf{rf}$ then (E_1, E_2) can be derived using rf instead of po-loc, contradicting the minimality of po-loc edges in C. So assume (E_1, E_2) not in rf. So $(W', E_2) \in \mathsf{rf}$ for some W'. If $(W, W') \in \mathsf{co}$ it is $(E_1, E_2) \in \mathsf{co}; \mathsf{rf}$; contradicting the minimality of po-loc edges in C. So assume $(W', W) \in \mathsf{co}$. But then it is $(E_2, E_1) \in \mathsf{fr}$ and $(E_2, E_1); (E_1, E_2)$ is a cycle in fr; po-loc with a smaller number of po-loc edges than C.

$E_1 \to E_2 : R \to W$. Let W' be such that $(W', R) \in \mathsf{rf}$. Now it is either $W' = W$ or not. If $W' = W$ then $(E_2, E_1); (E_1, E_2)$ is a cycle in rf; po-loc with fewer po-loc edges. So assume $W \neq W'$. Then it is either $(W, W') \in \mathsf{co}$ or $(W', W) \in \mathsf{co}$. If $(W', W) \in \mathsf{co}$ then $(E_1, E_2) \in \mathsf{fr}$ and the same cycle can be derived with fewer po-loc edges, contradiction. If $(W, W') \in \mathsf{co}$ then $(W', E_1); (E_1, E_2); (E_2, W')$ is a cycle in rf; po-loc; co with fewer po-loc edges.

$E_1 \to E_2 : R \to R'$. Now there are two cases: R and R' read the same write or not. Assume R and R' read from the same write W. By type $C'' = (E_2, E_6); C'''$ with $(E_2, E_6) \in \mathsf{fr}$. But since R and R' read from the same write it is also $(E_1, E_6) \in \mathsf{fr}$. Contradiction, since then $C'; (E_1, E_6); C'''; (E_3, E_4)$ is a cycle with fewer po-loc edges.

So assume $(W, R) \in \mathsf{rf}$ and $(W', R') \in \mathsf{rf}$ for $W \neq W'$. Now there are two cases: $(W, W') \in \mathsf{co}$ or $(W', W) \in \mathsf{co}$. If $(W, W') \in \mathsf{co}$ it is $(R, W') \in \mathsf{fr}$, so (E_1, E_2) can be derived using fr; rf contradicting the minimality of po-loc edges in C. So assume $(W', W) \in \mathsf{co}$. But then $(R', W) \in \mathsf{fr}$ and $(E_2, W); (W, E_1); (E_1, E_2)$ is a cycle in fr; rf; po-loc with fewer po-loc edges, contradiction.

So assume C was derived with exactly one po-loc edge. Then $C \in (\mathsf{co} \mid \mathsf{rf} \mid \mathsf{fr})\mathsf{+}; \mathsf{po\text{-}loc}$, so $C = C'; P$ for some $C' \in (\mathsf{co} \mid \mathsf{rf} \mid \mathsf{fr})\mathsf{+}$ and $P \in \mathsf{po\text{-}loc}$. Now look at all possible cases for the type of P.

Case $P : W \to W'$. Then $C' : W' \to W \in \mathsf{co}\mathsf{+}$. Assume otherwise. Then by type it must be $C' = C''; (E_1, E_2); (E_2, E_3); C'''$ for some $(E_1, E_2) \in \mathsf{rf}$ and $(E_2, E_3) \in \mathsf{fr}$. But then $(E_1, E_3) \in \mathsf{co}$ and $C''; (E_1, E_3); C'''; P$ is a shorter cycle in $\mathsf{co} \mid \mathsf{rf} \mid \mathsf{fr} \mid \mathsf{po\text{-}loc}$ with the same number of po-loc edges. Contradiction to the assumption of C's minimality.

So $C' \in \mathsf{co}\mathsf{+}$. Then by definition of the operational model's coherence it must be that W' propagates to memory before W. But by definition of the write-propagate transition and $(W, W') \in \mathsf{po\text{-}loc}$ the write W' can only propagate after W is propagated to memory, contradiction.

Case $P : W \to R$. Then C': $R \to W \in \mathsf{fr}$. Assume otherwise. Then by type

- Either $C' \in \mathsf{fr}; \mathsf{co}\mathsf{+} = \mathsf{rf}^{-1}; \mathsf{co}; \mathsf{co}\mathsf{+} \subseteq \mathsf{rf}^{-1}; \mathsf{co}\mathsf{+} \subseteq \mathsf{rf}^{-1}; \mathsf{co} \subseteq \mathsf{fr}$, as by definition of Flat Operational's coherence relation co is transitively closed.
- Or $C' = (E_1, E_2); C''; (E_3, E_4); (E_4, E_5); C'''$ for some $(E_1, E_2) \in \mathsf{fr}$, $(E_3, E_4) \in \mathsf{rf}$, $(E_4, E_5) \in \mathsf{fr}$, and $C'' \in \mathsf{co}^*$ and $C''' \in (\mathsf{co} \mid \mathsf{rf} \mid \mathsf{fr})^*$. But then $(E_3, E_5) \in \mathsf{co}$ and $(E_1, E_2); C''; (E_3, E_5); C'''; P$ is a shorter cycle with the same number of po-loc edges,

contradicting the assumption of minimality of C.

So $C' : R \rightarrow W \in \mathsf{fr}$. Let W' be the write R reads from. So we have $(W', R) \in \mathsf{rf}$ and $(W', W) \in \mathsf{co}$. Now there are two cases: R is satisfied by forwarding or in memory.

***R* is satisfied by forwarding.** Then it must be $(W', R) \in \mathsf{po\text{-}loc}$ because by definition of the satisfy-read-by-forwarding transition W' must be before R in the instruction tree. Now assume there is some write po-between W' and R to the same address in the completed execution. Let W'' be the closest po-predecessor write of R to the same address (this is well-defined for the completed trace).

If the footprint/address of W'' was known at the time of satisfying R from W' by forwarding, by definition of the satisfy-read-by-forwarding transition W' would not have been able to forward to R. So W'' did not have its footprint/address known when W' forwarded to R. But then when W'' later propagated, by definition of the propagate-write transition it restarted R since R did not read from W'' and not from a po-successor of W''. Contradiction to the assumption of Tr not having restarts.

So W' must be the closest po-predecessor write of R to the same address. Now there are two cases: $(W, W') \in \mathsf{po}$ or $(W', W) \in \mathsf{po}$.

$(W, W') \in \mathsf{po}$. But then by definition of the write-propagate transition W' can only propagate when W is propagated, so it is $(W, W') \in \mathsf{co}$. Contradiction to the assumption $(W', W) \in \mathsf{co}$.

$(W', W) \in \mathsf{po}$. Since W' is the closest po-predecessor write of R to the same address it must be $(R, W) \in \mathsf{po}$. Contradiction to $(W, R) \in \mathsf{po\text{-}loc}$.

***R* is satisfied in memory.** Then W' is the most recent write to the same address in memory when R is satisfied. By $(W', W) \in \mathsf{co}$ the write W' propagates before W. So R is satisfied before W propagates. It cannot be the case that W' is program-order-after W, otherwise it would not have been able to propagate to memory before W. Hence, when W propagates by definition of the write-propagate transition R is restarted since R is to the address of W and did not read from W or a write program-order-after W. Contradiction to the assumption that Tr has no restarts.

Case $P : R \rightarrow W$. Then $C' : W \rightarrow R \in \mathsf{co?; rf}$. Proof. Assume otherwise. Then by type C' has to end with an edge from rf, and so $C' = C''; (E_1, E_2); (E_2, E_3); C'''$ with $(E_1, E_2) \in \mathsf{rf}$, $(E_2, E_3) \in \mathsf{fr}$, and $C''' \in \mathsf{co?;rf}$. But then $(E_1, E_3) \in \mathsf{co}$ and $C''; (E_1, E_3); C'''; P$ is a shorter cycle. Contradiction to the assumption of the minimality of C.

So there exists W' such that $(W, W') \in \mathsf{co?}$ and $(W', R) \in \mathsf{rf}$ for some W'. Now there are two cases: $W = W'$ or $W \neq W'$.

$W = W'$. If R is satisfied by forwarding it is $(W, R) \in \mathsf{po}$, contradiction to the assumption. So assume R is satisfied in memory. Then W must be propagated before R is satisfied. But by definition of the propagate-write transition the read R must be satisfied before W can propagate, and it will not be restarted and satisfied again later, contradiction.

$W \neq W'$. Now there are two cases: R satisfied by forwarding or from storage.

***R* satisfied by forwarding.** Then it is $(W', R) \in \mathsf{po\text{-}loc}$ and by $(R, W) \in \mathsf{po\text{-}loc}$ also $(W', W) \in \mathsf{po\text{-}loc}$. But then W can only propagate when W' is propagated and it

must be $(W', W) \in \mathsf{co}$, contradiction.

***R* satisfied in memory.** Then W' reaches memory before R is satisfied. By assumption of no restarts in Tr the read R is only satisfied once. By definition of the write-propagate transition the read R must be satisfied before W propagates. So W' propagates before W and it must be $(W', W) \in \mathsf{co}$, contradiction.

Case $P : R \to R'$. Then $C' : R' \to R \in \mathsf{fr};\mathsf{rf}$. Proof. Assume otherwise. By type C' starts with fr and ends with rf. So it is either (1.) $C' \in \mathsf{fr};\mathsf{co}+;\mathsf{rf}$ or (2.) $C' \in \mathsf{fr};C'';\mathsf{rf};\mathsf{fr};C''';\mathsf{rf}$ for some C'' and C'''.

1. But then $C' \in \mathsf{rf}^{-1};\mathsf{co};\mathsf{co}+;\mathsf{rf} \subseteq \mathsf{rf}^{-1};\mathsf{co}+;\mathsf{rf} \subseteq \mathsf{rf}^{-1};\mathsf{co};\mathsf{rf} \subseteq \mathsf{fr};\mathsf{rf}$.
2. But then $C' \in \mathsf{fr};C'';\mathsf{rf};\mathsf{rf}^{-1};\mathsf{co};C''';\mathsf{rf} \subseteq \mathsf{fr};C'';\mathsf{co};C''';\mathsf{rf}$, for some C'' and C'''. Contradiction to the assumption of minimality of C.

So assume $C' \in \mathsf{fr};\mathsf{rf}$.

Let $(W, R) \in \mathsf{rf}$ and $(W', R') \in \mathsf{rf}$. Then it is $(W', W) \in \mathsf{co}$ by definition of fr. W is either from the same thread as R and R' or not. Assume W is from the same thread as R and R'. Then it must be either $(R, W) \in \mathsf{po}$ or $(W, R) \in \mathsf{po}$. If $(R, W) \in \mathsf{po}$ then there is a cycle in rf;po-loc using only R and W that has already been dealt with in Case $P : R \to W$. If $(W, R) \in \mathsf{po}$, then it is $(W, R) \in \mathsf{po\text{-}loc}$ and $(R, R') \in \mathsf{po\text{-}loc}$, so also $(W, R') \in \mathsf{po\text{-}loc}$ and there is a smaller cycle in po-loc;fr using only R' and W that has been dealt with in Case $P : W \to R$.

So assume W is from a different thread than R and R'. Then R must read from W in memory. It cannot be (R, W') in po since otherwise there would be a smaller cycle of R, W', and W in po-loc;co;rf. Now there are two cases: R' is satisfied before R or after.

***R'* is satisfied before *R*.** If R' is satisfied before R, at the point where R is satisfied by definition of the satisfy-read transitions (satisfy by forwarding or in memory) the read R' is restarted since it reads from a different write from R that is not po-after R, contradicting the assumption of no restarts in Tr. So assume R is satisfied before R'.

***R'* is satisfied after *R*.** By $(W', W) \in \mathsf{co}$ it must be that W' propagates to memory before W does, and since R reads from W in memory, W propagates to memory before R satisfies. So W' propagates, and then W propagates, before R' satisfies. But then R' cannot read from W': it cannot read from W' by thread-local forwarding, since W' is already propagated when R' is satisfied; and it cannot read from W' in memory since W propagated to memory after W' before R' is satisfied, overwriting W'.

Thus in Tr by definition of Flat Operational: $\mathsf{po\text{-}loc} \mid \mathsf{fr} \mid \mathsf{co} \mid \mathsf{rf}$ acyclic. □

D.1.3 Show rmw & (fre; coe) empty

Lemma 19. *Let Tr be a valid finite trace of Flat Operational that has no restarts or discarded instruction tree branches, which induces the relations* po*,* co*,* rf*, and* rmw*. Then* $\mathsf{rmw} \,\&\, (\mathsf{fre};\mathsf{coe})$ *empty.*

Proof. Now assume $\mathsf{rmw} \,\&\, (\mathsf{fre};\mathsf{coe})$ non-empty. So there exists a successful load/store exclusive pair (RE, WE) such that RE reads from a write W and there exists a write W' to the same address as RE but from a different thread such that $(W, W') \in \mathsf{co}$ and $(W', WE) \in \mathsf{co}$. Then it must be that

W propagates to memory before W', and W' propagates to memory before *WE*. Now there are two cases: *RE* is satisfied by thread-internal forwarding or in memory.

***RE* satisfied by forwarding.** Here there are again two cases: (1.) At the point where W propagates to memory *WE* has promised its success or (2.) not.

1. When W propagates to memory it adds the mapping $t = (RE \rightarrow \{\text{full-write-slice } W\})$ into the exclusives map, and by definition of Flat Operational t can only be removed by propagating the write exclusive *WE* paired with *RE*.
   Since W' propagates after W and before *WE*, when W' propagates t is in the exclusives map. But since *RE* and W' are from a different thread and W is to the same address as W', W' cannot propagate, contradiction.
2. *WE* must promise its success before propagating. When *WE* promises its success W is already propagated to memory, by assumption. Since *RE* reads from W by forwarding *RE* must already be satisfied when *WE* promises its success. For *WE* to be able promise its success there can be no write coherence-after W (overlapping W) from a different thread than *RE* in memory. So W' cannot be propagated yet and must propagate after the promise-write-exclusive-success transition of *WE*.
   WE's promise-write-exlusive-success transition now adds the mapping $t = (RE \rightarrow \{\text{full-write-slice } W\})$ to the exclusives map. The mapping t can only be removed from the exclusives map when propagating *WE*; W' propagates before *WE* by assumption. Hence t must be in exclusives map when W' propagates. But since W' overlaps W and is from a different thread than *RE*, the write W' cannot propagate. Contradiction.

***RE* satisfied in memory.** Since *RE* reads from W in memory it has to enter memory after W. *RE*'s satisfy-read transition returns the last memory write written to its location, and since W' propagates after W, the read *RE* satisfies before W' propagates, which in turn is before *WE* propagates. Now there are two cases: 1. when *RE* is satisfied *WE* has promised its success or 2. not.

1. Then when *RE* reads from W in memory it adds $t = (RE \rightarrow \{\text{full-write-slice } W\})$ to the exclusives map. Since t can only be removed when propagating *WE* and since W' propagates before *WE*, when W' propagates t must be in the exclusives map. But since W' is to the same address as W but from a different thread than *RE*, the write W' cannot propagate, contradiction.
2. *WE* must promise its success before propagating. For *WE* to promise its success after *RE* is satisfied there must be no write overlapping W from a different thread coherence-after W in memory. So W' cannot be propagated to memory when promising the success of *WE*.
   The promise-write-exclusive-success transition for *WE* adds the element $t = (RE \rightarrow \{\text{full-write-slice } W\})$ to the exclusives map. Since t can only be removed when propagating *WE* and since W' propagates before *WE*, when W' propagates, t must be in the exclusives map. But since W' is to the same address as W but from a different thread than *RE*, the write W' cannot propagate, contradiction.

Therefore rmw & (fre;coe) empty. □

Theorem 11. *Let Tr be a valid finite trace of Flat Operational that induces the relations* po*,* rf*,* co*, and* rmw*. Then* $x = (\mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{rmw})$ *is a legal finite execution in ARMv8-axiomatic.*

Proof. Let Tr' be an equivalent finite Flat Operational trace that induces the same relations po, rf, co, and rmw and has no restarts or discarded instruction tree branches, by Lemma 16. By Corollary 3 x satisfies the external axiom, by Lemma 18 x satisfies the internal axiom, and by Lemma 19 x satisfies the atomic axiom. □

D.2 Flat-axiomatic definition

In order to show that any behaviour allowed by the ARMv8-axiomatic model is allowed by Flat Operational, we define an intermediate model, called *Flat Axiomatic*. Flat Axiomatic is supposed to capture the details of Flat Operational in an axiomatic model defined in herd. Some rules, mostly concerned with the finishing of loads in Flat, are difficult to precisely define axiomatically; the model slightly over-approximates these in a way that is "locally" stronger but "globally" equivalent for non-mixed-size programs. The model is defined as follows, with the intuition for the definitions given as inlined comments.

```
(* Exclude all executions that violate coherence ... *)
acyclic po—loc | rf | fr | co as internal
(* ... and the read/write exclusive guarantee. *)
empty rmw & (fre;coe) as exclusives

let Xw = range(rmw) (* successful store exclusives *)
let Xr = domain(rmw) (* successful load exclusives *)

(* auxiliary definitions, explained when used in the rules below: *)
let po—R—loc = po—loc \ (po—loc; [W]; po—loc)
let po—no—W—loc = po \ (po; [W]; po—loc)

(* Acq = A, Rel = L, for readability of the rules *)

(* Define the relations in the Flat thread subsystem as relations
between barrier commitment / write propagation / read satisfaction or
finishing.

    R: read, W: write, B: barrier,
    S: satisfy, C: barrier commit / write propagate / read finish

BC_RS, for instance, captures which barriers (on the left—hand side)
have to be committed (or finished) before the reads (on the right—hand
side) can be satisfied. *)

let BC_RS = [DMB.SY|ISB|DMB.LD]; po; [R] (* 1 *)
let WC_RS = [Rel];po;[Acq] (* 2 *)
          | [Xw];rfi;[Acq] (* 3 *)
```

```
                | [W];(po—loc\rf\(po;rf));[R] (* 4 *)
let RS_RS = [Acq];po;[R] (* 5 *)
                | [R];addr;[R] (* 6 *)
                | [R];(addr|data);rfi;[R] (* 7 *)
                | [R];(po—loc\(rf^—1;rf)\(po;rf));[R] (* 8 *)
let RS_RC = id (* 9 *)
let RC_RC = [R];addr;[R] (* 10 *)
                | [R];addr;po—no—W—loc;[R] (* 11 *)
                | [Acq];po;[R] (* 12 *)
                | [R];ctrl;[R] (* 13 *)
                | [R];(addr|data);rfi;[R] (* 14 *)
                | [R];po—R—loc;[R] (* 15 *)
let WC_RC = [Rel];po;[Acq] (* 16 *)
                | [W];(po—R—loc\rf);[R] (* 17 *)
let BC_RC = [DMB.SY|ISB|DMB.LD];po;[R] (* 18 *)
let BC_BC = [DMB.SY];po;[F] (* 19 *)
                | [F];po;[DMB.SY] (* 20 *)
let RC_BC = [R];po;[DMB.SY|DMB.LD] (* 21 *)
                | [R];ctrl;[F] (* 22 *)
                | [R];addr;po;[ISB] (* 23 *)
let WC_BC = [W];po;[DMB.SY|DMB.ST] (* 24 *)
let RC_WC = [R];po;[Rel] (* 25 *)
                | [R];addr;[W] (* 26 *)
                | [R];data;[W] (* 27 *)
                | [R];ctrl;[W] (* 28 *)
                | [R];addr;po;[W] (* 29 *)
                | [Acq];po;[W] (* 30 *)
                | [R];po—loc;[W] (* 31 *)
                | [R];rmw;[W] (* 32 *)
let WC_WC = [W];po—loc;[W] (* 33 *)
                | [W];po;[Rel] (* 34 *)
let BC_WC = [F];po;[W] (* 35 *)

(* For writes, being finished implies being propagated and committed.
   For barrier, being finished implies being committed.

1: BARRIERS: Reads can only satisfy when po—earlier DMB.SYs, ISBs,
   DMB.LDs are finished.
2: REL/ACQ: Load acquires can only satisfy
   when po—earlier write releases are finished.
3: REL/ACQ: Load acquires cannot be satisfied by forwarding from
   store exclusives.
```

4: COHERENCE: The propagation of a write restarts all po—later reads to the same location that did not read from it or a po—later write.
5: REL/ACQ: Reads can only satisfy when po—earlier acquire reads are satisfied.
6: DATAFLOW: Reads cannot satisfy until their address is known.
7: DATAFLOW: Reads can only satisfy when the write they read from has its address and data.
8: COHERENCE: If two reads to the same location, R1 —po—> R2, read from different writes where R2's write is not po—after R1, they must satisfy in order, or R1's satisfaction restarts R2.
9: A read can only finish when it is satisfied.
10: DATAFLOW: Reads can only finish if their dataflow is finished.
11: COHERENCE: Reads can only finish if all instructions up to the closest po—predecessor write to the same address have their address—feeding reads finished.
12: REL/ACQ: Reads can only finish if all po—earlier acquires are finished.
13: COHERENCE: Reads can only finish when the memory reads feeding into po—earlier conditional/computed branches are finished.
14: COHERENCE: Reads can only finish when the dataflow of the write they read from is finished.
15: COHERENCE: (Aproximation) Reads can only finish when all same—location reads in—between them and the closest po—predecessor write to the same address are finished.
16: REL/ACQ: Acquire reads can only finish if all po—earlier release writes are finished.
17: COHERENCE: Reads that are not satisfied by the nearest po—predecessor write to the same location can only finish when that write is propagated.
18: BARRIERS: Reads can only finish if all po—earlier DMB.SYs, ISBs, DMB.LDs are finished.
19: BARRIERS: Barriers can only commit if po—earlier DMB.SYs are finished.
20: BARRIERS: DMB.SYs can only commit if po—earlier barriers are finished.
21: BARRIERS: DMB.SYs and DMB.LDs can only commit if po—earlier reads are finished.
22: BARRIERS: Barriers can only commit if all memory reads feeding into po—earlier conditional/computed branches are finished.
23: BARRIERS: ISBs can only commit if po—earlier memory accesses

have their address—feeding memory reads finished.
24: BARRIERS: DMB.SYs and DMB.STs can only commit if po—earlier writes are finished.
25: REL/ACQ: Release writes can only commit if po—earlier reads are finished.
26: COHERENCE: Writes can only commit if they have calculated their address and have fully determined data.
27: COHERENCE: Writes can only commit if they have calculated their value and have fully determined data.
28: COHERENCE: Writes can only commit if the memory reads feeding into po—earlier conditional/computed branches are finished.
29: COHERENCE: Writes can only commit if po—earlier memory accesses have their address—feeding memory reads finished.
30: REL/ACQ: Writes can only commit if po—earlier read acquires are finished.
31: COHERENCE: (Approximation) Writes can only propagate if po—predecessor same—address reads are finished.
32: EXCLUSIVES: Exclusive writes can only commit after their matching exclusive reads are finished.
33: COHERENCE: (No write subsumption) writes can only propagate when all po—previous writes to the same location are propagated.
34: REL/ACQ: Release writes can only commit when po—previous writes are finished.
35: BARRIERS: A write can only commit when po—earlier barriers are finished.

*)

(* Now compose these edges in the relation "order", where
order(E,E') iff
E commits (if E is a barrier) / propagates (E write) / satisfies (E read)
before
E' commits (E' barrier) / propagates (E' write) / satisfies (E' read).

Edges XX_RS; RC_YY are composable, edges XX_RC; RS_YY are not. To list all the edges that are composable — remove all edges XX_RC and replace them with edges of type XX_RS, XX_BC, or XX_WC;

So:
— delete RC_RC
— replace RS_RC by RS_RC; RC_RC*; (RC_BC | RC_WC)
— replace WC_RC by WC_RC; RC_RC*; (RC_BC | RC_WC)
— replace BC_RC by BC_RC; RC_RC*; (RC_BC | RC_WC)

The resulting order is below.

— Coherence in operational flat axiomatic is determined by the order in which writes propagate to memory. Conversely, for the candidate execution to be allowed by the operational model this write propagation order has to be compatible with the other constraints on the thread behaviour. Therefore include co in the order.

— If a read reads in storage it reads the most recent write that went into memory before it. Conversely, to get an operational trace where the reads—from is as in the candidate execution for (w,r) in rf the write w has to propagate before r is satisfied and any write w' with (w,w') co —— so (r,w') in fr —— must not propagate until after r is satisfied. If a read reads by forwarding it must do this before the read write propagates and therefore also before any coherence later writes propagate. So include fr in the order.

— For a read to read from a different—thread write it has to go into memory after it. So include rfe in the order.

To match a valid Flat trace with no restarts or discards this order has to be acyclic.

*)

```
let Order = BC_RS
          | WC_RS
          | RS_RS
          | RS_RC; RC_RC*; (RC_BC | RC_WC)
          | WC_RC; RC_RC*; (RC_BC | RC_WC)
          | BC_RC; RC_RC*; (RC_BC | RC_WC)
          | BC_BC
          | RC_BC
          | WC_BC
          | RC_WC
          | WC_WC
          | BC_WC

          | co
          | rfe
          | fr

Acyclic (Order+) as external
```

D.3 Flat Axiomatic behaviour included in Flat Operational

D.3.1 Auxiliary result for load/store exclusives

Lemma 20. *Let* $(\mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{rmw})$ *be a finite candidate execution accepted by Flat-axiomatic. Let* S *be a linearisation of* Order *for this candidate execution. Let* $(R, W) \in \mathsf{rmw}$ *with* R *and* W *to the same address,* WR *the write such that* $(WR, R) \in \mathsf{rf}$*. Then*

1. $(WR, W) \in S$,
2. $(R, W) \in S$,
3. *and there is no* W' *with* $(WR, W') \in S$, $(W', W) \in S$ *for a write* W' *to the same address as* W *but from a different thread than* W.

Proof. It is $(R, W) \in \mathsf{rmw} \subseteq \mathsf{RC\_WC} \subseteq S$, so (2.) holds.

Now either $(WR, R) \in \mathsf{rfi}$, then $(WR, R) \in \mathsf{po}$ by internal axiom and by $(R, W) \in \mathsf{po}$ also have $(WR, W) \in \mathsf{po}$. Therefore it is $(WR, W) \in \mathsf{[W];po\text{-}loc;[W]} \subseteq \mathsf{WC\_WC} \subseteq S$, so have (1.).

Or $(WR, R) \in \mathsf{rfe} \subseteq S$ and by $(R, W) \in \mathsf{rmw} \subseteq \mathsf{RC\_WC} \subseteq S$ it is $(WR, W) \in S$, so have (1.).

Since co totally orders same-address writes and $\mathsf{co} \subseteq S$ it is $(WR, W) \in \mathsf{co}$ by (1.). (3.) Now assume there is such a write W'. Then since $\mathsf{co} \subseteq S$ and co totally orders same-address writes, $(WR, W') \in \mathsf{co}$ and $(W', W) \in \mathsf{co}$. But then $(R, W') \in \mathsf{fre}$ and $(W', W) \in \mathsf{coe}$ so that $(R, W) \in \mathsf{(fre;coe)\ \&\ rmw}$. Contradiction to the exclusives axiom. □

Lemma 21. *Let* $(\mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{rmw})$ *be a finite candidate execution accepted by Flat-axiomatic. Let* $(R, W) \in \mathsf{rmw}$ *and* $(R', W') \in \mathsf{rmw}$ *and* R, R', W, W' *all from the same thread to the same address, with* $R \neq R'$*. Let* $(RW, R) \in \mathsf{rf}$ *and* $(RW', R') \in \mathsf{rf}$*. Then it cannot be* $RW = RW'$.

Proof. Assume $RW = RW'$. We know $W \neq W'$, since otherwise the pairing of W with R and W' with R' would not be possible. It is $(R, W) \in \mathsf{po}$, and $(R', W') \in \mathsf{po}$. Assume without loss of generality $(R, R') \in \mathsf{po}$. Now because of $(R, W) \in \mathsf{rmw}$ it cannot be $(R, R'); (R', W) \in \mathsf{po}$. So po must order them as $R; W; R'; W'$. Have $RW \neq W$, since otherwise there is a cycle in $\mathsf{rf; po\text{-}loc}$ for R and W. Similarly, it is $RW \neq W'$, since otherwise there is a cycle in $\mathsf{rf; po\text{-}loc}$ for R' and W'. The coherence order must be $(RW, W) \in \mathsf{co}$, since otherwise there is a cycle in $\mathsf{rf; po\text{-}loc; co}$. So it is $(R', W) \in \mathsf{fr}$. Contradiction to internal axiom: cycle in $\mathsf{fr|po\text{-}loc}$. □

D.3.2 Write-finish lemma

Lemma 22. *Let St be a state of Flat operational with all enabled eager non-fetch, non-barrier-commit transitions taken. For any store* W*: if the write of* W *is propagated, then* W *is finished.*

Proof. Since W by assumption is aligned, W has only a single write. Hence after propagating, W was eagerly completed. Since all register reads done by a store are reads to determine its footprint/address and data, such register reads have already been done by W, and the pseudocode execution of W can be eagerly finished. Since W has committed its data must be fully determined and all program-order-preceding conditional/computed branches are finished. Thus the condition for finishing W holds and W has been eagerly finished. □

D.3.3 Main proof

Theorem 12. *Let* (po, rf, co, rmw) *be a finite candidate execution allowed in Flat-axiomatic. Then there exists a finite trace Tr of Flat Operational that induces the candidate execution* (po, rf, co, rmw).

Proof. Let S be a linearisation of Order for this candidate execution. Define rfs $= S$&rf, rft $=$ rf $\setminus$ rfs, $Rs =$ range rfs, $Rt =$ range rft. Then rfe $\subseteq$ rfs since rfe $\subseteq$ Order.

Trace construction

Now construct Tr as follows:

1. Start with an empty trace.
2. Fetch all instructions one-by-one following po of the candidate-execution. In program order, for each write exclusive W:
   - if $W \in$ range rmw promise the success of the write exclusive
   - if $W \notin$ range rmw promise the failure of the write exclusive
3. (Repeatedly) Take the enabled eager non-fetch, non-barrier-commit transitions.
4. For each next element E of S:
   - 4.1. **If E is a read $R \in Rt$** satisfy R by forwarding from the unique W with $(W,R) \in$ rf using the transition $T =$ satisfy-read-by-forwarding for R and W.
     If E is a read $R \in Rs$ satisfy R in memory with $T =$ satisfy-read-in-memory for R and the write W with $(W,R) \in$ rf.
     If E is a write W take transition $T =$ propagate-memory-write for W.
     If B is a barrier take transition $T =$ commit-barrier for B.
   - 4.2. (Repeatedly) Take all enabled eager non-fetch, non-barrier-commit transitions (these include the pseudocode-internal transitions, register reads and writes, initiating of reads and writes, instantiating of writes, committing stores, completing loads and stores that have done all their respective reads and writes, and finishing instructions).

Note that the success of store exclusive instructions is determined right after fetching. Subsequently certain eager transitions are taken, including the success register writes of these store exclusive instructions. This ensures that dependencies on store exclusive success register writes do not create memory ordering (as intended in ARMv8). Note also that the boolean recording whether a store exclusive instruction has been determined to succeed is stable under restarts. Hence if another instruction relies on the success register write of a store exclusive instruction this register write is immediately fully determined (see definition of fully determined).

Given a finite candidate execution the above definition constructs a finite trace.

Read-finish lemma

Lemma 23. *Let St be a state of Flat operational reached after a prefix of a trace constructed as above, and with all enabled eager non-fetch, non-barrier-commit transitions taken. For any read R: if R is satisfied, all writes and barriers E with* $(E,R) \in$ (WC_RC | BC_RC); RC_RC* *are finished, and all reads* R' *with* $(R',R) \in$ RC_RC+ *are satisfied, then R is finished.*

Proof. By induction on the instruction tree of the thread of R.

Induction start: empty instruction tree. If the instruction tree is empty there is no such read R in it.

Induction hypothesis: assume the statement holds for some instruction tree *IT*, show it also holds for adding a leaf *II* to *IT*. By the induction assumption the induction hypothesis holds for all satisfied reads R in IT also with II added. (The condition of the finish-instruction transition for loads is unaffected by po-later instructions.) So remains to show that the induction hypothesis holds for II. Let R be the leaf II and assume R is a satisfied read, and that all writes and barriers E with $(E,R) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\mathsf{RC\_RC}^*$ are finished and all reads R' with $(R',R) \in \mathsf{RC\_RC}^+$ are satisfied. Have to show that R is finished.

For R to finish the following have to hold:

1. R has to have fully determined data. Hence, the memory reads feeding into the register reads of R have to be finished. Let R' be one such read. Have $(R',R) \in [\mathsf{R};\mathsf{addr};[\mathsf{R}] \subseteq \mathsf{RC\_RC}$. Therefore R' is satisfied in St, all reads R'' with $(R'',R') \in \mathsf{RC\_RC}^+$ are satisfied since $(R'',R');(R',R) \in \mathsf{RC\_RC}^+;\mathsf{RC\_RC} \subseteq \mathsf{RC\_RC}^+$, and all writes and barriers E' finished for $(E',R') \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\mathsf{RC\_RC}^*$, since have $(E',R');(R',R) \in (\mathsf{WC\_RC}|\mathsf{BC\_RC});\mathsf{RC\_RC}^*;\mathsf{RC\_RC} \subseteq (\mathsf{WC\_RC}|\mathsf{BC\_RC});\mathsf{RC\_RC}^*$. Then by induction hypothesis R' is finished.
2. Conditional/computed branches po-before R have to be finished. Assume there is an unfinished branch instruction po-before R, let BR be the po-earliest one. BR's finish transition is taken eagerly, so if BR is not finished, then it is because BR cannot finish yet.
   Since BR is the po-earliest unfinished branch its control flow is finished. So it must be the dataflow going into BR that is unfinished: there is at least one read R' that feeds into the register reads of BR that is unfinished. But then $(R',R) \in [\mathsf{R};\mathsf{ctrl};[\mathsf{R}] \subseteq \mathsf{RC\_RC}$ and therefore R' satisfied and all writes and barriers E' finished for $(E',R') \in (\mathsf{WC\_RC}|\mathsf{BC\_RC});\mathsf{RC\_RC}^*$, since $(E',R');(R',R) \in (\mathsf{WC\_RC}|\mathsf{BC\_RC});\mathsf{RC\_RC}^*$, and all reads R'' with $(R'',R') \in \mathsf{RC\_RC}^+$ satisfied, since $(R'',R');(R',R) \in \mathsf{RC\_RC}^+$. Therefore by induction hypothesis R' is finished, contradiction.
3. All po-earlier `dmb sy`, `dmb ld`, `isb` are finished. Since $[\mathsf{DMB.SY}|\mathsf{ISB}|\mathsf{DMB.LD}];\mathsf{po};[\mathsf{R}] \subseteq \mathsf{BC\_RC}$, by assumption all of these are finished in St.
4. All po-earlier acquire reads are finished.
   Let R' be a po-earlier acquire. So $(R',R) \in [\mathsf{Acq}];\mathsf{po};[\mathsf{R}] \subseteq \mathsf{RC\_RC}$. Then R' satisfied. And all writes and barriers E' finished for $(E',R') \in (\mathsf{WC\_RC}|\mathsf{BC\_RC});\mathsf{RC\_RC}^*$, by $(E',R');(R',R) \in (\mathsf{WC\_RC}|\mathsf{BC\_RC});\mathsf{RC\_RC}^*$, and all reads R'' with $(R'',R') \in \mathsf{RC\_RC}^+$ are satisfied, since $(R'',R');(R',R) \in \mathsf{RC\_RC}^+$. Then by induction hypothesis R' is finished.
5. If R is an acquire then all po-earlier releases are finished. This is true in St, since by $[\mathsf{Rel}];\mathsf{po};[\mathsf{Acq}] \subseteq \mathsf{WC\_RC}$ these write releases are finished.
6. Let W be the closest po-earlier write with known address to the same address, if such a write exists. Then:
   (a) If such a write W exists and was forwarded to R the memory reads feeding into the register reads of W must be finished.

(b) If such a write W exists and was not forwarded to R it must be propagated.

(c) All memory accesses between this write W and R (if no such W exists, then all memory accesses po-before R) have fully determined footprint/address: have done enough steps to compute their footprint/address and have their footprint/address-feeding memory reads finished.

(d) All reads R' to the same address between W and R (if no such W exists, then all such memory reads po-before R) must be satisfied and not-restartable.

First show that if in the completed Flat-axiomatic execution there exists a closest same address program-order-predecessor write $\widetilde{W}$ to R then its footprint/address is known in St, and so $\widetilde{W} = W$. (By construction of the Flat operational trace there cannot be another write program-order-between $\widetilde{W}$ and R to the same address.) Assume such a write $\widetilde{W}$ exists in the completed Flat-axiomatic execution. Then $(\widetilde{W}, R) \in$ po-no-W-loc;[R]. Then in St the address-feeding memory reads of $\widetilde{W}$ are satisfied: let R' be one such read; then $(R', R) \in$ [R];addr;po-no-W-loc;[R] $\subseteq$ RC_RC, and by assumption R' satisfied. Since all such reads are satisfied, they have eagerly done their register writes, (arithmetic instructions $\widetilde{W}$ might depend on have eagerly done their register reads and writes) and $\widetilde{W}$ has eagerly done its register reads and initiated, and so has known address.

So assume $W = \widetilde{W}$ (if such a $\widetilde{W}$ exists). Now show (a) – (d).

(a) Let R' be one such read. Then $(R', R) \in$ [R];(addr|data);rfi $\subseteq$ RC_RC. Then by assumption R' is satisfied, all reads R'' with $(R'', R') \in$ RC_RC+ are satisfied, since it is $(R'', R'); (R', R) \in$ RC_RC+, and all writes and barriers E' are finished for $(E', R') \in$ (WC_RC | BC_RC); RC_RC*, since $(E', R'); (R', R) \in$ (WC_RC | BC_RC); RC_RC*. Then by induction hypothesis R' is finished.

(b) By assumption R is satisfied in St. Now there are two cases:

$(W, R) \in$ rf. Since by assumption W was not forwarded to R, the read R read from, W, is in memory, so W propagated.

$(W, R) \notin$ rf. Then $(W, R) \in$ [W];(po-R-loc\rf);[R] $\subseteq$ WC_RC, so by assumption W is finished and therefore propagated.

(c) Let R' be one such footprint/address-feeding read. Then $(R', R) \in$ [R];addr;po-no-W-loc;[R] $\subseteq$ RC_RC $\subseteq S$. Therefore R' is satisfied, all reads R'' with $(R'', R') \in$ RC_RC+ are satisfied, since $(R'', R'); (R', R) \in$ RC_RC+, and all writes and barriers E' are finished for $(E', R') \in$ (WC_RC | BC_RC); RC_RC*, since $(E', R'); (R', R) \in$ (WC_RC | BC_RC); RC_RC*. Then by induction hypothesis R' is finished. Moreover, since all such R' are satisfied, all memory accesses between W and R (if no such W exists, then all memory accesses po-before R) have eagerly done the register reads necessary to determine their address and eagerly initiated.

(d) Let R' be such a read. Then $(R', R) \in$ [R];po-R-loc;[R] $\subseteq$ RC_RC. Therefore R' is satisfied, all reads R'' with $(R'', R') \in$ RC_RC+ are satisfied, since $(R'', R'); (R', R) \in$ RC_RC+, and all writes and barriers E' are finished for $(E', R') \in$ (WC_RC|BC_RC); RC_RC*, because it is $(E', R'); (R', R) \in$ (WC_RC|BC_RC); RC_RC*. Then by induction hypothesis R' is finished.

Since R by assumption is aligned, it is the only read of its load instruction. Now have: R is satisfied, so the load is eagerly completed. Since there are no additional register reads to be done by R its pseudocode execution is eagerly finished. Since all the conditions for finishing R hold in St and the read-finish transition is an eager transition, R is finished and the induction hypothesis holds. □

Trace is valid trace

Now show by induction on n: if Tr is the (partial or full) trace constructed for $S[0..n]$, with n within the range of S, then:

0. Assume $RE \rightarrow \{\text{full-write-slice } W\}$ in the exclusives map after executing Tr. Then $(W, RE) \in \mathsf{rf}$ and there exists a write exclusive WE such that $(RE, WE) \in \mathsf{rmw}$. Let WE be this write exclusive. Then $W \in S[0..n]$ and $WE \notin S[0..n]$ and RE satisfied after Tr.
1. For all satisfy-read transitions in Tr for reads R (whether by forwarding or from memory), the write W it reads from is the write in (W, R) in the rf relation from the candidate execution above. For all write-propagate transitions in Tr for writes W, they occur in Tr consistent with the co relation in the candidate execution above. And $(R, W) \in \mathsf{rmw}$ holds in the candidate execution above, if-and-only-if the write exclusive W is successfully paired with R in Tr. The instruction trees viewed as a relation (restricted to reads/writes/barriers) is po.
2. Tr involves no restarts and no discarding of branches in the instruction trees.
3. Tr is a valid (partial or full) trace of Flat Operational.
4. If there are any enabled transitions after Tr, they can only be one of:
    - satisfy memory read by forwarding
    - satisfy memory read from storage
    - propagate memory write
    - commit barrier
    - fetch instruction

Induction start, empty prefix of S, with steps 1 – 3 from the trace construction done.

0. After executing only fetch, promise-write-exclusive-success or promise-write-exclusive-failure, and non-fetch, non-barrier-commit eager transitions the exclusives map is empty. (In the case of promise-write-exclusive-success transitions this is because the paired loads have by construction not been satisfied yet.)
1. Since the fetch and promise-write-exclusive-success/failure transitions and eager transitions do not satisfy reads or propagate writes (1.) holds for rf and co. Since by construction of Tr the success of a write exclusive W is promised if-and-only-if $W \in \mathsf{range}\,\mathsf{rmw}$, (1.) holds for rmw as well. By construction the instruction-trees unfolding matches po. By proof of (2.) the instruction trees are not pruned.
2. By definition of Flat Operational none of these transitions can cause restarts. Neither fetching nor the promise-write-exclusive-success/failure transition discard instruction tree branches. Since the instruction trees viewed as a relation matches po of the candidate execution any eagerly taken finish-instruction transition for a branch will not discard instruction tree

branches; the other eagerly taken transitions by definition do not discard instruction tree branches.

3. By definition of Flat Operational for non-computed conditional branches both possible targets of the branch can be fetched, for computed branches any address can be fetched. Hence, the model allows fetching all the instructions according to po.
   Since rmw will only relate a load exclusive and a store exclusive that can be successfully paired, the promise-write-exclusive-success transitions are enabled for these store exclusives. (Since the paired load exclusive instructions by construction of Tr have not been satisfied yet the store exclusive does not need to provide any atomicity guarantee yet.) Store exclusives can unconditionally fail, so the promise-write-exclusive-failure transitions are also enabled for the store exclusives that do not have a write in rmw.
   By assumption the taken eager transitions are enabled.
4. By definition, all the eager transitions (all except the ones in the inductive hypotheses) are taken.

Induction step: n → n+1. Now extend Tr to Tr' for the next element $E = S[n+1]$ (if $n+1$ within range of S).

Case E is a read R from Rt. Show extending the trace for E preserves properties 0. – 3.

0. Only the promise-write-success, satisfy-read-in-memory and propagate-memory-write transitions change the exclusives map. Since E is a read in Rt and satisfied by forwarding, 0. still holds by induction hypothesis.
1. By induction hypothesis this is true for Tr for rf, co, rmw, and po. Extending Tr to Tr' preserves this: the relations po, co, and rmw are unaffected; only have to show it also holds for rf, but this follows from the construction of Tr': R is satisfied by W for $(W,R) \in \mathsf{rf}$.
2. Have to show that the satisfaction of R does not cause the restart or discarding of any instructions. By definition T does not discard instruction tree branches.
   By definition of the satisfy-read-by-forwarding transition T might restart certain instructions violating coherence with respect to R and their dataflow dependents. We call such violating instructions (without the dataflow dependents) the *restart-roots* set. Show that the *restart-roots* set of instructions is empty, and hence that T restarts no instructions. From that follows that extending the trace for E preserves 2.
   According to the definition of satisfy-read-by-forwarding *restart-roots* is the set of all program-order-later reads R' to the same location where R' is unfinished and has been satisfied, but by neither W nor a write po-after R after executing Tr.
   Assume there is such a read R' that is satisfied by a write W' that is neither W nor po-after R after executing Tr. Then have $(R,R') \in \mathsf{po\text{-}loc}$, and therefore $(R,R') \in [\mathsf{R}];(\mathsf{po\text{-}loc} \setminus (\mathsf{rf}^{-1};\mathsf{rf}) \setminus (\mathsf{po};\mathsf{rf}));[\mathsf{R}] \subseteq \mathsf{RS\_RS} \subseteq S$. By construction R' can only have already been satisfied if $(R',R) \in S$. But already have $(R,R') \in S$. Contradiction to the acyclicity of S.
3. To be able satisfy R from W by thread-internal forwarding,

3.1. R must be in Pending_mem_reads state, not already satisfied, and the read-request-condition hold,
3.2. W must be in state Pending_mem_writes with data available, and not propagated yet,
3.3. W must be before R in the instruction tree,
3.4. there must be no write W' to the same address between R and W in the instruction tree,
3.5. there must be no R' to the same address in between R and W in the instruction tree that read from another write $W' \neq W$,
3.6. if R is a read acquire, then W is not a store exclusive.

Now show those conditions hold.

3.1. After executing Tr the read R is in Pending_mem_reads state, it is not satisfied yet, and the read-request-condition holds.

The reads feeding into the register reads of R have been satisfied and therefore eagerly completed: for any read R' whose read value feeds into the address of R it is $(R', R) \in \mathsf{RS\_RS} \subseteq \mathsf{Order} \subseteq S$; therefore by construction R' is satisfied and eagerly completed after executing Tr; the register writes of any write exclusive WE's success bits feeding into the address of R are eagerly done, since by construction WE has determined its success/failure; any other, non-memory, instructions feeding into R's registers reads have been done eagerly. Hence R has eagerly done its register reads and initiated, and therefore is in state Pending_mem_reads.

R cannot already be satisfied yet after Tr: by construction R could only be satisfied after Tr if $(R, R) \in S$, but S is acyclic. Remains to show that the read-request-condition holds. Then 3.1. follows.

Hence, show:

3.1.1. All program-order-earlier `dmb sy`, `isb`, `dmb ld` are finished. By definition of BC_RS have [DMB.SY|ISB|DMB.LD];po;[R] $\subseteq$ BC_RS $\subseteq S$. So by construction of Tr all such `dmb sy`, `isb`, `dmb ld` are committed after executing Tr, and therefore eagerly finished after Tr.

3.1.2. If R is an acquire read then all po-earlier write releases are finished. It is [Rel];po;[Acq] $\in$ WC_RS $\subseteq$S. So by construction all po-earlier write releases are propagated if R is an acquire, and therefore eagerly completed and finished after Tr.

3.1.3. All po-earlier acquire reads are entirely satisfied. It is [Acq];po;[R] $\subseteq$ RS_RS $\subseteq S$. So by construction all acquires po-before R are satisfied after executing Tr.

3.2. After executing Tr the write W is in state Pending_mem_writes with data available and not propagated yet.

As shown above, rfe $\subseteq$ rfs. Therefore, since rft $=$ rf $\setminus$ rfs it is $(W, R) \in$ rfi, so W and R from the same thread.

The memory reads feeding into the register reads of W have been completed: for any read R' that feeds into W's address or data it is $(R', R) \in$ [R];(addr|data);rfi;[R] $\subseteq$

RS_RS $\subseteq$ Order $\subseteq S$; therefore by construction R' is satisfied after executing Tr and therefore eagerly completed; any register write of a write exclusive's success bit feeding into the address or data of W has been done eagerly, since all write exclusives have already promised success or failure after Tr; all non-memory instructions feeding into W's registers reads have been done eagerly. Hence, W has eagerly initiated and instantiated and is in state Pending_mem_writes.
Remains to show that W has not propagated yet.
By construction W can only be propagated if $(W,R) \in S$, so assume $(W,R) \in S$. But by definition of rft it is $(R,W) \in S$. Since S acyclic, contradiction: $(W,W) \in S$.

3.3. This follows by definition of the operational model if $(W,R) \in$ po. As shown above, $(W,R) \in$ rfi, so W and R from the same thread. And it must be $(W,R) \in$ po because the internal axiom requires the acyclicity of po-loc | rf.

3.4. This follows by definition of the operational model if there is no $(W,W') \in$ po and $(W',R) \in$ po for a write W' to the same address. There cannot be such $(W,W') \in$ po and $(W',R) \in$ po since by the internal axiom in the candidate execution po-loc | co | rf | fr acyclic.

3.5. Assume there is such a read R'.
Then by construction $(R',R) \in S$, and by induction hypothesis $(W',R') \in$ rf of the candidate execution. Now in the candidate execution W and W' must be coherence related. If it is $(W',W) \in$ co, then $(R',W');(W',W) \in \mathsf{rf}^{-1};\mathsf{co} = \mathsf{fr}$ and there is a cycle in fr; po-loc, contradiction to the assumption that Flat-axiomatic's internal axiom holds for the candidate execution.
So assume it is $(W,W') \in$ co. By assumption it is $(W,R) \in$ rf, so $(R,W') \in$ fr of the candidate execution. But then there is a cycle in fr; rf; po-loc in the candidate execution. Contradiction to the assumption that the internal axiom holds.

3.6. Assume R is a read acquire and W a store exclusive. But then by definition of rfs the read R is required to be in Rs, since [Xw];rfi;[A] $\subseteq$ WC_RS $\subseteq$ Order $\subseteq S$, contradiction.

Case E is a read R from Rs. Show extending the trace for E preserves properties 0. – 3.

0. The transition T for R only changes the exclusives map in case R is a read exclusive RE with a program-order-following write exclusive WE for which Tr contains the promise-write-exclusive-success transition. By construction in that case $(RE,WE) \in$ rmw. Let W be the write with $(W,RE) \in$ rf. Then T adds the element $RE \rightarrow \{\text{full-write-slice } W\}$ to the exclusives map. It is $(W,RE) \in$ rfs $\subseteq S$, so $W \in S[0..n+1]$. By construction, after Tr' the read $R = RE$ is satisfied. Left to show that $WE \notin S[0..n+1]$. This follows by $(RE,WE) \in$ fr $\subseteq S$. Since R is a read, for the other elements in the exclusives map 0. still holds.
1. By induction hypothesis the trace Tr matches the rf and co relation from the candidate execution, the instruction-trees unfolding matches po, and the load/store-exclusive pairing is as in rmw. Have to show that extending Tr to Tr' for E preserves this. But this follows from the construction: R is satisfied by W for $(W,R) \in$ rf. po, co, and rmw

are unchanged (since, as shown now, R does not cause any instruction restarts or discards).

2. Have to show that the satisfaction of R does not cause the restart or discarding of any instructions. By definition T does not discard instruction-tree branches.
   T might restart some program-order-succeeding loads that violate coherence with respect to R, and their dataflow dependents. We call *restart-roots* these violating loads (without their dependents).
   Then *restart-roots* is the set of all reads R' program-order-after R to the same location where R' was satisfied neither from W nor from a write po-after R. Show that *restart-roots* is empty, from which follows that R causes no instruction restarts, and hence that extending the trace to Tr' for E preserves 2.
   Assume a read R' in *restart-roots* that was satisfied by a write W' that is neither W nor po-after R after Tr. Then $(R,R') \in [\mathsf{R}];(\mathsf{po\text{-}loc} \setminus (\mathsf{rf}^{-1};\mathsf{rf}) \setminus (\mathsf{po};\mathsf{rf}));[\mathsf{R}] \subseteq \mathsf{RS\_RS} \subseteq S$. But by construction R' can only have been satisfied if $(R',R) \in S$. Contradiction to the acyclicity of S.
3. To be able to satisfy R from W in memory after Tr,
   3.1. R must be in Pending_mem_reads state, unsatisfied, and the read-request-condition hold,
   3.2. W must be propagated to memory.
   3.3. There must not be a write W' to the same address that propagated after W
   3.4. If R is a successful exclusive read, in the storage subsystem state there must be no element RE' $\rightarrow$ {full-write-slice W'} in the exclusives map where RE' and R are to the same address but from different threads.

   Now show those conditions hold.

   3.1. After executing Tr the read R is in state Pending_mem_reads, it is unsatisfied, and the read-request-condition holds.
      The memory reads feeding into the register reads of R have been completed: for any read R' whose read value feeds into the address of R it is $(R',R) \in \mathsf{RS\_RS} \subseteq \mathsf{Order} \subseteq S$; therefore by construction R' is satisfied and therefore eagerly completed after executing Tr; the register writes of the success bit of any store exclusive feeding into the register reads of R have been done eagerly since all write exclusives have promised success or failure after Tr; all non-memory instructions feeding into the register reads of R have been done eagerly. Hence, R has been eagerly initiated and is in state Pending_mem_reads. R cannot be satisfied after Tr: by construction R could only have been satisfied if there was $(R,R) \in S$. But S is acyclic.
      Remains to show the read-request-condition holds, then 3.1. follows. This is if:
      3.1.1. All program-order-earlier `dmb sy`, `isb`, `dmb ld` are finished. By definition of BC_RS it is $[\mathsf{DMB.SY}|\mathsf{ISB}|\mathsf{DMB.LD}];\mathsf{po};[\mathsf{R}] \subseteq \mathsf{BC\_RS} \subseteq S$. So by construction of Tr all `dmb sy`, `isb`, `dmb ld` are committed after executing Tr, and therefore eagerly finished.
      3.1.2. If R is an acquire read then all po-earlier write releases are finished. It is

[Rel];po;[Acq] ⊆ WC_RS ⊆ S. So by construction all po-earlier write releases are propagated if R is an acquire, and therefore eagerly completed and finished.

3.1.3. All po-earlier acquires are entirely satisfied. It is [Acq];po;[R] ⊆ RS_RS ⊆ S. So by construction all acquires po-before R are satisfied and eagerly completed after executing Tr.

3.2. After executing Tr the write W is propagated. By definition of rfs it is $(W,R) \in$ rfs ⊆ S, so by construction W has already been propagated.

3.3. Assume after executing Tr there is a write W' to the same address that propagated after W. Then this must be because $(W,W') \in S$ and $(W',R) \in S$. Since co ⊆ S and co totally orders same-address writes it must be $(W,W') \in$ co and therefore $(R,W') \in$ fr ⊆ S. But then S cyclic. Contradiction.

3.4. Assume R is a read exclusive $R = RE$ and $(RE,WE) \in$ rmw. Assume there is such a $RE' \to$ {full-write-slice W'} in the exclusives map. Then by induction hypothesis it is $(W',RE') \in$ rf, there is WE' such that $(RE',WE') \in$ rmw, $W' \in S[0..n]$, RE' satisfied after Tr', and WE' not in $S[0..n]$. Then have $(W',RE) \in S$. Also have $(W,RE) \in$ rfs ⊆ S. Since RE' is satisfied, by construction it must be $RE' \in S[0..n]$ and therefore $(RE',RE) \in S$. And it is $(RE,WE') \in S$.

Also have $(RE,WE) \in$ fr ⊆ S. Since by assumption RE and RE' from different threads it must also be that WE and WE' are from different threads.

Now there are two cases:

$(WE,WE') \in S$. Then it is $(W',RE) \in S$, $(RE,WE) \in S$, and $(WE,WE') \in S$. But WE and WE' are to the same address from different threads, contradicting Lemma 20 for $(RE',WE') \in$ rmw.

$(WE',WE) \in S$. Then it is $(W,RE) \in S$, $(RE,WE') \in S$, and $(WE',WE) \in S$. But WE and WE' are to the same address from different threads, contradicting Lemma 20 for $(R,WE) \in$ rmw.

Case E is a write W. Show extending the trace for E preserves properties 0. – 3.

0. Assume $t = RE' \to$ {full-write-slice W'} in the exclusives map after executing Tr'. Now there are two possibilities:

t in the exclusives map before T. Then by IH there is $(W',RE') \in$ rf and there exists WE' such that $(RE',WE') \in$ rmw with $W' \in S[0..n]$ and WE' not in $S[0..n]$ and RE' satisfied after Tr. Have to show T preserves this. rf and rmw are unaffected by T; after T it will still be W' in $S[0..n+1]$ and RE' still satisfied (by proof of 2.). So have to show WE' not in $S[0..n+1]$. Assume it is. Then $W = WE'$. But then $T =$ propagate-memory-write W deletes $RE' \to$ {full-write-slice W'} from the exclusives map, contradiction.

otherwise. Then RE' is a read exclusive that was satisfied by thread-internal forwarding from $W = W'$ and it is paired with a write exclusive WE' for which Tr contains the promise-write-success transition. Then have $(W,RE') \in$ rf. By construction it is $(RE',WE') \in$ rmw. RE' is satisfied (and remains satisfied after T by proof of 2.) and it is W in $S[0..n+1]$. Still have to show that WE' not in $S[0..n+1]$.

By $(RE', WE') \in \mathsf{rmw}$ it is $(RE', WE') \in \mathsf{po}$. Since RE' read from W by thread-internal forwarding it is also $(W, RE') \in \mathsf{po}$, so $(W, WE') \in \mathsf{po}$ and they are writes to the same location. But then it is $(W, WE') \in [\mathsf{W}];\mathsf{po\text{-}loc};[\mathsf{W}] \subseteq \mathsf{WC\_WC} \subseteq S$, and since $W = S[n+1]$ it must be that $WE' \notin S[0..n+1]$.

1. By induction hypothesis this holds for Tr. Have to show that extending Tr to Tr' for E preserves this. Since in the operational model the coherence relation is determined by the order in which writes reach memory, have to show: (1.1.) for all $(W', W) \in \mathsf{co}$ the write W' has already been propagated after Tr and (1.2.) for all W' to the same address as W that have been propagated before W in Tr it is $(W', W) \in \mathsf{co}$.
   1.1. Since $\mathsf{co} \subseteq S$ by construction of Tr all such writes W' have already been propagated.
   1.2. Let W' be any same-address write that is propagated after in Tr. Then by construction of Tr it must be $(W', W) \in S$. But since $\mathsf{co} \subseteq S$, since co totally orders all same-address writes, and since S is acyclic it must be $(W', W) \in \mathsf{co}$.

   po, rf, and rmw are unaffected by T, so 1. follows.
2. Have to show that the propagation of W does not cause the restart or discarding of any instructions. By definition T does not discard instruction-tree branches.
   By definition of the propagate-memory-write transition T might restart certain instructions violating coherence with respect to W and their dataflow dependents. We call such violating instructions (without the dataflow dependents) the *restart-roots* set. Show that the *restart-roots* set of instructions is empty, and hence that T restarts no instructions. From that follows that extending the trace for E preserves 2.
   Then *restart-roots* is the set of all program-order-later non-finished reads R that have been satisfied from a write to the same location as W, but neither from W nor from a write po-after W, after Tr. Assume after executing Tr, R is such a read that is satisfied by write W' that is neither W nor po-after W. So $(W, R) \in [\mathsf{W}];(\mathsf{po\text{-}loc} \backslash \mathsf{rf} \backslash (\mathsf{po};\mathsf{rf}));[\mathsf{R}] \subseteq \mathsf{WC\_RS} \subseteq S$. By construction of Tr the read R can only have already been satisfied if $(R, W) \in S$. Contradiction to the acyclicity of S.
3. To propagate W, W must be in state Pending_mem_writes with its data available, its store has to be committed and the conditions for propagating W have to hold.
   First show that after executing Tr the write W is in state Pending_mem_writes. The memory reads feeding into the register reads of W have been satisfied: for any read R whose read value feeds into the address and data of W it is $(R, W) \in \mathsf{RC\_WC} \subseteq \mathsf{Order} \subseteq S$; therefore by construction of Tr the read R is satisfied and eagerly completed after executing Tr; the register writes of the success bit of any write exclusives are done eagerly since all write exclusives have promised their success or failure; all non-memory data dependent instructions that feed into the register reads of W have been done eagerly. Hence, W has eagerly initiated and instantiated and is in state Pending_mem_writes.
   Remains to show W is committed and the conditions for propagating W hold. Since write commitment transitions are eager only have to show the conditions for committing W hold. From this follows that W has been eagerly committed. Hence, remains

to show that the conditions for committing W and the conditions for propagating W hold.

(The conditions 3.1. – 3.6. and 3.12. are conditions sufficient for committing W, the others for propagating W. We prove some conditions for propagating W before 3.12., for convenience.)

3.1. All po-earlier dmb sy, isb, dmb ld, dmb st are finished.
3.2. If W is a release then all po-earlier reads and writes are finished.
3.3. All po-earlier read acquires are finished.
3.4. W has fully determined data.
3.5. Conditional/computed branches po-before W have to be finished.
3.6. All po-previous memory accesses have their address-feeding memory reads finished and are initiated.
3.7. W is unpropagated.
3.8. All program-order-previous same-address writes have to be propagated.
3.9. All po-previous memory reads to the same address must be satisfied, and not restartable.
3.10. Any read R that was partially satisfied from W must be entirely satisfied.
3.11. There exists no *RE*' → {full-write-slice W'} in the exclusives map after *Tr* where *RE*' and W are to the same address but from different threads.
3.12. If W is a successful store exclusive *WE* paired with a load exclusive *RE*, then *RE* is finished, and if *RE* read from a same-thread write W', that write is propagated.

Now prove the above conditions hold:

3.1. It is [F];po;[W] ⊆ BC_WC ⊆ Order ⊆ S. So by construction of *Tr* all po-earlier dmb sy, isb, dmb ld, dmb st are committed, and therefore eagerly finished.
3.2. Assume W is a write release and let W' be a po-earlier write. Then by $(W', W) \in$ [W];po;[Rel] ⊆ WC_WC ⊆ S and by construction of *Tr*, the write W' is propagated and therefore eagerly finished. Let R be a program-order-earlier read. Then by $(R, W) \in$ [R];po;[Rel] ⊆ RC_WC ⊆ S, and by construction of *Tr* the read R is satisfied. Moreover, by construction of *Tr* all reads R' with $(R', R) \in$ RC_RC+ are satisfied, since it is $(R', R); (R, W) \in$ RC_RC+; RC_WC ⊆ S, and all writes E' are propagated and barriers E' are committed and E' thus eagerly finished for all $(E', R) \in$ (WC_RC|BC_RC);RC_RC*, since $(E', R); (R, W) \in$ (WC_RC|BC_RC); RC_RC*; RC_WC ⊆ S. Then by Lemma 23 R is finished.
3.3. Let R be a read acquire po-before W. Then $(R, W) \in$ [Acq];po;[W] ⊆ RC_WC ⊆ S. So by construction of *Tr* the read R is satisfied, all reads R' with $(R', R) \in$ RC_RC+ are satisfied, by $(R', R); (R, W) \in$ RC_RC+;RC_WC ⊆ S, and all writes E' are propagated and barriers E' are committed and E' therefore eagerly finished for $(E', R) \in$ (WC_RC|BC_RC);RC_RC*, since $(E', R); (R, W) \in$ (WC_RC|BC_RC);RC_RC*; RC_WC. Then by Lemma 23, R is finished.
3.4. The memory reads feeding into the register reads of W have to be finished. Let R be one such read. Then the read value of R feeds into the address or data of

W and have $(R, W) \in$ [R];(addr|data);[W] $\subseteq$ RC_WC $\subseteq S$. Then by construction R is satisfied and eagerly completed, all reads R' with $(R', R) \in$ RC_RC+ are satisfied since $(R', R); (R, W) \in$ RC_RC+; RC_WC and writes E are propagated and barriers E committed and E therefore eagerly finished for $(E, R) \in$ (WC_RC | BC_RC); RC_RC* since $(E, R); (R, W) \in$ (WC_RC | BC_RC); RC_RC*; RC_WC. Then by Lemma 23 R is finished.

3.5. Conditional/computed branches po-before W have to be finished.
Assume there is an unfinished branch instruction po-before W, let BR be the po-earliest one. The finish transition for BR is taken eagerly, so if BR is unfinished, then it is because BR cannot finish yet.
Since BR is the po-earliest unfinished branch its control flow is finished. So it must be the dataflow of BR that is unfinished: there is at least one read R that feeds into the register reads of BR that is unfinished.
But then $(R, W) \in$ [R];ctrl;[W] $\subseteq$ RC_WC $\subseteq S$. So by construction of Tr the read R is satisfied, all writes E are propagated and barriers E are committed for $(E, R) \in$ (WC_RC | BC_RC); RC_RC* are therefore eagerly finished, since $(E, R); (R, W) \in$ (WC_RC | BC_RC); RC_RC*; RC_WC $\subseteq S$, and all reads R' with $(R', R) \in$ RC_RC+ are satisfied, since $(R', R); (R, W) \in$ RC_RC*; RC_WC $\subseteq S$. So by Lemma 23 R is finished.

3.6. All po-previous memory accesses have their address-feeding memory reads finished and are initiated.
Let R be such an address-feeding read. Then $(R, W) \in$ [R];addr;po;[W] $\subseteq$ RC_WC $\subseteq S$. So by construction R is satisfied, all reads R' with $(R', R) \in$ RC_RC+ are satisfied, since $(R', R); (R, W) \in$ RC_RC+; RC_WC $\subseteq S$, and all writes E' are propagated and barriers E' are committed and thus E' eagerly finished for all E' with $(E', R) \in$ (WC_RC|BC_RC); RC_RC*, since for these it is $(E', R); (R, W) \in$ (WC_RC | BC_RC); RC_RC*; RC_WC. Then by Lemma 23 R is finished. Moreover, since all such R are satisfied, all memory accesses po-before W have eagerly done the register reads necessary to determine their address and have eagerly initiated.

3.7. W is unpropagated. By construction, W can only be propagated if $(W, W) \in S$. But S is acyclic, hence W unpropagated.

3.8. All program-order-previous same-address writes have to be propagated. Have [W];po-loc;[W] $\subseteq$ WC_WC $\subseteq S$, so all po-earlier writes to the address of W are propagated (and therefore eagerly finished).

3.9. All po-previous memory reads to the same address must be satisfied, and not restartable.
Let R be one such read. Then it is $(R, W) \in$ [R];po-loc;[W] $\subseteq$ RC_WC $\subseteq S$ and by construction the read R is satisfied. Remains to show that R is not restartable. Show instead that R is already finished.
All reads R' with $(R', R) \in$ RC_RC$^+$ are satisfied, since $(R', R); (R, W) \in$ RC_RC*; RC_WC. And all writes E' are propagated and barriers E' are committed and hence E' eagerly finished for $(E', R) \in$ (WC_RC | BC_RC); RC_RC*, since

for these E' it is $(E', R); (R, W) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC}); \mathsf{RC\_RC}^*; \mathsf{RC\_WC}$. Then by Lemma 23 R is finished.

3.10. Any read R that was partially satisfied from W must be entirely satisfied.
By assumption all memory accesses have the same size, so if R was partially satisfied from W it is entirely satisfied.

3.11. There exists no $RE' \rightarrow$ {full-write-slice W'} in the exclusives map after Tr where RE' and W are to the same address but from different threads.
Assume there exists $RE' \rightarrow$ {full-write-slice W'} in the exclusives map after Tr where RE' and W are to the same address but from different threads.
Then by 0. of the induction hypothesis it is $(W', RE') \in \mathsf{rf}$ and there exists a write exclusive WE' such that $(RE', WE') \in \mathsf{rmw}$, $W' \in S[0..n]$, and WE' not in $S[0..n]$. Since RE' and W from different threads by assumption, also W and WE' from different threads, so $WE' \neq W$ and WE' not in $S[0..n+1]$ either, and therefore $(W, WE') \in S$. Since it is $W' \in S[0..n]$ it is also $(W', W) \in S$. So $(W', W) \in S$ and $(W, WE') \in S$ where W and WE' are from different threads but to the same address. Contradiction to Lemma 20 for $(RE', WE') \in \mathsf{rmw}$.

3.12. Assume W is a successful store exclusive WE that is paired with a load exclusive RE. Then RE must be po-before WE. By proof of 3.9. all po-earlier memory reads to the same address are already finished, so RE finished. Assume RE read from a same-thread write W'. Then it is $(W', RE) \in \mathsf{po}$ and $(RE, W) \in \mathsf{po}$. And since W' and W have the same address, by proof of 3.8. the write W' is propagated.

Case E is a barrier B. Show extending the trace for E preserves properties 0. – 3.

0. Committing a barrier does not change the exclusives map, and since E is not a write, 0. still holds.
1. By induction hypothesis this holds for Tr. Since B does not fetch, satisfy reads, propagate writes, or promise the success/failure of write exclusives this is still true for Tr'.
2. Have to show that committing B does not restart any instructions or discard instruction-tree branches. But this follows by definition of the barrier-commit transition.
3. Show Tr' is a valid trace of Flat Operational: committing B is enabled after Tr.
   3.1. Conditional/computed branches po-before B have to be finished.
   Assume there is an unfinished branch instruction po-before B, let BR be the *po*-earliest one. The finish transition for BR is taken eagerly, so if BR is unfinished, then it is because BR cannot finish yet.
   Since BR is the *po*-earliest unfinished branch its control flow is finished. So it must be BR's dataflow that is unfinished: there is at least one read R that feeds into the register reads of BR that is unfinished. But then $(R, B) \in [\mathsf{R}];\mathsf{ctrl};[\mathsf{F}] \subseteq \mathsf{RC\_BC} \subseteq S$. So by construction of Tr the read R is satisfied, all writes E are propagated and barriers E are committed and E therefore eagerly finished for $(E, R) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC}); \mathsf{RC\_RC}^*$, since $(E, R); (R, B) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC}); \mathsf{RC\_RC}^*; \mathsf{RC\_BC}$, and all reads R' with $(R', R) \in \mathsf{RC\_RC}+$ are satisfied, since $(R', R); (R, B) \in$

RC_RC*; RC_BC ⊆ S. Then R is finished.

3.2. If B is a dmb sy all po-earlier barriers, reads, and writes are finished. Assume B is a dmb sy.
Let B' be a barrier po-before B. Then $(B', B) \in$ [F];po;[DMB.SY] ⊆ BC_BC ⊆ S, so by construction of Tr the barrier B' is committed and therefore eagerly finished.
Let W be a po-earlier write. then $(W, B) \in$ [W]; po; [DMB.SY] ⊆ WC_BC ⊆ S. So by construction of Tr the write W is propagated and therefore eagerly completed and finished.
Let R be a po-earlier read. So $(R, B) \in$ [R];po;[DMB.SY] ⊆ RC_BC ⊆ S. So by construction R is satisfied, all writes E are propagated and barriers E are committed and E thus eagerly finished for $(E, R) \in$ (WC_RC | BC_RC); RC_RC*, since (E, R); $(R, B) \in$ (WC_RC | BC_RC); RC_RC*; RC_BC ⊆ S, and all reads R' with $(R', R) \in$ RC_RC+ are satisfied, since (R', R); $(R, B) \in$ RC_RC+; RC_BC. Then by Lemma 23 R is finished.

3.3. All po-earlier dmb sy are finished. Let B' be a dmb sy po-before B. Then it is $(B', B) \in$ [DMB.SY];po;[F] ⊆ BC_BC ⊆ S. So by construction of Tr the barrier B' is committed and therefore eagerly finished.

3.4. If B is an isb all po-earlier memory accesses have their address-feeding memory reads finished and have initiated.
Let R be a memory read feeding into the address of a po-earlier memory access. Then $(R, B) \in$ [R]; addr; po; [ISB] ⊆ RC_BC ⊆ S. So by construction R is satisfied, all writes E are propagated and barriers E committed and E therefore eagerly finished for $(E, R) \in$ (WC_RC | BC_RC); RC_RC*, since (E, R); $(R, B) \in$ (WC_RC | BC_RC); RC_RC*; RC_BC ⊆ S, and all reads R' with $(R', R) \in$ RC_RC+ are satisfied, since (R', R); $(R, B) \in$ RC_RC+; RC_BC. Then by Lemma 23 R is finished. Moreover, since all such R are satisfied, all memory accesses po-before B have eagerly done the register reads necessary to determine their address and have eagerly initiated.

3.5. If B is a dmb ld all po-earlier memory loads are finished. Assume B is a dmb ld. Let R be a po-earlier read. Then $(R, B) \in$ [R];po;[DMB.LD] ⊆ RC_BC ⊆ S. So by construction R is satisfied, all writes E are propagated and barriers E are committed and E therefore eagerly finished for $(E, R) \in$ (WC_RC | BC_RC); RC_RC*, since (E, R); $(R, B) \in$ (WC_RC | BC_RC); RC_RC*; RC_BC ⊆ S, and all reads R' with $(R', R) \in$ RC_RC+ are satisfied, since (R', R); $(R, B) \in$ RC_RC+; RC_BC. Then by Lemma 23 R is finished.

3.6. If B is a dmb st all po-earlier memory stores are finished. Assume B is a dmb st. Let W be a po-earlier write. Then $(W, B) \in$ [W];po;[DMB.ST] ⊆ WC_BC ⊆ S. So by construction of Tr the write W is propagated and therefore eagerly finished.

Take eager transitions. Repeatedly extend the trace Tr' to Tr'' for enabled eager (non-fetch, non-barrier-commit) transitions T, until there are no more such enabled transitions.

0. Only the promise-write-success, satisfy-read-in-memory and propagate-memory-write

transitions change the exclusives map. These are not eager, so the exclusives map is unchanged. And since the prefix of S has not changed: 0. still holds.

1. By definition of transition-eagerness, T does not fetch, satisfy a read, propagate a write, or promise success or failure of a write exclusive, so 1. is preserved. (By proof of 2. there are no instruction restarts or discards.)
2. Restarts are caused only by the transitions satisfy-read-by-forwarding, satisfy-read-from-memory, and propagate-memory-write. By definition these transitions are not eager, so T does not cause restarts. Only finishing of branch instructions can cause instruction tree branches to be discarded. A branch BR can only finish when the memory reads feeding into its register reads are finished. Let R be any such memory read. If $R = S[n+1]$ (R was the last event from S to be handled before the eager steps) then R reads from the unique W such that $(W, R) \in \mathsf{rf}$ by construction; for all other R the induced reads-from relation is a subset of rf of the candidate execution by induction hypothesis. So the successor of BR is determined as in the candidate execution's po, and by induction hypothesis the instruction trees viewed as a relation matches po. Therefore, finishing such a branch BR does not discard any instruction tree branches.
3. Since by assumption T is enabled, Tr'' is a valid trace.
4. When no more such transitions are enabled, establish property 4.: the eager (non-fetch, non-barrier-commit) transitions have all been taken by construction.

□

D.4 ARMv8 Axiomatic behaviour included in Flat Axiomatic

Theorem 13. *Let C be a finite candidate execution accepted by ARMv8-axiomatic. Then C is accepted by Flat-axiomatic.*

Proof. Since the internal and atomic (or exclusives in Flat-axiomatic) axioms are the same in both models, we only have to show that when ARMv8-axiomatic's axioms hold, Flat-axiomatic's external axiom holds: Order is acyclic. To do this, we start with Order, and show step-by-step that any edges in Order that are not directly included in ARMv8-axiomatic's ob relation can be deleted safely: if there is a cycle in Order with these edges then there is also one in the relation without them.

```
(Order)+
= (BC_RS
  | WC_RS
  | RS_RS
  | RS_RC; RC_RC*; (RC_BC | RC_WC)
  | WC_RC; RC_RC*; (RC_BC | RC_WC)
  | BC_RC; RC_RC*; (RC_BC | RC_WC)
  | BC_BC
  | RC_BC
  | WC_BC
  | RC_WC
  | WC_WC
  | BC_WC
  | co
```

```
    | rfe
    | fr
    )+
```

Apply most of the definitions.

```
  ...
  = ([DMB.SY|ISB|DMB.LD]; po; [R]
    | [Rel];po;[Acq]
    | [Xw];rfi;[Acq]
    | [W];(po—loc \ rf \ (po;rf));[R]
    | [Acq];po;[R]
    | [R];addr;[R]
    | [R];(addr|data);rfi;[R]
    | [R];(po—loc \ (rf^-1;rf) \ (po;rf));[R]
    | RC_RC*; (RC_BC | RC_WC)
    | WC_RC; RC_RC*; (RC_BC | RC_WC)
    | BC_RC; RC_RC*; (RC_BC | RC_WC)
    | [DMB.SY];po;[F]
    | [F];po;[DMB.SY]
    | [R];po;[DMB.SY|DMB.LD]
    | [R];ctrl;[F]
    | [R];addr;po;[ISB]
    | [W];po;[DMB.SY|DMB.ST]
    | [R];po;[Rel]
    | [R];addr;[W]
    | [R];data;[W]
    | [R];ctrl;[W]
    | [R];addr;po;[W]
    | [Acq];po;[W]
    | [R];po—loc;[W]
    | [R];rmw;[W]
    | [W];po—loc;[W]
    | [W];po;[Rel]
    | [F];po;[W]
    | co
    | rfe
    | fr
    )+
```

Simplify.

```
  ...
  = ([DMB.SY];po
    | po;[DMB.SY]
    | [F];po;[W]
    | [R];po;[DMB.LD]
    | [W];po;[DMB.ST]
    | [R];addr;po;[ISB]
    | [ISB|DMB.LD]; po; [R]
    | ctrl;[F]
    | po;[Rel]
    | [Acq];po;[R|W]
    | [Rel];po;[Acq]
    | [Xw];rfi;[Acq]
    | rmw
    | addr
    | data
    | ctrl;[W]
    | addr;po;[W]
    | (addr|data);rfi
    | RC_RC*; (RC_BC | RC_WC)
    | WC_RC; RC_RC*; (RC_BC | RC_WC)
    | BC_RC; RC_RC*; (RC_BC | RC_WC)
    | co
    | rfe
    | fr
    | [R];(po—loc \ (rf^-1;rf) \ (po;rf));[R]
    | [W];(po—loc \ rf \ (po;rf));[R]
```

```
  | [W];po—loc;[W]
  | [R];po—loc;[W]
  )+
```

[W];po-loc;[W] is included in co, so can delete this edge. [R];po-loc;[W] included in fr, so can delete this edge.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC)
  | WC_RC; RC_RC*; (RC_BC | RC_WC)
  | BC_RC; RC_RC*; (RC_BC | RC_WC)
  | co
  | rfe
  | fr
  | [R];(po—loc \ (rf^-1;rf) \ (po;rf));[R]
  | [W];(po—loc \ rf \ (po;rf));[R]
  )+
```

Consider $(W,R) \in$ [W];(po-loc\rf\(po;rf));[R]. By definition $(W,R) \notin$ rf. Let $(W',R) \in$ rf with $W \neq W'$. By definition it is not $(W,W') \in$ po. By coherence axiom it also cannot be $(W',W) \in$ po, because otherwise cycle in fr;po-loc. So $(W',R) \in$ rfe, and by coherence axiom again it must be $(W,W') \in$ co, otherwise cycle in fr;po-loc. So then $(W,R) \in$ co;rfe. So the above edge is subsumed by co;rfe.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC)
  | WC_RC; RC_RC*; (RC_BC | RC_WC)
  | BC_RC; RC_RC*; (RC_BC | RC_WC)
  | co
```

```
  | rfe
  | fr
  | [R];(po—loc \ (rf^-1;rf) \ (po;rf));[R]
  )+
```

Consider $(R, R') \in$ [R];(po-loc\ (rf^{-1};rf)\(po;rf));[R] and let $(W, R) \in$ rf, $(W', R') \in$ rf. By definition $W \neq W'$ and $(R, W') \notin$ po. It must be $(W, W') \in$ co as otherwise $(R', W) \in$ fr and there is a cycle in fr;rf;po-loc.

By per-thread-coherence it cannot be $(W', R) \in$ po, since otherwise cycle in fr;po-loc. So W' not from the same thread as R and R' and it is $(R, R') \in$ fr;rfe. So [R];(po-loc \ (rf^{-1};rf) \ (po;rf));[R] subsumed by fr;rfe and can delete the edge.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC)
  | WC_RC; RC_RC*; (RC_BC | RC_WC)
  | BC_RC; RC_RC*; (RC_BC | RC_WC)
  | co
  | rfe
  | fr
  )+
```

Now apply the definitions of WC_RC and BC_RC.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC) (* E2 *)
  | ([Rel];po;[Acq] | [W];(po—R—loc\rf);[R]); RC_RC*; (RC_BC | RC_WC)
  | [DMB.SY|ISB|DMB.LD];po;[R]; RC_RC*; (RC_BC | RC_WC) (* E1 *)
  | co
```

```
  | rfe
  | fr
  )+
```

Have E1 = [DMB.SY|ISB|DMB.LD];po;[R];E2. Here [DMB.SY|ISB|DMB.LD];po;[R] is already contained in the relation using [DMB.SY];po and [ISB|DMB.LD];po;[R]. So E1 is already contained in the relation using [DMB.SY];po, [ISB|DMB.LD];po;[R], and E2, and can delete E1.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC) (* E2 *)
  | ([Rel];po;[Acq] | [W];(po—R—loc\rf);[R]); RC_RC*; (RC_BC | RC_WC) (* E1 *)
  | co
  | rfe
  | fr
  )+
```

Have E1 = ([Rel];po;[Acq] | [W];(po-R-loc\rf);[R]);E2. Now assume W and R such that $(W,R) \in$ [W];(po-R-loc \ rf);[R]. By definition R does not read from W, so assume $(W',R) \in$ rf for a write W' such that $W' \neq W$. W and W' have to be coherence related. It cannot be $(W',W) \in$ co, because then there is a cycle in po-loc;fr. So have $(W,W') \in$ co.

By $(W,W') \in$ co it cannot be $(W',W) \in$ po. By $(W',R) \in$ rf it cannot be $(R,W') \in$ po. And by definition of po-R-loc it cannot be $\{(W,W'),(W',R)\} \subseteq$ po. So W' not from the same thread as R and it is $(W',R) \in$ rfe. Therefore it is $(W,W');(W',R) \in$ co;rfe, so $(W,R) \in$ co;rfe. So [W];(po-R-loc\rf);[R] is included in co;rfe.

Then ([Rel];po;[Acq] | [W];(po-R-loc\rf);[R]) is already included in the above relation using [Rel];po;[Acq] and co and rfe, and E1 is subsumed by the combination of these and E2. So can delete E1.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
```

```
| rmw
| addr
| data
| ctrl;[W]
| addr;po;[W]
| (addr|data);rfi
| RC_RC*; (RC_BC | RC_WC)
| co
| rfe
| fr
)+
```

Now consider RC_RC*:

```
  ([R];addr;[R] | [R];addr;po—no—W—loc;[R] | [Acq];po;[R] | [R];ctrl;[R] |
  [R];(addr|data);rfi;[R] | [R];po—R—loc;[R])*
= (addr;[R] | addr;po—no—W—loc;[R] | [Acq];po;[R] | ctrl;[R] |
  (addr|data);rfi;[R] | [R];po—R—loc;[R])*
```

This is included in the following:

```
(addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
  (addr|data);rfi;[R] | [R];po—R—loc;[R])*.
```

Can rewrite this to the following, using the fact that [R];po-R-loc;[R] is transitive:

```
([R];po—R—loc;[R])?; (addr;[R]
                    | addr;[R];po—R—loc;[R]
                    | addr;po;[R]
                    | addr;po;[R];po—R—loc;[R]
                    | [Acq];po;[R]
                    | [Acq];po;[R];po—R—loc;[R]
                    | ctrl;[R]
                    | ctrl;[R];po—R—loc;[R]
                    | (addr|data);rfi;[R]
                    | (addr|data);rfi;[R];po—R—loc;[R])*
```

But some edges are subsumed by others:

- addr;[R];po-R-loc;[R] by addr;po;[R],
- addr;po;[R];po-R-loc;[R] by addr;po;[R],
- [Acq];po;[R];po-R-loc;[R] by [Acq];po;[R],
- ctrl;[R];po-R-loc;[R] by ctrl;[R].

So rewrite to:

```
([R];po—R—loc;[R])?; (addr;[R]
                    | addr;po;[R]
                    | [Acq];po;[R]
                    | ctrl;[R]
                    | (addr|data);rfi;[R]
                    | (addr|data);rfi;[R];po—R—loc;[R])* =
```

So RC_RC* included in:

```
([R];po—R—loc;[R])?; (addr;[R] | addr;po;[R] | [Acq];po;[R] |
  ctrl;[R] | (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*
```

So can strengthen the order below by including this edge instead.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
```

```
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
        (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*; (RC_BC | RC_WC)
  | co
  | rfe
  | fr
  )+
```

Apply the definition of RC_BC and RC_WC.

```
  ...
  = ([DMB.SY];po
    | po;[DMB.SY]
    | [F];po;[W]
    | [R];po;[DMB.LD]
    | [W];po;[DMB.ST]
    | [R];addr;po;[ISB]
    | [ISB|DMB.LD]; po; [R]
    | ctrl;[F]
    | po;[Rel]
    | [Acq];po;[R|W]
    | [Rel];po;[Acq]
    | [Xw];rfi;[Acq]
    | rmw
    | addr
    | data
    | ctrl;[W]
    | addr;po;[W]
    | (addr|data);rfi
    | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
          (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
            ([R];po;[DMB.SY|DMB.LD] | [R];ctrl;[F] | [R];addr;po;[ISB] |
              [R];po;[Rel] | [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] |
              [R];addr;po;[W] | [Acq];po;[W] | [R];po—loc;[W] | [R];rmw;[W]) (* E *)
    | co
    | rfe
    | fr
    )+
```

Some cases of E are subsumed by other edges in the relation, so can delete these: set of edges ending in [R];po;[DMB.SY|DMB.LD] is subsumed by po;[DMB.SY] and [R];po;[DMB.LD]; the set of edges ending with [R];po;[Rel] is subsumed by po;[Rel].

```
  ...
  = ([DMB.SY];po
    | po;[DMB.SY]
    | [F];po;[W]
    | [R];po;[DMB.LD]
    | [W];po;[DMB.ST]
    | [R];addr;po;[ISB]
    | [ISB|DMB.LD]; po; [R]
```

```
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
        (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
          ([R];ctrl;[DMB.ST|ISB] | [R];addr;po;[ISB] | [R];addr;[W] |
           [R];data;[W] | [R];ctrl;[W] | [R];addr;po;[W] | [Acq];po;[W] |
           [R];po—loc;[W] | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr
  )+
```

The set of edges E is a subset of program order and thus cannot create cycles by itself. So it can only create cycles in composition with other edges. In particular, the subset of edges of E ending with [DMB.ST] can only create cycles in composition with other edges in the (bigger) relation starting with a `dmb st`. (By type E cannot be composed with itself.)

Hence replace the subset of E ending with [DMB.ST] with its post-composition with other edges from the bigger relation starting with a `dmb st`.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
        (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
          ([R];ctrl;[DMB.ST];(po;[DMB.SY]|po;[W]|po;[Rel]) | [R];ctrl;[ISB] |
           [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] |
           [R];addr;po;[W] | [Acq];po;[W] | [R];po—loc;[W] | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr
  )+
```

The definition of E contains some duplication, so simplify.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
```

```
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
      (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
        ([R];ctrl;[DMB.ST];po;[DMB.SY] | [R];ctrl;[ISB] | [R];addr;po;[ISB] |
          [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] | [R];addr;po;[W] |
          [Acq];po;[W] | [R];po—loc;[W] | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr)+
```

Now the subset of E ending with DMB.SY is subsumed by the edges po;[DMB.SY], so can delete it. Also rewrite [R];po-loc;[W] to fri (by Internal axiom).

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
      (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*; ([R];ctrl;[ISB] |
        [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] |
        [R];addr;po;[W] | [Acq];po;[W] | fri | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr)+
```

Split E by definition of the '?' operator.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
```

```
| [Rel];po;[Acq]
| [Xw];rfi;[Acq]
| rmw
| addr
| data
| ctrl;[W]
| addr;po;[W]
| (addr|data);rfi
| [R];(addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi;[R] |
      (addr|data);rfi;[R];po—R—loc;[R])*;
        ([R];ctrl;[ISB] | [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] |
        [R];ctrl;[W] | [R];addr;po;[W] | [Acq];po;[W] | fri |
        [R];rmw;[W]) (* E1 *)
| [R];po—R—loc;[R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
      (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
        ([R];ctrl;[ISB] | [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] |
         [R];ctrl;[W] | [R];addr;po;[W] | [Acq];po;[W] | fri |
         [R];rmw;[W]) (* E2 *)
| co
| rfe
| fr
)+
```

Now consider $(W, R') \in$ rfi;[R];po-R-loc;[R]. Then there exists a read R such that $(W, R) \in$ rfi and $(R, R') \in$ po-R-loc. Let $(W', R') \in$ rf. Now there are two cases $W = W'$ or $W \neq W'$.

$W = W'$ Then $(W, R') \in$ rfi.

$W \neq W'$ W and W' must be coherence-related. Assume $(W', W) \in co$. Then $(R', W) \in$ fr and there is a cycle in po-loc;fr. So $(W, W') \in$ co. It cannot be $(W', R) \in$ po because otherwise cycle in po-loc;fr, and it cannot be $(R', W') \in$ po because otherwise cycle in po-loc;rf. By definition of po-R-loc, W' not po-between R and R'. So W' and R' from different threads and it is $(W', R') \in$ rfe. Therefore $(W, W'); (W', R') \in$ co;rfe, so $(W, R') \in$ co;rfe.

So rfi;[R];po-R-loc;[R] included in rfi | (co;rfe) & po-loc. Use this to strengthen E1 and E2.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | [R];(addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
        (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
        addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
  | [R];po—R—loc;[R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
      (addr|data);rfi | (addr|data);((co;rfe) & po—loc))*; [R]; (ctrl;[ISB] |
      addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] | addr;po;[W] |
      [Acq];po;[W] | fri | rmw;[W])
  | co
  | rfe
```

```
  | fr
  )+
```

Simplify: E1 is subsumed by the combination of the following edge sets:

- addr;po;[ISB],
- addr;po;[W],
- [Acq];po;[R],
- ctrl;[ISB],
- [Acq];po;[W],
- ctrl;[W],
- addr;[W],
- data;[W],
- fr,
- rmw,
- (addr|data);rfi,
- co,
- rfe,
- rmw.

So can drop E1.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | [R];po—R—loc;[R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
       (addr|data);rfi | (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
  | co
  | rfe
  | fr
  )+
```

The edge set E2 cannot create cycles by itself since it is a subset of program order (which in turn is acyclic). Any cycle contained in the order above that has a cycle using an edge from E2 must be one that uses it in composition with more edges from the relation. E2 does not compose with itself. So it suffices to replace E2 by the post-composition of all other edges with this one.

```
...
only has a cycle if the following has a cycle
```

```
([DMB.SY];po
| [DMB.SY];po;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1*)
| po;[DMB.SY]
| [F];po;[W]
| [R];po;[DMB.LD]
| [W];po;[DMB.ST]
| [R];addr;po;[ISB]
| [ISB|DMB.LD];po;[R]
| [ISB|DMB.LD];po;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
| ctrl;[F]
| po;[Rel]
| [Acq];po;[R|W]
| [Acq];po;[R|W];[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E3 *)
| [Rel];po;[Acq]
| [Rel];po;[Acq];[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E4 *)
| [Xw];rfi;[Acq]
| [Xw];rfi;[Acq];[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E5 *)
| rmw
| addr
| addr;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E6 *)
| data
| ctrl;[W]
| addr;po;[W]
| (addr|data);rfi
| (addr|data);rfi;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
| co
| rfe
| rfe;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
| fr)+
```

Some of these are easily subsumed by existing edges:

- E1 by [DMB.SY];po
- E2 by [F];po;[W] and [ISB|DMB.LD];po;[R] and ctrl;[ISB] | addr;po;[ISB]
- E3 by [Acq];po;[W] and [Acq];po;[R] and ctrl;[ISB] | addr;po;[ISB]

- E4 by the sets [Rel];po;[Acq] and [Acq];po;[W] and [Acq];po;[R] and ctrl;[ISB] and addr;po;[ISB].
- E5 by the sets [Xw];rfi;[Acq] and [Acq];po;[W] and [Acq];po;[R] and ctrl;[ISB] and addr;po;[ISB].
- E6 is subsumed by addr;po;[ISB] and addr;po;[W].

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr|data);rfi;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
  | co
  | rfe
  | rfe;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | fr
  )+
```

As shown before, rfi;[R];po-R-loc;[R] included in (rfi | ((co;rfe)&po-loc)). Use this to strengthen E1.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr|data);(rfi | ((co;rfe) & po—loc));[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | co
  | rfe
```

```
  | rfe;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
        (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
  | fr)+
```

Now consider $(W, R') \in$ rfe;[R];po-R-loc;[R]. Then $(W, R) \in$ rfe for some read R and $(R, R') \in$ po-R-loc. Now there are two cases: $(W, R') \in$ rf or otherwise.

$(W, R') \in$ rf Then $(W, R') \in$ rfe.

otherwise Then there is a write W' with $(W', R') \in$ rf and $W \neq W'$. W and W' must be coherence-related. Assume the coherence is $(W', W) \in$ co. Then it is $(R', W) \in$ fr and there is a cycle in fr;rf;po-loc. So the coherence must be $(W, W') \in$ co. It cannot be $(W', R) \in$ po because then there would be a cycle in po-loc;fr. By definition of po-R-loc, W' not po-between R and R'. And it cannot be $(R', W') \in$ po since then there would be a cycle in po-loc;rf. So W' not from the same thread as R and R'. But then it is $(W, R); (R, W'); (W', R') \in$ rfe;fre;rfe.

So rfe;[R];po-R-loc;[R] included in (rfe | rfe;fre;rfe). Use this to strengthen E2.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr|data);(rfi | ((co;rfe) & po—loc));[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
        (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | co
  | rfe
  | (rfe | rfe;fre;rfe);
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
        (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | fr
  )+
```

Split the first and second long edge.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
```

```
| [ISB|DMB.LD];po;[R]
| ctrl;[F]
| po;[Rel]
| [Acq];po;[R|W]
| [Rel];po;[Acq]
| [Xw];rfi;[Acq]
| rmw
| addr
| data
| ctrl;[W]
| addr;po;[W]
| (addr|data);rfi
| (addr|data);rfi;[R];
    (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
     (addr|data);((co;rfe) & po—loc))*; [R];
       (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
        addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
| (addr|data);((co;rfe) & po—loc);[R];
    (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
     (addr|data);((co;rfe) & po—loc))*; [R];
       (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
        addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
| co
| rfe
| rfe;
    (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
     (addr|data);((co;rfe) & po—loc))*; [R];
       (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
        addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E3 *)
| rfe;fre;rfe;
    (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
     (addr|data);((co;rfe) & po—loc))*; [R];
       (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
        addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E4 *)
| fr
)+
```

Now check E1 – E4, each with a different prefix.

- E1 starting with (addr|data);rfi. Since already have (addr|data);rfi in the relation can strengthen the order by dropping this prefix.
- E2 starts with (addr|data);((co;rfe)&po-loc). Since have the edges addr, data, co, rfe, can delete this prefix and strengthen the order.
- E3 starts with rfe. Since have rfe in the relation, can strengthen the order by deleting this prefix.
- E4 starts with rfe;fre;rfe. Since rfe and fr are already in the relation, can strengthen it by deleting this prefix.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
```

```
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
        (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
        (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
  | co
  | rfe
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
        (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E3 *)
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
        (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E4 *)
  | fr
  )+
```

Now have four copies of the same edge, can delete all but one.

```
  ...
  = ([DMB.SY];po
    | po;[DMB.SY]
    | [F];po;[W]
    | [R];po;[DMB.LD]
    | [W];po;[DMB.ST]
    | [R];addr;po;[ISB]
    | [ISB|DMB.LD];po;[R]
    | ctrl;[F]
    | po;[Rel]
    | [Acq];po;[R|W]
    | [Rel];po;[Acq]
    | [Xw];rfi;[Acq]
    | rmw
    | addr
    | data
    | ctrl;[W]
    | addr;po;[W]
    | (addr|data);rfi
    | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
          (addr|data);((co;rfe) & po—loc))*; [R];
            (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
              addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E *)
    | co
    | rfe
    | fr
    )+
```

E is subsumed by the combination of other edges already in the relation:

- addr;po;[ISB]
- [Acq];po;[R]
- ctrl;[ISB]
- (addr|data);rfi
- addr
- data
- co

- rfe
- addr;po;[W]
- [Acq];po;[W]
- ctrl;[W]
- addr;[W]
- data;[W]
- fri
- rmw.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | co
  | rfe
  | fr
  )+
```

Split co and fr.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | coi
  | rfe
  | fre
  | fri
  )+
```

coi is acyclic itself. It can only contribute to cycles in composition with other edges. Post-compose

every edge EDGE in the relation with coi and add EDGE;coi.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | [DMB.SY];po;coi
  | po;[DMB.SY]
  | [F];po;[W]
  | [F];po;[W];coi
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Acq];po;[R|W];coi
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | rmw;coi
  | addr
  | addr;coi
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | addr;po;[W];coi
  | (addr|data);rfi
  | coe
  | coe;coi
  | rfe
  | fre
  | fre;coi
  | fri
  | fri;coi
  )+
```

Most of those edges are subsumed by others:

- [DMB.SY];po;coi subsumed by [DMB.SY];po,
- [F];po;[W];coi by [F];po;[W],
- [Acq];po;[W];coi by [Acq];po;[W],
- rmw;coi subsumed by fri,
- addr;coi subsumed by addr;po;[W],
- addr;po;[W];coi subsumed by addr;po;[W],
- coe;coi subsumed by coe,
- fre;coi subsumed by fre,
- fri;coi subsumed by fri.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
```

```
| po;[Rel];coi
| [Acq];po;[R|W]
| [Rel];po;[Acq]
| [Xw];rfi;[Acq]
| rmw
| addr
| data
| data;coi
| ctrl;[W]
| ctrl;[W];coi
| addr;po;[W]
| (addr|data);rfi
| coe
| rfe
| fre
| fri
)+
```

fri is acyclic itself, so can only create cycles in composition with other edges. Post-compose all edges EDGE with fri and add EDGE;fri.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | [DMB.SY];po;fri
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | [ISB|DMB.LD];po;[R];fri
  | ctrl;[F]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Acq];po;[R|W];fri
  | [Rel];po;[Acq]
  | [Rel];po;[Acq];fri
  | [Xw];rfi;[Acq]
  | [Xw];rfi;[Acq];fri
  | rmw
  | addr
  | addr;fri
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | (addr|data);rfi;fri
  | coe
  | rfe
  | rfe;fri
  | fre
  )+
```

But these edges are subsumed by others in the relation.

- [DMB.SY];po;fri by [DMB.SY];po,
- [ISB|DMB.LD];po;[R];fri by [F];po;[W],
- [Acq];po;[R|W];fri by [Acq];po;[R|W],
- [Rel];po;[Acq];fri by [Rel];po;[Acq] and [Acq];po;[R|W],
- [Xw];rfi;[Acq];fri by [Xw];rfi;[Acq] and [Acq];po;[R|W],

- addr;fri by addr;po;[W],
- (addr|data);rfi;fri by addr;po;[W] and data;coi,
- rfe;fri by coe.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Simplify.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [DMB.ST];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[DMB.ST]
  | ctrl;[ISB]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

The edges [DMB.ST];po;[W] themselves are acyclic. Replace them by adding the postcomposition of all edges with them.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [W];po;[DMB.ST];po;[W]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[DMB.ST]
  | ctrl;[DMB.ST];po;[W]
  | ctrl;[ISB]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

The edges [W];po;[DMB.ST] and ctrl;[DMB.ST] themselves are acyclic. Replace them by adding the precomposition of all edges with them.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST];po;[W]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[DMB.ST];po;[W]
  | ctrl;[ISB]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

The ctrl;[DMB.ST];po;[W] edge is subsumed by ctrl;[W].

```
...
= ([DMB.SY];po
```

```
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST];po;[W]
  | [R];addr;po;[ISB]
  | [ISB];po;[R]
  | [DMB.LD];po;[R]
  | ctrl;[ISB]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Merge two ISB edge sets.

```
...
=
  ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST];po;[W]
  | [R];(ctrl|(addr;po));[ISB]
  | [ISB];po;[R]
  | [DMB.LD];po;[R]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

The sets [ISB];po;[W] and [ISB];po;[R] themselves are acyclic. Replace them by the postcomposition of all other edges with them.

```
...
only has a cycle if the following has one
  ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST];po;[W]
  | [R];(ctrl|(addr;po));[ISB]
```

```
 | [R];(ctrl|(addr;po));[ISB];po;[R|W];
 | [DMB.LD];po;[R]
 | po;[Rel]
 | po;[Rel];coi
 | [Acq];po;[R|W]
 | [Rel];po;[Acq]
 | [Xw];rfi;[Acq]
 | rmw
 | addr
 | data
 | data;coi
 | ctrl;[W]
 | ctrl;[W];coi
 | addr;po;[W]
 | (addr|data);rfi
 | coe
 | rfe
 | fre
 )+
```

The [R];(ctrl|(addr;po));[ISB] edge set itself is acyclic. Replace it by all possible precompositions of other edges with it.

```
...
only has a cycle if the following has one
  ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD];po;[W|R]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST];po;[W]
  | [R];(ctrl|(addr;po));[ISB];po;[W|R]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Similarly, replace [DMB.LD];po;[W|R] with the postcomposition of all edges with it.

```
...
only has a cycle if the following has one
  ([DMB.SY];po
  | po;[DMB.SY]
  | [R];po;[DMB.LD]
  | [R];po;[DMB.LD];po;[W|R]
  | [W];po;[DMB.ST];po;[W]
  | [R];(ctrl|(addr;po));[ISB];po;[W|R]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
```

```
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Similarly, replace [R];po;[DMB.LD] with its precomposition with all other edges.

```
...
only has a cycle if the following has one
  ([DMB.SY];po
  | po;[DMB.SY]
  | [R];po;[DMB.LD];po;[W|R]
  | [W];po;[DMB.ST];po;[W]
  | [R];(ctrl|(addr;po));[ISB];po;[W|R]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Replace [DMB.SY];po by the postcomposition of all other edges with it.

```
...
only has a cycle if the following has one
  ( po;[DMB.SY]
  | po;[DMB.SY];po
  | [R];po;[DMB.LD];po;[W|R]
  | [W];po;[DMB.ST];po;[W]
  | [R];(ctrl|(addr;po));[ISB];po;[W|R]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Replace po;[DMB.SY] with its precomposition with all other edges.

```
...
```

```
only has a cycle if the following has one
  (po;[DMB.SY];po
  | [R];po;[DMB.LD];po;[W|R]
  | [W];po;[DMB.ST];po;[W]
  | [R];(ctrl|(addr;po));[ISB];po;[W|R]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Rearrange.

```
...
= (rfe | fre | coe
  | addr | data
  | ctrl; [W]
  | (ctrl| (addr; po)); [ISB]; po; [W|R]
  | addr; po; [W]
  | (ctrl | data); coi
  | (addr | data); rfi
  | rmw
  | [Xw];rfi;[Acq]
  | po;[DMB.SY];po
  | [Rel];po;[Acq]
  | [R];po;[DMB.LD];po;[W|R]
  | [Acq];po;[R|W]
  | [W];po;[DMB.ST];po;[W]
  | po;[Rel]
  | po;[Rel];coi
  )+
```

(ctrl|(addr;po));[ISB];po;[W] is subsumed by ctrl;[W], addr;po;[W]

```
...
= (rfe | fre | coe
  | addr | data
  | ctrl; [W]
  | (ctrl| (addr; po)); [ISB]; po; [R]
  | addr; po; [W]
  | (ctrl | data); coi
  | (addr | data); rfi
  | rmw
  | [Xw];rfi;[Acq]
  | po;[DMB.SY];po
  | [Rel];po;[Acq]
  | [R];po;[DMB.LD];po;[W|R]
  | [Acq];po;[R|W]
  | [W];po;[DMB.ST];po;[W]
  | po;[Rel]
  | po;[Rel];coi
  )+
```

Apply Xw = range(rmw). And strengthen the DMB.LD and [Acq];po[R|W] edges.

```
...
⊆ (rfe | fre | coe
   | addr | data
   | ctrl; [W]
   | (ctrl | (addr; po)); [ISB]; po; [R]
   | addr; po; [W]
   | (ctrl | data); coi
   | (addr | data); rfi
   | rmw
   | [range(rmw)];rfi;[Acq]
   | po;[DMB.SY];po
   | [Rel];po;[Acq]
   | [R];po;[DMB.LD];po
   | [Acq];po
   | [W];po;[DMB.ST];po;[W]
   | po;[Rel]
   | po;[Rel];coi
   )+
```

Finally, the relation above is the same as ARMv8-axiomatic's ob (dropping "Q") with the definitions of obs, dob, aob, and bob inlined, and the transitive closure replacing the recursive definition.

□

Appendix E

RISC-V

E.1 RISC-V operationally, continued

Atomic memory operations Atomic memory operations cannot fail, and do not have a register write indicating success. In order to implement RISC-V's stronger atomicity of atomic memory operations — corresponding to the axiomatic model describing the behaviour of an atomic memory operation in terms of a single event — the operational model implements the instruction with a global lock. The steps in its execution are the following:[1]

1. fetch the instruction
2. do the register reads necessary to determine the address
3. announce the address
4. do the register reads necessary for the instruction's arithmetic/data
5. check whether the instruction fulfils the conditions for doing both the "load part" and the "store part" of the instruction (including the checks of the read-request-condition, the store-commit condition, and the write-propagate condition)
6. if so, take a global lock in the operational model that stops any instruction except the current one from executing
7. perform the atomic operation's reads
8. complete the load part of the instruction
9. perform the arithmetic based on the returned value
10. perform the atomic operation's writes
11. complete the store part of the instruction
12. release the global lock
13. write the return value of the atomic operation's read to the particular register
14. finish the instruction

The instruction holding a global lock from the point of performing its reads until having done its writes ensures the strong atomicity captured in the axiomatic model's by expressing the atomic operation's behaviour using a single event in the memory ordering.

Additional barriers The additional barriers that RISC-V has compared to ARMv8 mostly work analogously to the existing ones: for example, a $\texttt{fence}_{\texttt{W,R}}$ can commit only when all program-order-preceding store instructions are finished, and any load can only satisfy when every program-order-preceding $\texttt{fence}_{\texttt{W,R}}$ is finished (and hence committed).

[1] As before: the work on RISC-V concurrency is joint work with Shaked Flur, Peter Sewell, Luc Maranget, Susmit Sarkar, and the Memory Model Task Group, chaired by Daniel Lustig; the adapted Flat RISC-V operational model is principally due to Shaked Flur and the author; the text description of the Flat RISC-V model was published as Appendix B.3 of the RISC-V ISA manual [121].

However, for RISC-V the model has to be locally relaxed compared to the ARMv8 Flat model to take care of two aspects. (1) In ARMv8, barriers wait for the control flow to be determined; as described before, in ARMv8 this additional "local ordering" is not observable; in RISC-V the additional barriers would make this observable and create stronger ordering than architecturally intended. (2) The status register write of store exclusives creates dependency ordering originating from *stores* (as opposed to previously only loads), and the model has to ensure that this does not create unintentional ordering with respect to fences.

(1) Consider the case of a $\texttt{fence}_{\texttt{W,R}}$. The architecture's intention is for this fence to only create ordering from stores to loads. If, however, the fence, like in the ARMv8 case, also waited for the control flow to be determined, this would create observable ordering between loads affecting the control flow before the fence and program-order-succeeding loads. In the left example of Figure E.1, for instance, d would then create ordering between c and e.

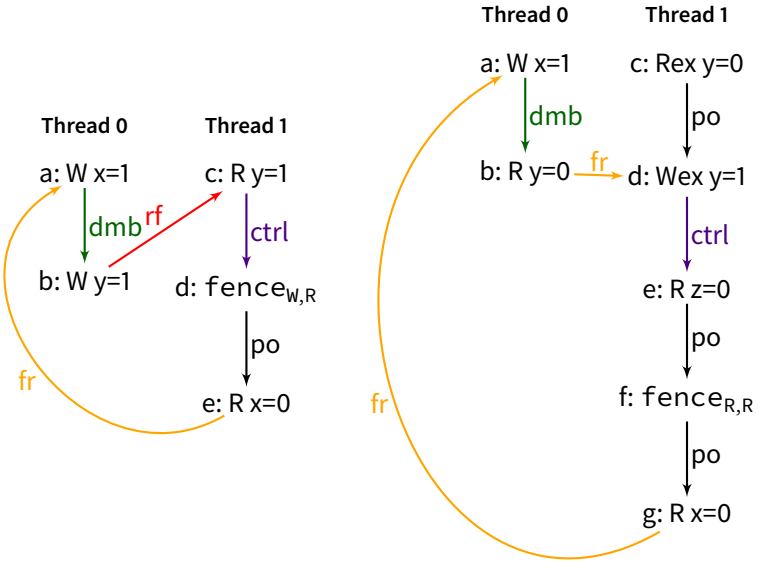


Figure E.1

But in this variant of the MP litmus test, RISC-V allows *e* to read from memory before *c* is satisfied, and thus allow the behaviour where *e* reads from the initial write, coherence-before *a*. To allow for this behaviour, the operational model allows "speculatively executing" the fence *d*: allow it to commit and finish even if the control flow in its program-order prefix is not determined yet. Therefore in the RISC-V Flat model, the barrier commitment and finishing conditions do not require the control flow to be determined, and when a conditional/computed branch finishes and discards any untaken branches, these branches may include already-finished fence instructions.

(2) Similarly, the RISC-V Flat model has to ensure that dependencies out of the status register write of store exclusive instructions do not create unintended ordering with respect to barriers. Consider the variant of the SB litmus test on the right of Figure E.1, for instance. Here the write exclusive *d* becomes coherence-after the write *b* reads from; the status register write of *d*'s

store exclusive determines the condition of some branch program-order-before e; the load of e is followed by a $\texttt{fence}_{R,R}$ f; and the program-order-succeeding read g reads from a write coherence-before a.

This behaviour should be allowed despite the fence f: f is only supposed to order reads. With the relaxation described in the previous paragraph, f is allowed to be committed and finished speculatively. If, however, as in the case of the ARMv8 Flat model, g was only allowed to satisfy when f was finished and hence e finished, then the behaviour would not be allowed by the model: e can only finish when its control flow is determined, and thus when d has written the status register and is propagated to memory. Therefore, in RISC-V, certain conditions concerning fences are relaxed compared to ARMv8. Here, while f can still only finish when e is finished, the condition for satisfying g no longer checks for f to be finished; in the case of $\texttt{fence}_{R,R}$ in order for g to satisfy, it is sufficient for all loads preceding the fence to have their reads *satisfied*.[2]

E.2 The RISC-V Flat model

The RISC-V operational model closely follows the ARMv8 Flat model for RISC-V, but is adapted for the semantic differences and additional barriers described earlier. Moreover, it includes RISC-V's AMO instructions (atomic memory operations) that the ARMv8 model currently does not cover.

Terminology: The term "acquire" refers to an instruction or its memory event/request with the acquire-RCpc or acquire-RCsc annotation. The term "release" refers to an instruction or its memory event/request with the release-RCpc or release-RCsc annotation. The version of the RISC-V model as presented here differs from that found in the RISC-V ISA manual in some terminology. RISC-V calls read and write events/requests "load" and "store operations". For uniformity, this chapter uses the former and also renames some related concepts to match the rest of this thesis. The text sometimes refers to "load instructions" and "store instructions"; both these are meant to include AMO instructions.

E.3 Limitations

- The model covers user-level RV64I and RV64A. In particular, it does not support the misaligned atomics extension "Zam" or the total store ordering extension "Ztso". (It should be easy to adapt the model to RV32I/A and to the G, Q and C extensions. This should involve, mostly, writing Sail code for the instructions, with minimal, if any, changes to the concurrency model.)
- The model covers only normal memory accesses (it does not handle I/O accesses).
- The model does not cover TLB-related effects.
- The model assumes the instruction memory is fixed. In particular, the Fetch instruction transition does not generate memory reads, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when

[2] For this test, alternatively the model could also be relaxed to allow loads to finish "speculatively", before conditional/computed branches are determined.

given a memory location.
- The model does not cover exceptions, traps and interrupts.

Model states A model state consists of a shared memory and a tuple of thread states. The shared memory state records all the memory writes that have propagated so far, in the order they propagated. Each thread state consists principally of a tree of instruction instances, some of which have been *finished*, and some of which have not. Non-finished instruction instances can be subject to *restart*, e.g. if they depend on an out-of-order or speculative load that turns out to be unsound. Conditional branch and indirect jump instructions may have multiple successors in the instruction tree. When such an instruction is finished, any untaken alternative paths in the instruction tree are discarded. Each instruction instance in the instruction tree has a state that includes a pseudocode execution state in the form of a Sail outcome value. As in the ARMv8 Flat model, an instruction instance state also includes information about the instance's memory and register footprints, its register reads and writes, its memory reads and writes, whether it is finished, etc.

Model transitions The model transitions of the RISC-V Flat model mostly closely follow that of the ARMv8 RISC-V model. The transitions are introduced below and defined in Section E.3.5, with a precondition and a construction of the post-transition model state for each transition.

Note that in the following, store exclusive instructions that have already been determined to fail are treated like "plain" non-memory instructions.

Transitions for all instructions

- Fetch instruction: This transition represents a fetch and decode of a new instruction instance, as a program order successor of a previously fetched instruction instance (or the initial fetch address). The model assumes the instruction memory is fixed; it does not describe the behaviour of self-modifying code. In particular, the Fetch instruction transition does not generate memory reads, and the shared memory is not involved in the transition.
  - Register write: This is a write to a register.
  - Register read: This is a read of a register value from the most recent program-order-predecessor instruction instances that write that part of the register.
  - Pseudocode internal step: This covers pseudocode internal computation: arithmetic, function calls, etc.
  - Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted, memory accesses cannot be discarded, and all memory effects have taken place. For conditional branch and indirect jump instructions, any program order successors that were fetched from an address that is not the one that was written to the *pc* register are discarded, together with the sub-tree of instruction instances below them.

Load instructions

  - Initiate memory read of load instruction: At this point the memory footprint of the load instruction is provisionally known (it could change if earlier instructions are restarted)

and its individual reads can start being satisfied.

- Satisfy memory read by forwarding from writes: This partially or entirely satisfies a single memory read by forwarding, from program-order-previous memory writes.
- Satisfy memory read from memory: This entirely satisfies the outstanding slices of a read, from memory.
- ◦ Complete load instruction: At this point all reads of the instruction have been entirely satisfied and the instruction pseudocode can continue executing. A load instruction can be subject to being restarted until the Finish instruction transition. But, under some conditions, the model might treat a load instruction as non-restartable even before it is finished, e.g. when all the instructions po-before the load instruction are finished.

Store instructions

- ◦ Initiate memory writes of store instruction: At this point the memory footprint of the store is provisionally known.
- ◦ Instantiate memory write values of store instruction: At this point the memory writes have their values and program-order-successor memory reads can be satisfied by forwarding from them.
- ◦ Commit store instruction: At this point the memory writes are guaranteed to happen (the instruction can no longer be restarted or discarded), and they can start being propagated to memory.
- Propagate memory write: This propagates a single write to memory.
- ◦ Complete store instruction: At this point all the writes of the instruction have been propagated to memory, and the instruction pseudocode can continue executing.

Store conditional (sc) instructions

- Early sc fail: This causes the `sc` to fail, either spontaneously or because it is not paired with a program-order-previous load reserve (`lr`).
- Paired sc: This transition indicates the `sc` is paired with an `lr` and might succeed.
- Commit and propagate write of an sc: This is an atomic execution of the transitions Commit store instruction and Propagate memory write, it is enabled only if the writes from which the `lr` read from have not been overwritten.
- Late sc fail: This causes the `sc` to fail, either spontaneously or because the writes from which the `lr` read have been overwritten.

AMO instructions

- Satisfy read, commit and propagate write of an AMO: This is an atomic execution of all the transitions needed to satisfy the memory read, do the required arithmetic, and commit and propagate the memory write of the AMO instruction.

Fence instructions

- ◦ Commit fence

As in the case of the ARMv8 Flat model, the transitions labelled ◦ can always be taken eagerly, as soon as their precondition is satisfied, without excluding other behaviour; those with • cannot.

Although Fetch instruction is marked with as a non-eager transition, it can be taken eagerly as long as it is not taken infinitely many times.

The following describes the formal operational model.

E.3.1 Intra-instruction Pseudocode Execution

The Sail outcomes used in the RISC-V Flat model are the following:

Load_mem (*kind*, *address*, *size*, *read_continuation*)	Memory read
Continue_sc_or_fail (*res_continuation*)	Allow `sc` to fail early
Write_ea (*kind*, *address*, *size*, *next_state*)	Write effective address
Write_memv (*mem_value*, *write_continuation*)	Write value
Fence (*kind*, *next_state*)	Fence
Read_reg (*reg_name*, *read_continuation*)	Register read
Write_reg (*reg_name*, *reg_value*, *next_state*)	Register write
Internal (*next_state*)	Pseudocode internal step
Done ( )	End of pseudocode

Here:

- for a load/store, *kind* identifies whether it is `lr/sc`, acquire-RCpc/release-RCpc, acquire-RCsc/release-RCsc, acquire-release-RCsc; and
- for a fence, *kind* identifies whether it is a "normal" fence or a `fence.tso`, and (for normal fences) the predecessor and successor ordering bits.

E.3.2 Instruction Instance State

Each instruction instance *i* has a state comprising:

- program_loc, the address from which the instruction was fetched;
- instruction_kind, identifying whether this is a load, store, AMO, fence, branch/jump or a 'simple' instruction (this also includes a *kind* similar to that in the pseudocode execution states);
- regs_in, the set of source *reg_name*s, as statically determined from the pseudocode of the instruction;
- regs_out, the destination *reg_name*s, as statically determined from the pseudocode of the instruction;
- pseudocode_state (or sometimes just 'state' for short), one of:

Plain (*pseudocode_state*)	ready to make a pseudocode transition
Pending_mem_reads (*read_continuation*)	requesting a memory read
Pending_mem_writes (*write_continuation*)	requesting a memory write

- *reg_reads*, the register reads the instance has performed, including, for each one, the register write slices it read from;
- *reg_writes*, the register writes the instance has performed;

- *mem_reads*, a set of memory reads, and for each one the as-yet-unsatisfied slices (the byte indices that have not been satisfied yet), and for the satisfied slices the write slices that satisfied it (each consisting of a memory write and subset of its byte indices);
- *mem_writes*, a set of memory writes, and for each a flag that indicates whether it has been propagated (passed to the shared memory) or not; and
- information recording whether the instance is committed, finished, etc.

Each memory read includes a memory footprint (address and size); each memory write includes a memory footprint, and, when available, a value. A load instruction instance with a non-empty *mem_reads*, for which all memory reads are satisfied (i.e. there are no unsatisfied read slices) is said to be *entirely satisfied*.

Informally, an instruction instance is said to have *fully determined data* if the load and `sc` instructions feeding into its source registers are finished. Similarly, it is said to have a *fully determined memory footprint* if the load and `sc` instructions feeding into its memory address register are finished. Formally, first define the notion of a *fully determined register write*: a register write w from *reg_writes* of an instruction instance i is said to be *fully determined* if one of the following conditions hold:

1. i is finished; or
2. the value written by w is not affected by a read or write that i has made (i.e. a value loaded from memory or the result of `sc`), and for every register read that i has mades that affects w the register write from which i read is fully determined (or i read from the initial register state).

Now, an instruction instance i is said to have *fully determined data* if for every register read r from *reg_reads*, the register writes that r reads from are fully determined. An instruction instance i is said to have a *fully determined memory footprint* if it has done enough instruction steps to compute its footprint and for every register read r from *reg_reads* that feeds into its footprint the register writes that r read from are fully determined.

The rmem model approximates this by recording, for every register write, the set of register writes from other instructions that have been read by this instruction at the point of performing the write. By arranging the pseudocode of the instructions currently covered by the tool in the maximally liberal way, this approximation is exactly the set of register writes on which the write depends.

E.3.3 Thread State

The model state of a single thread comprises:

- thread_id, a unique identifier of the thread;
- register_data, the name, bit width, and start bit index for each registers;
- initial_register_state, the initial register value for each register;
- initial_fetch_address, the initial instruction fetch address for this thread;
- instruction_tree, a tree of the instruction instances that have been fetched (and not discarded), in program order.

E.3.4 Shared Memory State

The model state of the shared memory comprises a list of memory writes, in the order they propagated to the shared memory. When a write is propagated to the shared memory it is added to the end of the list. When a read is satisfied from memory, for each byte of the read, the most recent corresponding write slice is returned.

For most purposes, it is simpler to think of the shared memory as an array, i.e., a map from memory locations to write slices, where each memory location is mapped to a one-byte slice of the most recent memory write to that location. However, this abstraction is not detailed enough to properly handle the `sc` instruction. The RISC-V concurrency model allows writes from the same thread as the `sc` to intervene between the write of the `sc` and the writes the paired `lr` read from. To allow such writes to intervene and forbid others, the array abstraction must be extended to record more information. The model uses a list for simplicity, but a more efficient and scalable model could implement this differently.

E.3.5 Transitions

Each of the paragraphs below describes a single kind of model transition. The description starts with a condition over the current system state. The transition can be taken in the current state only if the condition is satisfied. The condition is followed by an action that is applied to that state when the transition is taken, in order to generate the new system state.

Fetch instruction A possible program-order successor of instruction instance i can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of i in the thread's *instruction_tree* are from *loc*; and
2. if i's pseudocode has already written an address to *pc*, then *loc* must be that address, otherwise *loc* is:
   - for a conditional branch, the successor address or the branch target address;
   - for a (direct) jump and link instruction (`jal`), the target address;
   - for an indirect jump instruction (`jalr`), any address; and
   - for any other instruction, $i.program\_loc + 4$.

Action: construct a freshly initialised instruction instance i' for the instruction in the program memory at *loc*, with state Plain (*isa_state*), computed from the instruction pseudocode, including the static information available from the pseudocode such as its instruction_kind, regs_in, and regs_out, and add i' to the thread's instruction_tree as a successor of i.

The possible next fetch addresses (*loc*) are available immediately after fetching i and the model does not need to wait for the pseudocode to write to *pc*; this allows out-of-order execution and speculation past conditional branches and jumps. For most instructions these addresses are easily obtained from the instruction pseudocode. The only exception to that is the indirect jump instruction (`jalr`), where the address depends on the value held in a register. In principle the mathematical model should allow speculation to arbitrary addresses here; the executable model

in rmem approximates this.

Initiate memory read of load instruction An instruction instance i in state Plain (Read_mem (*kind*, *address*, *size*, *read_continuation*)) can always initiate the corresponding memory reads. Action:

1. Construct the appropriate memory reads *mrs*:
   - if *address* is aligned to *size* then *mrs* is a single memory read of *size* bytes from *address*;
   - otherwise, *mrs* is a set of *size* memory reads, each of one byte, from the addresses $address \ldots address + size - 1$.
2. set *mem_reads* of i to *mrs*; and
3. update the state of i to Pending_mem_reads (*read_continuation*).

The RISC-V ISA manual specifies that misaligned memory accesses may be decomposed at any granularity. The model decomposes them to one-byte accesses as this granularity subsumes all others.

Satisfy memory read by forwarding from writes For a non-AMO load instruction instance i in state Pending_mem_reads (*read_continuation*), and a memory read *mr* in i.*mem_reads* that has unsatisfied slices, the memory read can be partially or entirely satisfied by forwarding from unpropagated memory writes by store instruction instances that are program-order-before i if:

1. all program-order-previous `fence` instructions with `.sr` and `.pw` set are finished;
2. for every program-order-previous `fence` instruction, f, with `.sr` and `.pr` set, and `.pw` not set, if f is not finished then all load instructions that are program-order-before f are entirely satisfied;
3. for every program-order-previous `fence.tso` instruction, f, that is not finished, all load instructions that are program-order-before f are entirely satisfied;
4. if i is a load-acquire-RCsc, all program-order-previous store-release-RCsc instructions are finished;
5. if i is a load-acquire-release, all program-order-previous instructions are finished;
6. all non-finished program-order-previous load-acquire instructions are entirely satisfied; and
7. all program-order-previous store-acquire-release instructions are finished.

Let *mwss* be the set of all unpropagated memory write slices from non-`sc` store instruction instances that are program-order-before i and have already calculated the value to be written, that overlap with the unsatisfied slices of *mr*, and which are not superseded by intervening writes (with known address) or writes that are read from by any intervening loads. The last condition requires, for each memory write slice *mws* in *mwss* from instruction i':

- that there is no store instruction program-order-between i and i' with a memory write overlapping *mws*; and
- that there is no load instruction program-order-between i and i' that was satisfied from an overlapping memory write slice from a different thread.

Action:

1. update i.*mem_reads* to indicate that *mr* was satisfied by *mwss*; and
2. restart any speculative instructions which have violated coherence as a result of this, i.e., for

every non-finished instruction i' that is a program-order successor of i, and every memory read mr' of i' that was satisfied from $mwss'$, if there exists a memory write slice mws' in $mwss'$, and an overlapping memory write slice from a different memory write in $mwss$, and mws' is not from an instruction that is a program-order successor of i, restart i' and its *restart-dependents*.

Where, the *restart-dependents* of instruction j are:

- program-order successors of j that have data-flow dependency on a register write of j;
- program-order successors of j that have a memory read that reads from a memory write of j (by forwarding);
- if j is a load-acquire, all the program-order successors of j;
- if j is a load, for every `fence`, f, with `.sr` and `.pr` set, and `.pw` not set, that is program-order successor of j, all the load instructions that are program-order successors of f;
- if j is a load, for every `fence.tso`, f, that is program-order successor of j, all the load instructions that are program-order successors of f; and
- (recursively) all the restart-dependents of all the instruction instances above.

Forwarding memory writes to a memory load might satisfy only some slices of the load, leaving other slices unsatisfied.

A program-order-previous write that was not available when taking the transition above might make *mwss* provisionally unsound (violating coherence) when it becomes available. That store will prevent the load from being finished (see Finish instruction), and will cause it to restart when that write is propagated (see Propagate memory write).

A consequence of the transition condition above is that store-release-RCsc memory writes cannot be forwarded to load-acquire-RCsc instructions: *mwss* does not include memory writes from finished stores (as their writes must be propagated), and the condition above requires all program-order-previous store-release-RCsc instructions to be finished when the load is acquire-RCsc.

Satisfy memory read from memory For an instruction instance i of a non-AMO load instruction or an AMO instruction in the context of the Satisfy read, commit and propagate write of an AMO transition, any memory read mr in $i.mem\_reads$ that has unsatisfied slices, can be satisfied from memory if all the conditions of Satisfy memory read by forwarding from writes are satisfied. Action: let *mwss* be the most recent memory write slices from memory covering the unsatisfied slices of mr, and apply the action of Satisfy memory read by forwarding from writes.

Note that Satisfy memory read by forwarding from writes might leave some slices of the memory read unsatisfied; those will have to be satisfied by taking the transition again, or taking Satisfy memory read from memory. Satisfy memory read from memory, on the other hand, will always satisfy all the unsatisfied slices of the memory read.

Complete load instruction (when all its reads are entirely satisfied) A load instruction instance i in state Pending_mem_reads(*read_continuation*) can be completed (not to be confused with finished) if all the memory reads $i.mem\_reads$ are entirely satisfied (i.e. there are no unsatis-

fied slices). Action: update the state of i to Plain (*read_continuation mem_value*), where *mem_value* is assembled from all the memory write slices that satisfied i.*mem_reads*.

Early sc fail An `sc` instruction instance i in state Plain (Continue_sc_or_fail *res_continuation*) can always fail. Action: update the state of i to Plain (*res_continuation* false).

Paired sc An `sc` instruction instance i in state Plain (Continue_sc_or_fail *res_continuation*) can continue its (potentially successful) execution if i is *paired* with a load reserve instruction (`lr`). It is paired with a load reserve, if the load reserve, is the closest program-order-preceding load reserve or store conditional instruction. Action: update the state of i to Plain (*res_continuation* true).

Initiate memory writes of store instruction, with their footprints An instruction instance i in state Plain (Store_ea (*kind*, *address*, *size*, *next_state*)) can always announce its pending memory write footprint. Action:

1. construct the appropriate memory writes *mws* (without the store value):
   - if *address* is aligned to *size* then *mws* is a single memory write of *size* bytes to *address*;
   - otherwise, *mws* is a set of *size* memory writes, each of one-byte size, to the addresses $address \ldots address + size - 1$.
2. set i.*mem_writes* to *mws*; and
3. update the state of i to Plain (*next_state*).

Note that after taking the transition above the memory writes do not yet have their values. The importance of splitting this transition from the transition (below) for instantiating the writes with their values is that it allows other program-order successor store instructions to observe the memory footprint of this instruction, and if they don't overlap, propagate out of order as early as possible (i.e. before the data register value becomes available).

Instantiate memory write values of store instruction An instruction instance i in state Plain (Store_memv (*mem_value*, *write_continuation*)) can always instantiate the values of the memory writes i.*mem_writes*. Action:

1. split *mem_value* between the memory writes i.*mem_writes*; and
2. update the state of i to Pending_mem_writes (*write_continuation*).

Commit store instruction An uncommitted instruction instance i of a non-`sc` store instruction or an `sc` instruction in the context of the Commit and propagate write of an sc transition, in state Pending_mem_writes (*write_continuation*), can be committed (not to be confused with propagated) if:

1. i has fully determined data;
2. all program-order-previous conditional branch and indirect jump instructions are finished;
3. all program-order-previous `fence` instructions with `.sw` set are finished;
4. all program-order-previous `fence.tso` instructions are finished;
5. all program-order-previous load-acquire instructions are finished;
6. all program-order-previous store-acquire-release instructions are finished;

7. if i is a store-release, all program-order-previous instructions are finished;
8. all program-order-previous memory access instructions have a fully determined memory footprint;
9. all program-order-previous store instructions, except for failed `sc` instructions, have initiated and so have non-empty *mem_writes*; and
10. all program-order-previous load instructions have initiated and so have non-empty *mem_reads*.

Action: record that i is committed.

Notice that if condition 8. is satisfied the conditions 9. and 10. are also satisfied, or will be satisfied after taking some eager transitions, and therefore they do not strengthen the model. By requiring them, however, the model guarantees that previous memory access instructions have taken enough transitions to make their memory requests visible for the condition check of Propagate memory write (which is the next transition the instruction will take) making that condition simpler.

Propagate memory write For a committed instruction instance i in state Pending_mem_writes (*write_continuation*), and an unpropagated memory write mw in i.*mem_writes*, mw can be propagated if:

1. i is committed;
2. all memory writes of program-order-previous store instructions that overlap with mw have already propagated;
3. all memory reads of program-order-previous load instructions that overlap with mw have already been satisfied, and (the load instructions) are *non-restartable* (see definition below); and
4. all memory reads that were satisfied by forwarding mw are entirely satisfied.

Where a non-finished instruction instance j is *non-restartable* if:

1. there does not exist a store instruction s and an unpropagated memory write mw of s such that applying the action of the Propagate memory write transition to mw will result in the restart of j; and
2. there does not exist a non-finished load instruction l and a memory read mr of l such that applying the action of the Satisfy memory read by forwarding from writes/Satisfy memory read from memory transition (even if mr is already satisfied) to mr will result in the restart of j.

Action:

1. update the shared memory state with mw;
2. update i.*mem_writes* to indicate that mw was propagated; and
3. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' program-order-after i and every memory read mr' of i' that was satisfied from $mwss'$, if there exists a memory write slice mws' in $mwss'$ that overlaps with mw and is not from mw, and mws' is not from a program-order successor of i, restart i' and its *restart-dependents* (where restart-dependents are as previously defined for

Satisfy memory read by forwarding from writes).

Commit and propagate write of an sc An uncommitted `sc` instruction instance i in pseudocode state Pending_mem_writes (*write_continuation*), with a paired `lr` i' that has been satisfied by some write slices *mwss*, can be committed and propagated at the same time if:

1. i' is finished;
2. every memory write that has been forwarded to i' is propagated;
3. the conditions of Commit store instruction are satisfied;
4. the conditions of Propagate memory write are satisfied (note that an `sc` instruction can only have one memory write); and
5. for every write slice *mws* from *mwss*, *mws* has not been overwritten in the shared memory by a write from another thread at any point since *mws* was propagated.

Action:

1. apply the actions of Commit store instruction; and
2. apply the action of Propagate memory write.

Late sc fail An `sc` instruction instance i in state Pending_mem_writes (*write_continuation*) that has not propagated its memory write can always fail. Action:

1. clear i.*mem_writes*; and
2. update the state of i to Plain (*write_continuation* false).

Complete store instruction (when its writes are all propagated) A store instruction instance i in state Pending_mem_writes (*write_continuation*), for which all the memory writes in i.*mem_writes* have been propagated, can always be completed (not to be confused with finished). Action: update the state of i to Plain (*write_continuation* true).

Satisfy read, commit and propagate write of an AMO An AMO instruction instance i in state Pending_mem_reads (*read_continuation*) can perform its memory access if it is possible to perform the following sequence of transitions with no intervening transitions:

1. Satisfy memory read from memory
2. Complete load instruction
3. Complete load instruction
4. Pseudocode internal step (zero or more times)
5. Instantiate memory write values of store instruction
6. Commit store instruction
7. Propagate memory write
8. Complete store instruction

and if the condition for finishing the ("load part" of the) AMO holds in the state reached after these transitions (except without requiring the instruction to have state Plain (Done ( ))). Action: perform the above sequence of transitions, one after the other, with no intervening transitions.

Notice that the above definition does not allow forwarding from program-order-previous writes to the load of an AMO. In addition, the write of an AMO cannot be forwarded to a program-order successor read: before taking the transition above, the write of the AMO does not have its value

and therefore cannot be forwarded; after taking the transition above the write is propagated and therefore cannot be forwarded.

Commit fence A fence instruction instance i in state Plain (Fence (*kind*, *next_state*)) can be committed if:

1. if i is a normal fence and it has `.pr` set, all program-order-earlier load instructions are finished;
2. if i is a normal fence and it has `.pw` set, all program-order-earlier store instructions are finished; and
3. if i is a `fence.tso`, all program-order-earlier load and store instructions are finished.

Action:

1. record that i is committed; and
2. update the state of i to Plain (*next_state*).

Register read An instruction instance i in state Plain (Read_reg (*reg_name*, *read_cont*)) can do a register read of *reg_name* if every instruction instance that it needs to read from has already performed the expected *reg_name* register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source for this bit is the initial register value from *initial_register_state*. Let *reg_value* be the value assembled from *read_sources*. Action:

1. add *reg_name* to i.*reg_reads* with *read_sources* and *reg_value*; and
2. update the state of i to Plain (*read_cont reg_value*).

Register write An instruction instance i in state Plain (Write_reg (*reg_name*, *reg_value*, *next_state*)) can always do a register write to register *reg_name*. Action:

1. add *reg_name* to i.*reg_writes* with *deps* and *reg_value*; and
2. update the state of i to Plain (*next_state*).

where *deps* is a pair of the set of the *read_sources* from i.*reg_reads*, and a flag that is true if i is a load instruction instance that has already been entirely satisfied.

Pseudocode internal step An instruction instance i in state Plain (Internal (*next_state*)) can always do that pseudocode-internal step. Action: update the state of i to Plain (*next_state*).

Finish instruction A non-finished instruction instance i in state Plain (Done ()) can be finished if:

1. if i is a load instruction:
   1.1. all program-order-previous load-acquire instructions are finished;
   1.2. all program-order-previous `fence` instructions with `.sr` set are finished;
   1.3. for every program-order-previous `fence.tso` instruction f that is not finished, all load instructions that are program-order-before f are finished; and
   1.4. it is guaranteed that the values read by the memory reads of i will not cause coherence violations, i.e., for any program-order-previous instruction instance i', let *cfp* be the

combined footprint of propagated memory writes from store instructions program-order-between i and i', and *fixed memory writes* that were forwarded to i from store instructions program-order-between i and i' including i', and let $\overline{cfp}$ be the complement of cfp in the memory footprint of i. If $\overline{cfp}$ is not empty:

1.4.1. i' has a fully determined memory footprint;

1.4.2. i' has no unpropagated memory writes that overlap with $\overline{cfp}$; and

1.4.3. if i' is a load with a memory footprint that overlaps with $\overline{cfp}$, then all the memory reads of i' that overlap with $\overline{cfp}$ are satisfied and i' is *non-restartable*.

Here an instruction is non-restartable under the same conditions as specified previously for PROPAGATE MEMORY WRITE. Moreover, a memory write is called fixed if the store instruction has fully determined data.

2. i has a fully determined data; and
3. if i is not a fence, all program-order-previous conditional branch and indirect jump instructions are finished.

Action:

1. if i is a conditional branch or indirect jump instruction, discard any untaken paths of execution, i.e., remove all instruction instances that are not reachable by the branch/jump taken in *instruction_tree*; and
2. record the instruction as finished, i.e., set *finished* to *true*.

Appendix F

Promising-ARM/RISC-V appendix

F.1 Certification with ARMv8 store exclusives

Section 9.2.5 and Section 9.3 introduced a simple certification definition to avoid model executions in which not all promises are fulfilled. This certification is sound for RISC-V and ARMv8, and precise for RISC-V programs and for ARMv8 programs without store exclusives. In the presence of ARMv8 store exclusive instructions, however, it is imprecise: matching ARMv8's architecturally intended weak semantics of store exclusive instructions in Promising-ARM leads to executions in which the model gets stuck due to unfulfilled promises, of a similar sort as those present in the Flat model. This section, extends the model's machine state with locks and a certification that takes the locking into account to prevent these executions and make certification sound and precise for ARMv8 programs even with store exclusives, and makes the model deadlock-free.[1]

F.1.1 The challenge of certification with ARMv8 store exclusives

One of the main simplifications of the revised ARMv8 concurrency architecture was that, where the architecture previously distinguished between notions of "true" and "false" dependencies, it now makes no such distinction. Now syntactic dependencies of the right kind induce memory ordering, with no consideration of whether the result of the register computation varies as a function of its input or not. Therefore, in the revised ARMv8, it is not always sound to replace an expression by another expression that performs the same register computation. However, ARMv8's specification of store exclusives intends to allow processors to treat a load/store exclusive pair as a single atomic operation that is guaranteed to succeed, e.g. treating $r_1 := \texttt{load}_{\text{ex}}\,[x]; r_2 := \texttt{store}_{\text{ex}}\,[x]\,(r_1 + 1)$ as $r_1 := \texttt{fetch-and-add}\,[x]; r_2 := 0$. Now, r_2 still has to be set to indicate success (v_{succ}), but does not syntactically depend on the store. Therefore a dependency on r_2 in the program above does not induce ordering (and Promising-ARM sets its associated view to 0)[2]. But the value of r_2 after the store exclusive still depends on the success of the store exclusive!

Similar to the issues arising in the Flat model, this leads to surprising behaviours in the Promising model: in the following program despite the dependency from b to c they can be re-ordered. In particular, Thread 0 may (1.) execute a and read the initial value 0 (so $r_1 = 0$) and, (2.) assume the success of b (so $r_2 = \mathsf{v}_{\text{succ}} = 0$) and write $p = 1$ with c, *before b is in memory*: the architecture allows that d reads $p = 1$ and f reads $x = 0$ after the barrier e. (While in RISC-V the dependency

[1] As before, this chapter is from joint work with Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur, and the text is taken from a joint paper, currently under submission [103].

[2] This corresponds to the Flat model's handling of store exclusive instructions, where a store exclusive determines its success as the first action after fetching, thus introducing no ordering for later instructions depending on the success.

from b to c means b propagates before c, forbidding this behaviour.)

$$\begin{array}{l||l||l}
a : r_1 := \texttt{load}_{\text{ex}}\,[x]; & d : r_3 := \texttt{load}\,[p];\ //\ 1 & g : \texttt{store}\,[x]\,2 \\
b : r_2 := \texttt{store}_{\text{ex}}\,[x]\,(r_1+1); & e : \texttt{dmb sy}; & \\
c : \texttt{store}\,[p]\,(1-r_1-r_2) & f : r_4 := \texttt{load}\,[x]\ //\ 0 &
\end{array}$$

$$r_3 = 1 \wedge r_4 = 0 \text{ allowed}$$

This means whereas in all other cases a store may propagate to memory only when all its dependencies are "established", dependencies on the success of a store exclusive are special. In order to allow the above re-ordering of b and c, an operational model has to do extra work, since it has to ensure the success of b and therefore its atomicity with respect to the write a read from, until b is done. For the simple case above, mimicking the behaviour of a processor and replacing $r_1 := \texttt{load}_{\text{ex}}\,[x]; r_2 := \texttt{store}_{\text{ex}}\,[x]\,(r_1+1)$ with $r_1 := \texttt{fetch-and-add}\,[x]; r_2 := 0$ is easy. However, handling the dependency relaxation in its full generality (without deadlocks) is difficult.

This problem manifests in Promising-ARM in the following way: in the initial state Thread 0 is allowed to promise $p = 0$. Since c can produce a write $p = 0$ only if a reads $x = 0$ and b succeeds, the ability of Thread 0 to fulfil the promise now depends b's success, and so on whether b's write can enter memory as the next write to location x (after the initial write $x = 0$). If, however, g now writes to x, b will fail and the model gets stuck, with c unfulfilled. Since c's early promise has to be allowed to match ARMv8 semantics, the model must instead prevent g from writing until b's write, since g would break Thread 0's promise.

The certification definition presented in Section 9.2 and 9.3 works thread-locally: it takes into account only a thread's state and current memory in order to decide whether a thread step should be allowed or not. But whether g's write should be allowed cannot be determined based only on the state of Thread 2 and on the list of messages in memory: it is Thread 0's promises due to which g must not write. So a precise certification algorithm for ARMv8 needs to take into account some information about other threads. In this example, Thread 0 effectively requires "locking" location x, to constrain the behaviour of the other threads. The extended model for ARMv8 applies this intuition of a thread locking a location and extends memory with a lock state, in order to still allow certifying a thread by taking into account only its own state and the (extended) memory state.

F.1.2 Extended certification, take 1

The example indicates a pattern: in the problematic execution a thread's ability to fulfil its promises depends on both, a read exclusive, and the success of a paired write exclusive that has been promised. More precisely, the state in which a Thread n requires a lock on a location x is the following:

- Thread n depends on a load exclusive l to location x. This is if:
  - Thread n has already executed l, or
  - Thread n has an outstanding promise whose fulfilment depends on l due to register dataflow or due to coherence or view requirements.
- And Thread n relies on the success of the write of a store exclusive s to x that is paired

with l: Thread n has an outstanding promise whose fulfilment depends on s via register dataflow.

- And the write w' that l read from is already in memory, whereas the write of s is not.

If the above holds, Thread n's dependency on l "fixes" the write w' that l reads from, and due to the success dependency on s requires the write of s to succeed and be atomic with respect to w'.

The idea underlying the extended model is to precisely detect cases when this condition holds, and to then lock x for Thread n in memory for as long as the condition holds and prevent other threads from writing to locked locations. The main challenge here is in detecting the above condition. The model handles this by extending the certification and generalising the views of the thread state. During the certification the model tracks dependencies from load and store exclusive instructions to other stores, in order to detect when the fulfilment of a write promise by some store depends on such load/store exclusive pairs as described in the condition. To this end, the extended certification uses views that in addition to timestamps carry *taints* that keep track of the load/store exclusive instruction dependencies, including information about their memory location and pairing.

In the example above, in the state after Thread 0 has read $x = 0$ with a and promised $p = 1$ with c, the extended certification will work as follows:

- the only certifying execution of Thread 0 alone under current memory is one where a reads $x = 0$, and b succeeds. In this execution:
- a taints r_1 to indicate it is from a load exclusive a to location x reading from a value in memory.
- b taints r_2 to indicate it is from a successful store exclusive to location x whose write is not in memory yet and that is paired with a.
- when fulfilling $p = 0$ with c, c's pre-view includes a taint with information about both a and b: a and b are paired and to location x; a reads at timestamp 0, so from a write in memory; b is not propagated yet.
- Therefore Thread 0 requires locking x.

The information returned by the certification is then, which locations have to be locked in order to *guarantee* a thread can fulfil its promises, and a machine step is only allowed if the step is compatible with the current lock state in memory: not writing to a location locked by another thread, and not locking already-locked locations.

The taint tracking introduces complexity to the certification. Importantly however, during "normal" execution the extended model's views are simple timestamps just as before (Section 9.3), and taints are not persistent but local to the certification.

F.1.3 Extended certification, take 2

As the following example illustrates, unfortunately the ideas on certification above are still insufficient. In this example Thread 0's store d depends on the load exclusive b to x, and the "success register write" of the store exclusive c to location x. New here is that c has release ordering, and so c is ordered after the write a to y. Symmetrically in Thread 1 h depends on the load and

store exclusive instructions f and g to y, and g is a store exclusive release ordered after a store e to x.

$$\begin{array}{l|l} a : \texttt{store}\,[y]\,1; & e : \texttt{store}\,[x]\,1; \\ b : r_1 := \texttt{load}_{\text{ex}}\,[x]; & f : r_3 := \texttt{load}_{\text{ex}}\,[y]; \\ c : r_2 := \texttt{store}_{\text{ex,rel}}\,[x]\,1; & g : r_4 := \texttt{store}_{\text{ex,rel}}\,[y]\,1; \\ d : \texttt{store}\,[p]\,(1 - r_1 - r_2) & h : \texttt{store}\,[q]\,(1 - r_3 - r_4) \end{array}$$

Now assume an execution in which Thread 0 promises $p = 1$. Since this depends on b reading $x = 0$ and c eventually successfully writing $x = 1$, the extended certification requires a lock on x for Thread 0 to prevent Thread 1 from breaking its promise. Now Thread 1 could analogously promise $q = 1$, after which the model also locks location y for Thread 1, which does not contradict Thread 0 locking x. But now the model is stuck again: Thread 0 cannot execute a, since y is locked by Thread 1; c's write $x = 1$ would release the lock on x, but since c is a store release, this requires first promising a. Thread 1 in turn cannot execute e due to the lock on x, and cannot promise g's $y = 1$ (and unlock y) before executing e.

In order to avoid such executions, the extended certification has to be improved to take into account some information about the thread-internal ordering requirements in order to prevent model deadlocks. To this end, the extended model's taints carry additional information about stores preceding store exclusive release instructions and the lock state captures rely-guarantee style lock information per thread; a machine step is then only allowed if the rely-guarantee lock information of all thread states is consistent. Then in the previous bad execution:

- For the promise $p = 1$ the certification returns information of the form $([y]; x)$, meaning that for this step Thread 0 requires a lock on x, and that it *relies* on being able to write to y before releasing the lock on x. Promising $p = 1$ adds this to the memory's lock state.
- Since there are no other locks, Thread 0 can promise.
- In the following state, for the promise $q = 1$ by Thread 1, symmetrically, the certification returns the information $([x], y)$.
- Now $([x], y)$ is incompatible with $([y], x)$ due to the cyclic rely-guarantee dependency. Thus Thread 1 is not allowed to promise $q = 1$.

Certain sequences of multiple such store exclusive release instructions can lead to *nesting* of these rely-guarantee locks, making the consistency checking difficult. In particular, a naive algorithm for checking the consistency is exponential in the nesting depth. In practice it should seem that sequences with nesting depth greater than 1 do not occur "naturally". Hence, for the purpose of exhaustive exploration, the executable model could approximate the lock information and consistency checking up to depth one. The model may then still get stuck in cases requiring depth more than 1, but consistency checking becomes linear in the size of the lock information. This text omits the details of the extended certification.

Bibliography

[1] *The SPARC Architecture Manual, Version 9*. SPARC International, Inc., 1994. SAV09R1459912.

[2] *Power ISA^{TM} Version 2.07*. IBM, 2013.

[3] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Intel Corporation, 2018. 325462-066US.

[4] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):502–515, 2003. ISSN 1045-9219. doi: 10.1109/TPDS.2003.1199067. URL https://doi.org/10.1109/TPDS.2003.1199067.

[5] S. V. Adve. *Designing Memory Consistency Models for Shared-memory Multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354.

[6] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996. doi: 10.1109/2.546611. URL https://doi.org/10.1109/2.546611.

[7] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995. doi: 10.1007/BF01784241. URL https://doi.org/10.1007/BF01784241.

[8] J. Alglave. A shared memory poetics. *These de doctorat, L'université Paris Denis Diderot*, 2010.

[9] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012. doi: 10.1007/s10703-012-0161-5. URL https://doi.org/10.1007/s10703-012-0161-5.

[10] J. Alglave and L. Maranget. The `diy` tool. http://diy.inria.fr/. Online; accessed 9 March 2019.

[11] J. Alglave and L. Maranget. Stability in weak memory models. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 50–66, 2011. doi: 10.1007/978-3-642-22110-1_6. URL https://doi.org/10.1007/978-3-642-22110-1_6.

[12] J. Alglave and L. Maranget. http://diy.inria.fr/doc/index.html, April 2017. Online; accessed 9 March 2019.

[13] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, pages 13–24, 2009. doi: 10.1145/1481839.1481842. URL https://doi.org/10.1145/1481839.1481842.

[14] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 258–272, 2010. doi: 10.1007/978-3-642-14295-6_25. URL https://doi.org/10.1007/978-3-642-14295-6_25.

[15] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 41–44, 2011. doi: 10.1007/978-3-642-19835-9_5. URL https://doi.org/10.1007/978-3-642-19835-9_5.

[16] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 512–532. Springer, 2013. doi: 10.1007/978-3-642-37036-6_28. URL https://doi.org/10.1007/978-3-642-37036-6_28.

[17] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):7:1–7:74, 2014. doi: 10.1145/2627752. URL https://doi.org/10.1145/2627752.

[18] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *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, New York, NY, USA, 2015. doi: 10.1145/2694344.2694391. URL https://doi.org/10.1145/2694344.2694391.

[19] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. S. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *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, New York, NY, USA, 2018. ACM. doi: 10.1145/3173162.3177156. URL https://doi.org/10.1145/3173162.3177156.

[20] *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd., 2015. ARM DDI 0487A.h (ID092915).

[21] *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd., 2016. ARM DDI 0487A.k_iss10775 (ID092916).

[22] *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd., 2017. ARM DDI 0487B.a (ID033117).

[23] *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. ARM Ltd., 2018. ARM DDI 0406C.d (ID040418).

[24] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3(POPL):71:1–71:31, 2019. URL https://dl.acm.org/citation.cfm?id=3290384.

[25] Arvind and J. Maessen. Memory model = instruction reordering + store atomicity. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*, pages 29–40, 2006. doi: 10.1109/ISCA.2006.26. URL https://doi.org/10.1109/ISCA.2006.26.

[26] K. Asanović. The RISC-V Memory Consistency Model. https://riscv.org/2017/04/risc-v-memory-consistency-model/, April 2017. Online; accessed 9 March 2019.

[27] D. Aspinall and J. Sevcík. Formalising java's data race free guarantee. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 22–37, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-74591-4_4. URL https://doi.org/10.1007/978-3-540-74591-4_4.

[28] H. Attiya and R. Friedman. Programming dec-alpha based multiprocessors the easy way (extended abstract). In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '94, Cape May, New Jersey, USA, June 27-29, 1994*, pages 157–166, New York, NY, USA, 1994. ACM. doi: 10.1145/181014.192323. URL https://doi.org/10.1145/181014.192323.

[29] S. Awodey. Category theory, volume 49 of oxford logic guides, 2006.

[30] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66, 2011. doi: 10.1145/1926385.1926394. URL https://doi.org/10.1145/1926385.1926394.

[31] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520, 2012. doi: 10.1145/2103656.2103717. URL https://doi.org/10.1145/2103656.2103717.

[32] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. 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 283–307, 2015. doi: 10.1007/978-3-662-46669-8_12. URL https://doi.org/10.1007/978-3-662-46669-8_12.

[33] J. Bornholt and E. Torlak. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 467–481, 2017. doi: 10.1145/3062341.3062353. URL https://doi.org/10.1145/3062341.3062353.

[34] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 392–403, 2009. doi: 10.1145/1480881.1480930. URL https://doi.org/10.1145/1480881.1480930.

[35] G. Boudol, G. Petri, and B. P. Serpette. Relaxed operational semantics of concurrent programming languages. In *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS 2012, Newcastle upon Tyne, UK, September 3, 2012.*, pages 19–33, 2012. doi: 10.4204/EPTCS.89.3. URL https://doi.org/10.4204/EPTCS.89.3.

[36] A. Bradbury, G. Ferris, and R. Mullins. Tagged memory and minion cores in the lowrisc soc. Technical report, 2014. Online; accessed 9 March 2019.

[37] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 107–120, 2008. doi: 10.1007/978-3-540-70545-1_12. URL https://doi.org/10.1007/978-3-540-70545-1_12.

[38] S. Burckhardt, M. Musuvathi, V. Singh, and M. Musuvathi. Verifying compiler transformations for concurrent programs. Technical report, November 2008. URL https://www.microsoft.com/en-us/research/publication/verifying-compiler-transformations-for-concurrent-programs/.

[39] P. Cenciarelli, A. Knapp, and E. Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *Programming Languages and Systems, 16th European Symposium*

on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, pages 331–346, Berlin, Heidelberg, 2007. Springer. doi: 10.1007/978-3-540-71316-6_23. URL https://doi.org/10.1007/978-3-540-71316-6_23.

[40] T. J. W. I. R. Center and R. K. Treiber. *Systems programming: coping with parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986. URL https://books.google.co.uk/books?id=YQg3HAAACAAJ.

[41] P. Chatterjee and G. Gopalakrishnan. Towards A formal model of shared memory consistency for intel itanium^{tm}. In *19th International Conference on Computer Design (ICCD 2001), VLSI in Computers and Processors, 23-26 September 2001, Austin, TX, USA, Proceedings*, pages 515–518. IEEE, 2001. doi: 10.1109/ICCD.2001.955081. URL https://doi.org/10.1109/ICCD.2001.955081.

[42] N. Chong and S. Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008*, pages 16–19, 2008. doi: 10.1145/1353522.1353528. URL https://doi.org/10.1145/1353522.1353528.

[43] W. W. Collier. *Reasoning about parallel architectures*. Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-767187-3.

[44] A. A. Committee et al. *Alpha Architecture Reference Manual*. Digital Press, 2014.

[45] F. Corella, J. M. Stone, and C. M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM, 1993.

[46] K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 623–636, New York, NY, USA, 2015. ACM. doi: 10.1145/2676726.2676984. URL https://doi.org/10.1145/2676726.2676984.

[47] W. Deacon. The armv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat, 2016. Commit f7e72dc.

[48] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture, Tokyo, Japan, June 1986*, pages 434–442, 1986. URL https://dl.acm.org/citation.cfm?id=17406.

[49] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005. doi: 10.1145/1040305.1040315. URL https://doi.org/10.1145/1040305.1040315.

[50] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993. doi: 10.1145/155090.155113. URL https://doi.org/10.1145/155090.155113.

[51] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the armv8 architecture, operationally: concurrency and ISA. In *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 608–621, 2016. doi: 10.1145/2837614.2837615. URL https://doi.org/10.1145/2837614.2837615.

[52] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell. Mixed-size concurrency: Arm, power, c/c++11, and SC. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 429–442, 2017. URL http://dl.acm.org/citation.cfm?id=3009839.

[53] A. C. J. Fox. Directions in ISA specification. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 338–344, 2012. doi: 10.1007/978-3-642-32347-8_23. URL https://doi.org/10.1007/978-3-642-32347-8_23.

[54] A. C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 187–202, 2015. doi: 10.1007/978-3-319-22102-1_12. URL https://doi.org/10.1007/978-3-319-22102-1_12.

[55] A. C. J. Fox, M. O. Myreen, Y. K. Tan, and R. Kumar. Verified compilation of CakeML to multiple machine-code targets. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 125–137, 2017. doi: 10.1145/3018610.3018621. URL https://doi.org/10.1145/3018610.3018621.

[56] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. *WRL Research Report*, 95(9), 1995.

[57] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, pages 15–26, 1990. doi: 10.1145/325164.325102. URL https://doi.org/10.1145/325164.325102.

[58] J. R. Goodman. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[59] J. Gosling, W. N. Joy, and G. L. S. Jr. *The Java Language Specification*. Addison-Wesley Longman Publishing, 1996. ISBN 0-201-63451-1.

[60] K. E. Gray, G. Kerneis, D. P. Mulligan, C. Pulte, S. Sarkar, and P. 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, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 635–646, 2015. doi: 10.1145/2830772.2830775. URL https://doi.org/10.1145/2830772.2830775.

[61] S. Hangal, D. Vahia, C. Manovit, J. J. Lu, and S. Narayanan. Tsotool: A program for verifying memory systems using the memory consistency model. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, pages 114–123, 2004. doi: 10.1109/ISCA.2004.1310768. URL https://doi.org/10.1109/ISCA.2004.1310768.

[62] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models. Part I: Definitions and comparisons. Technical report, Department of Computer Science, The University of Calgary, 1998. Online; accessed 8 March 2019.

[63] L. Higham, L. Jackson, and J. Kawash. What is itanium memory consistency from the programmer's point of view? *Electronic Notes in Theoretical Computer Science*, 174(9):63–84, 2007. doi: 10.1016/j.entcs.2007.04.007. URL https://doi.org/10.1016/j.entcs.2007.04.007. Proceedings of the Thread Verification Workshop (TV 2006).

[64] D. Howells, P. E. McKenney, W. Deacon, and P. Zijlstra. Documentation/memory-barriers.txt. https://www.kernel.org/doc/Documentation/memory-barriers.txt, 2018. Online; accessed 9 March 2019.

[65] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002.

[66] *Intel® Itanium® Architecture Software Developer's Manual*. Intel Corporation, 2010. Revision 2.3.

[67] D. Jackson. Alloy: A logical modelling language. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, page 1, 2003. doi: 10.1007/3-540-44880-2_1. URL https://doi.org/10.1007/3-540-44880-2_1.

[68] A. Jeffrey and J. Riely. On thin air reads towards an event structures model of relaxed memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 759–767, New York, NY, USA, 2016. ACM. doi: 10.1145/2933575.2934536. URL https://doi.org/10.1145/2933575.2934536.

[69] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. R. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla$^+$. *Journal of Formal Methods in System Design*, 22(2):125–131, 2003. doi: 10.1023/A:1022969405325. URL https://doi.org/10.1023/A:1022969405325.

[70] J. Kang. crossbeam-rfcs. https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md, 2018.

[71] J. Kang, C. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189, 2017. URL http://dl.acm.org/citation.cfm?id=3009850.

[72] S. Kell, D. P. Mulligan, and P. Sewell. The missing link: explaining ELF static linking, semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 607–623, New York, NY, USA, 2016. ACM. doi: 10.1145/2983990.2983996. URL https://doi.org/10.1145/2983990.2983996.

[73] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL):17:1–17:32, 2018. doi: 10.1145/3158105. URL https://doi.org/10.1145/3158105.

[74] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *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, 2016. doi: 10.1145/2837614.2837643. URL https://doi.org/10.1145/2837614.2837643.

[75] O. Lahav, V. Vafeiadis, J. Kang, C. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 618–632, 2017. doi: 10.1145/3062341.3062352. URL https://doi.org/10.1145/3062341.3062352.

[76] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439. URL https://doi.org/10.1109/TC.1979.1675439.

[77] N. M. Lê, A. Pop, A. Cohen, and F. Z. Nardelli. Correct and efficient work-stealing for weak memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 69–80, 2013. doi: 10.1145/2442516.2442524. URL https://doi.org/10.1145/2442516.2442524.

[78] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

[79] Linux contributors. Documentation/locking/spinlocks.txt. https://www.kernel.org/doc/Documentation/locking/spinlocks.txt, 2014. Online; accessed 9 March 2019.

[80] R. J. Lipton and J. S. Sandberg. *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[81] P. N. Loewenstein, S. Chaudhry, R. Cypher, and C. Manovit. Multiprocessor memory model verification. *Proceedings of AFM (Automated Formal Methods)*, 2006.

[82] D. Lustig, M. Pellauer, and M. Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 635–646, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.38. URL http://dx.doi.org/10.1109/MICRO.2014.38.

[83] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi. Armor: defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, June 13-17, 2015*, pages 388–400, New York, NY, USA, 2015. ACM. doi: 10.1145/2749469.2750378. URL https://doi.org/10.1145/2749469.2750378.

[84] D. Lustig, G. Sethi, A. Bhattacharjee, and M. Martonosi. Transistency models: Memory ordering at the hardware-os interface. *IEEE Micro*, 37(3):88–97, 2017. doi: 10.1109/MM.2017.69. URL https://doi.org/10.1109/MM.2017.69.

[85] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux. Automated synthesis of comprehensive memory model litmus test suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 661–675, 2017. doi: 10.1145/3037697.3037723. URL https://doi.org/10.1145/3037697.3037723.

[86] S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 273–287, 2010. doi: 10.1007/978-3-642-14295-6_26. URL https://doi.org/10.1007/978-3-642-14295-6_26.

[87] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 495–512, 2012. doi: 10.1007/978-3-642-31424-7_36. URL https://doi.org/10.1007/978-3-642-31424-7_36.

[88] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. Ccicheck: using μhb graphs to verify the coherence-consistency interface. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 26–37, 2015. doi: 10.1145/2830772.2830782. URL https://doi.org/10.1145/2830772.2830782.

[89] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer. Rtlcheck: Verifying the memory consistency of RTL designs. In *Proceedings of the 50th Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017, pages 463–476, 2017. doi: 10.1145/3123939.3124536. URL https://doi.org/10.1145/3123939.3124536.

[90] Y. A. Manerkar, D. Lustig, M. Martonosi, and A. Gupta. Pipeproof: Automated memory consistency proofs for microarchitectural specifications. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 788–801, 2018. doi: 10.1109/MICRO.2018.00069. URL https://doi.org/10.1109/MICRO.2018.00069.

[91] J. Manson and W. Pugh. Core semantics of multithreaded java. In *Proceedings of the ACM 2001 Java Grande Conference, Stanford University, California, USA, June 2-4, 2001*, pages 29–38, New York, NY, USA, 2001. ACM. URL http://portal.acm.org/citation.cfm?id=376656.376806.

[92] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391, New York, NY, USA, 2005. ACM. doi: 10.1145/1040305.1040336. URL https://doi.org/10.1145/1040305.1040336.

[93] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012. Online; accessed 8 March 2019.

[94] P. E. McKenney. Memory ordering in modern microprocessors, part i. *Linux Journal*, 2005 (136), August 2005. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1080072.1080074.

[95] M. Might. A-Normalization: Why and How. http://matt.might.net/articles/a-normalization/. Online; accessed 30 January 2018.

[96] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188, 2014. doi: 10.1145/2628136.2628143. URL https://doi.org/10.1145/2628136.2628143.

[97] K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for C/C++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 111–128, New York, NY, USA, 2016. ACM. doi: 10.1145/2983990.2983997. URL https://doi.org/10.1145/2983990.2983997.

[98] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany,*

August 17-20, 2009. Proceedings, pages 391–407, 2009. doi: 10.1007/978-3-642-03359-9_27. URL https://doi.org/10.1007/978-3-642-03359-9_27.

[99] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *7th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '95, Santa Barbara, California, USA, July 17-19, 1995*, pages 34–41, 1995. doi: 10.1145/215399.215413. URL https://doi.org/10.1145/215399.215413.

[100] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *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 622–633, 2016. doi: 10.1145/2837614.2837616. URL https://doi.org/10.1145/2837614.2837616.

[101] A. Podkopaev, O. Lahav, and V. Vafeiadis. Promising compilation to armv8 POP. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pages 22:1–22:28, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.22. URL https://doi.org/10.4230/LIPIcs.ECOOP.2017.22.

[102] W. Pugh. Fixing the java memory model. In *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12-14, 1999*, pages 89–98, 1999. doi: 10.1145/304065.304106. URL https://doi.org/10.1145/304065.304106.

[103] C. Pulte, J. Pichon-Pharabod, J. Kang, S.-H. Lee, and C.-K. Hur. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. Submitted for publication. URL https://sf.snu.ac.kr/promising-arm-riscv/.

[104] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL):19:1–19:29, 2018. doi: 10.1145/3158107. URL https://doi.org/10.1145/3158107.

[105] A. Reid. Trustworthy specifications of arm® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 161–168, 2016. doi: 10.1109/FMCAD.2016.7886675. URL https://doi.org/10.1109/FMCAD.2016.7886675.

[106] A. Reid. ARM releases machine readable architecture specification. https://alastairreid.github.io/ARM-v8a-xml-release/, April 2017. Online; accessed 9 March 2019.

[107] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi. End-to-end verification of processors with isa-formal. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 42–58, 2016. doi: 10.1007/978-3-319-41540-6_3. URL https://doi.org/10.1007/978-3-319-41540-6_3.

[108] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. Fast and generalized polynomial time memory consistency verification. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 503–516, 2006. doi: 10.1007/11817963_46. URL https://doi.org/10.1007/11817963_46.

[109] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 379–391, 2009. doi: 10.1145/1480881.1480929. URL https://doi.org/10.1145/1480881.1480929.

[110] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011. doi: 10.1145/1993498.1993520. URL https://doi.org/10.1145/1993498.1993520.

[111] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 311–322, 2012. doi: 10.1145/2254064.2254102. URL https://doi.org/10.1145/2254064.2254102.

[112] J. Sevcík and D. Aspinall. On validity of program transformations in the java memory model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pages 27–51, Berlin, Heidelberg, 2008. Springer. doi: 10.1007/978-3-540-70592-5_3. URL https://doi.org/10.1007/978-3-540-70592-5_3.

[113] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)*, 60(3):22:1–22:50, 2013. doi: 10.1145/2487241.2487248. URL https://doi.org/10.1145/2487241.2487248.

[114] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. doi: 10.1145/1785414.1785443. URL https://doi.org/10.1145/1785414.1785443.

[115] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. *Formal Specification of Memory Models*, pages 25–41. Springer US, Boston, MA, 1992. ISBN 978-1-4615-3604-8. doi: 10.1007/978-1-4615-3604-8_2. URL https://books.google.co.uk/books?id=wBAp7xjAIGoC&lpg=PA25&ots=c9b64kvNJv&lr&hl=de&pg=PA25#v=onepage&q&f=false.

[116] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5):800–849, 2004. doi: 10.1145/1017460.1017464. URL https://doi.org/10.1145/1017460.1017464.

[117] K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. A separation logic for a promising semantics. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 357–384, 2018. doi: 10.1007/978-3-319-89884-1_13. URL https://doi.org/10.1007/978-3-319-89884-1_13.

[118] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4): 423–436, 2008. doi: 10.1017/S0956796808006758. URL https://doi.org/10.1017/S0956796808006758.

[119] E. Torlak, M. Vaziri, and J. Dolby. Memsat: checking axiomatic specifications of memory models. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 341–350, New York, NY, USA, 2010. ACM. doi: 10.1145/1806596.1806635. URL https://doi.org/10.1145/1806596.1806635.

[120] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 119–133, 2017. doi: 10.1145/3037697.3037719. URL https://doi.org/10.1145/3037697.3037719.

[121] A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. 2018.

[122] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 190–204, 2017. URL http://dl.acm.org/citation.cfm?id=3009838.

[123] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press. doi: 10.1109/ISCA.2014.6853201. URL https://doi.org/10.1109/ISCA.2014.6853201.

[124] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying java thread semantics using a uniform memory model. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande 2002, Seattle, Washington, USA, November 3-5, 2002*, pages 192–201, New York, NY, USA, 2002. ACM. doi: 10.1145/583810.583832. URL https://doi.org/10.1145/583810.583832.

[125] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the intel itanium memory ordering rules using logic programming and SAT. In *Correct Hardware Design and*

Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings, pages 81–95. Springer, 2003. doi: 10.1007/978-3-540-39724-3_9. URL https://doi.org/10.1007/978-3-540-39724-3_9.

[126] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, 2004. doi: 10.1109/IPDPS.2004.1302944. URL https://doi.org/10.1109/IPDPS.2004.1302944.

[127] S. Zhang, Arvind, and M. Vijayaraghavan. Taming weak memory models. *CoRR*, abs/1606.05416, 2016. URL http://arxiv.org/abs/1606.05416.

[128] S. Zhang, M. Vijayaraghavan, and Arvind. Weak memory models: Balancing definitional simplicity and implementation flexibility. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*, pages 288–302, 2017. doi: 10.1109/PACT.2017.29. URL https://doi.org/10.1109/PACT.2017.29.