Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8

CHRISTOPHER PULTE, University of Cambridge, UK
SHAKED FLUR, University of Cambridge, UK
WILL DEACON, ARM Ltd., UK
JON FRENCH, University of Cambridge, UK
SUSMIT SARKAR, University of St. Andrews, UK
PETER SEWELL, University of Cambridge, UK

ARM has a relaxed memory model, previously specified in informal prose for ARMv7 and ARMv8. Over time, and partly due to work building formal semantics for ARM concurrency, it has become clear that some of the complexity of the model is not justified by the potential benefits. In particular, the model was originally *non-multicopy-atomic*: writes could become visible to some other threads before becoming visible to all — but this has not been exploited in production implementations, the corresponding potential hardware optimisations are thought to have insufficient benefits in the ARM context, and it gives rise to subtle complications when combined with other ARMv8 features. The ARMv8 architecture has therefore been revised: it now has a multicopy-atomic model. It has also been simplified in other respects, including more straightforward notions of dependency, and the architecture now includes a formal concurrency model.

In this paper we detail these changes and discuss their motivation. We define two formal concurrency models: an operational one, simplifying the Flowing model of Flur et al., and the axiomatic model of the revised ARMv8 specification. The models were developed by an academic group and by ARM staff, respectively, and this extended collaboration partly motivated the above changes. We prove the equivalence of the two models. The operational model is integrated into an executable exploration tool with new web interface, demonstrated by exhaustively checking the possible behaviours of a loop-unrolled version of a Linux kernel lock implementation, a previously known bug due to unprevented speculation, and a fixed version.

CCS Concepts: • **Computer systems organization → Architectures**; • **Theory of computation → Concurrency**; **Semantics and reasoning**; • **Software and its engineering → Concurrent programming languages**; **Assembly languages**;

Additional Key Words and Phrases: Relaxed Memory Models, Semantics, Operational, Axiomatic

ACM Reference Format:
Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (January 2018), 29 pages. https://doi.org/10.1145/3158107

Authors' addresses: Christopher Pulte, University of Cambridge, UK, Christopher.Pulte@cl.cam.ac.uk; Shaked Flur, University of Cambridge, UK, Shaked.Flur@cl.cam.ac.uk; Will Deacon, ARM Ltd. UK, Will.Deacon@arm.com; Jon French, University of Cambridge, UK, Jon.French@cl.cam.ac.uk; Susmit Sarkar, University of St. Andrews, UK, ss265@st-andrews.ac.uk; Peter Sewell, University of Cambridge, UK, Peter.Sewell@cl.cam.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
2475-1421/2018/1-ART19
https://doi.org/10.1145/3158107

1 INTRODUCTION

What is the semantics of concurrent ARM machine-code programs? The vendor architecture manuals for ARMv7 and early ARMv8 described a relaxed memory model, with programmer-visible out-of-order and speculative execution, that was *non-multicopy-atomic*: a write could become visible to some other threads before becoming visible to all threads. In this it was broadly similar to the IBM POWER architecture. The two were unfortunately also alike in another way: their memory models were expressed as prose definitions that were hard to interpret precisely. (x86 and MIPS are similarly imprecise, while SPARC and Itanium have precise vendor-defined models.) This prompted extensive work to establish mathematically precise models for ARM and for IBM POWER, initially based on the vendor texts, but later, as the deficiencies of those became apparent, relying more heavily on experimental investigation of the behaviour of processor implementations, and on discussion with ARM and IBM architects of their architectural intent: Flur et al. [2017, 2016b]; Gray et al. [2015]; Alglave et al. [2014]; Maranget et al. [2012]; Sarkar et al. [2012]; Alglave et al. [2011]; Sarkar et al. [2011]; Alglave et al. [2010, 2009]; Sarkar et al. [2009]; Chong and Ishtiaq [2008]; Adir et al. [2003]; Corella et al. [1993].

Broadly speaking, over time these models have become more complete, covering more relaxed-memory phenomena and larger fragments of the architectures, and more accurate, with respect both to experimental results and to the architects' intentions. They have also become more complex — partly due to the increased coverage, but also as they have exposed more of the complexity implicit in those intentions.

This complexity has a cost, especially for architectures implemented by multiple vendors, and ARM has recently made a significant shift to a simpler model. In particular, while non-multicopy-atomic (non-MCA) behaviour is observable on IBM POWER implementations, the hardware implementation freedom it permits has not been exploited on production ARMv8 implementations, by any of the several vendors thereof. This was previously suggested by experimental data for some implementations [Maranget et al. 2012; Alglave et al. 2014; Flur et al. 2016b, 2017], and has been confirmed by discussion with ARM Architecture Partners; it is thought that in the ARM context the potential hardware performance benefits of non-MCA do not justify the ensuing complexity of implementation, validation and reasoning. Allowing non-MCA behaviour gives rise to substantial complexity in the model, especially when combined with the previous architectural desire for a model that provided as much implementation freedom as possible, and with the store-release/load-acquire instructions added in ARMv8. Accordingly, ARM have now revised their ARMv8 specification to prohibit non-MCA behaviour: when a write is visible to some other thread, it is now guaranteed to be visible to all [ARM Ltd. 2017, BS-84,90]. Further, the revised ARMv8 specification document now includes, for the first time, a formal memory model to specify (for a fragment of the architecture) exactly what behaviour is and is not allowed; the architecture contains a prose version of the formal model by Deacon [2016]. The specification is also being clarified in various other respects, including the definition of inter-instruction dependencies.

In this paper we detail these changes to the ARMv8 concurrency architecture and discuss the motivation for them. We define two formal concurrency models: an operational model that simplifies the Flowing model of Flur et al. [2016b, 2017], and the axiomatic model of the revised ARMv8 specification. The two models were developed by an academic group and by ARM staff, respectively, with an extended collaboration and discussion of examples that started for non-MCA ARMv8 and partly motivated the shift to MCA ARMv8. We then prove the two models equivalent, establishing the correspondence between particular operational model transitions and axiomatic model events. This increases confidence in both formal models and in the architecture itself, providing a clear operational intuition for the axiomatic model. The operational model is integrated into an executable

rmem tool, with a new web interface for interactive, pseudo-random, and exhaustive exploration of litmus tests and small ELF binaries. The tool can be used for testing and understanding the concurrency model, and for exploring the behaviour of concurrent software, at least in small instances. We demonstrate the latter for a Linux kernel lock implementation, by exhaustively checking the reachable states of a loop-unrolled version. In short, this paper contributes:

- discussion of the motivation for the new MCA ARMv8 concurrency architecture, including complexities that arose in the non-MCA case (§3);
- an informal description of the new architecture (§4);
- a formal semantics for MCA ARMv8 expressed as an operational model, with an abstract microarchitectural flavour (§5,6);
- a formal semantics for MCA ARMv8 expressed as an axiomatic model, as presented in the new architecture document and by Deacon [2016] (§5,7);
- proof of equivalence of the two models (§8);
- experimental validation of that equivalence and of the model soundness with respect to hardware (§9);
- a verification, simulation, and debugging tool, rmem, based on the operational model (§10);
- a demonstration of rmem in use, for verifying a lock implementation from the Linux kernel with respect to the MCA ARMv8 architecture semantics (§11).

The supplementary material includes details of the operational model, proof, and tests, at http://www.cl.cam.ac.uk/~pes20/armv8-mca/. The rmem tool is available at http://www.cl.cam.ac.uk/~pes20/rmem/, including the tests mentioned in the paper (best viewed in chrome/chromium). The new models should aid future work on reasoning about and verification of concurrent ARM code, as these become considerably simpler in the MCA case.

Caveats. The two formal semantics do not have identical coverage, and so our proof addresses the intersection of the two. The operational model is integrated with semantics for a substantial fragment of the instruction set, and supports mixed-size accesses. The axiomatic model does not include that ISA integration or support mixed-size, but it does cover the more relaxed form of load-acquire (LDAPR) introduced in ARMv8.3 and the atomic read-modify-write instructions introduced in ARMv8.1, which are not in the operational model. We believe that the operational model and proof could be straightforwardly extended to cover the LDAPR and read-modify-write instructions. Integrating the axiomatic model with the full ISA model requires work on the axiomatic model and tooling to support mixed-size (to the best of our knowledge no existing axiomatic model or tool does), including some open questions concerning the allowed behaviour, and work to handle the full ISA intra-instruction semantics.

The operational model can deadlock in cases where there are memory accesses or barriers between load/store-exclusive pairs, or when the store-exclusive has extra dependencies (these two are cases that are not really intended to be supported by the architecture), and when a load and a store-exclusive are paired successfully but have different addresses, which is still being clarified.

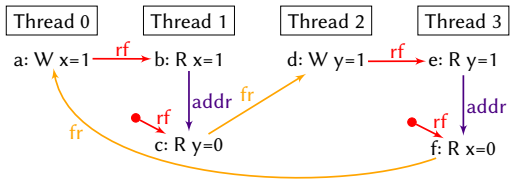
Neither model addresses systems aspects of the architecture, e.g. virtual memory. Finally, our proof is a hand proof, not mechanised, and currently is limited to finite executions.

2 BACKGROUND: THE NON-MCA FLOWING MODEL

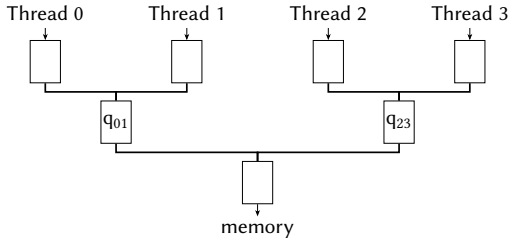
We briefly recall how the Flowing model exhibits non-MCA, as it was the model most often referred to in discussions with architects, matching their intuitions, and is simple to understand. We refer to Flur et al. [2016b] for a more complete description of this and the associated POP model. Flowing comprises an operational semantics for each hardware thread, with explicit out-of-order and speculative execution for each, connected via a storage subsystem comprising an arbitrary

tree-structured hierarchy of queues, of writes, read requests, and barriers, above a simple memory. When the storage subsystem receives a memory access or barrier, it is placed at the top of the queue that is associated with the submitting thread. The bottom-most event in a queue can "flow" to the top of the next queue in the topology. The memory is a map from byte locations to the most recent write events to those locations. The bottom-most event in the bottom-most queue can "flow" to memory: for a write event, updating the memory mapping; for a read event, sending a read-response to its thread with the appropriate value from the memory map; and, for a barrier event, removing the barrier. A read event may also be satisfied from within a queue if the event immediately below it is a write event to the same location. Finally, adjacent events in a queue (even if from different threads) may switch places with each other (reorder), subject to some constraints.

To demonstrate how Flowing exhibits non-MCA, consider the classic non-MCA `IRIW+addrs`[1] litmus test (`WRC+addrs` is similar but with (d) merged into (c)). Here Threads 0 and 2 write to the memory locations x and y, respectively, which are then read by Threads 1 and 3, in opposite orders; the intra-thread order of each pair of reads is preserved by address dependencies. In the execution shown, each thread observes the new value in its first read and the initial-state value in its second read. The vertical arrows are program-order (po) edges, describing the particular control-flow unfolding of the execution, including address (`addr`), data (`data`), and control (`ctrl`) dependencies; the reads-from (`rf`) edges indicate the source write (or initial state) for each memory read; and coherence order (`co`) indicates the order in which writes to the same location are sequentialised. The derived *from-reads* relation (`fr`) [Ahamad et al. 1995; Alglave et al. 2010] relates a read to all writes that are later in the coherence order than the write from which it read, defined as $\mathtt{rf}^{-1};\mathtt{co}$.



This behaviour can be observed in Flowing with the topology below. After Threads 0 and 2 perform their writes, the write `x=1` of Thread 0 can flow to the queue q_{01} that is shared between Threads 0 and 1, and the write `y=1` of Thread 2 can flow to the queue q_{23} that is shared between Threads 2 and 3. Now Thread 1 can perform its first read, which flows down until it reaches the write `x=1` (in queue q_{01}) and then it performs its second read which flows down all the way to memory, reordering with the write `x=1` on the way, and reads `y=0`. Finally Thread 3 performs its first read which flows down until it reaches the write `y=1` (in queue q_{23}) and then performs its second read which flows down all the way to memory, reordering with the write `y=1` on the way, and reads `x=0`.



3 COMPLEXITIES ARISING IN THE NON-MCA ARMv8 ARCHITECTURE

For regular memory accesses Flowing captures the original non-MCA intent with straightforward rules: memory accesses can be reordered only if to distinct memory locations. The strong memory barrier (dmb sy) is also straightforward: it cannot be reordered with any event. Flowing can exhibit non-architectural asymmetry between threads, which was addressed by the POP model.

[1]Throughout the paper, litmus test names highlighted in blue link to a state of `rmem` with the test loaded.

Things become more complex when considering other kinds of memory accesses that guarantee stronger properties (store-release, load-acquire and load/store-exclusive), and weaker kinds of barriers (dmb st and dmb ld). When the vendor documentation does not clearly define the architectural intent, our normal, and previously effective, strategy has been to create litmus tests that demonstrate the phenomenon, run those on real hardware using the litmus tool [Alglave et al. 2011], discuss with architects what microarchitectural mechanisms they implement (or think might reasonably be implemented) that would give rise to the phenomenon, and finally, adapt an operational model with abstractions of those mechanisms. For example, the "`PPOCA`" test of Sarkar et al. [2011] is observable on ARM and POWER because, microarchitecturally, reads can be satisfied from writes in thread-local store queues, even on speculative paths (if the write and read follow an as-yet-uncommitted conditional branch); that can be smoothly modelled with write-forwarding in the operational models, giving a clean envelope around all anticipated implementations.

For non-MCA ARM, however, the lack of non-MCA processor implementations meant that experiment was not informative, and indeed the space of hypothetical microarchitectural implementations was not clear. Our discussions did not identify realistic mechanisms that are exactly as relaxed as the architectural intent, and in fact, the lack of MCA in the ARM architecture was motivated by a general desire for implementation freedom rather than specific microarchitectural designs. Broadly, non-MCA permits two hardware optimisations: (1) a shared, pre-cache store-buffer that allows early forwarding of data between a subset of the system threads, and (2) the ability to post snoop invalidations to other caches participating in a cache-coherence protocol without waiting for their acknowledgement. Whilst these optimisations may be worthwhile (or even necessary) in some contexts, for ARM there was a clear internal conclusion that these optimisations offer little benefit in ARM's context, partly because the ARM bus architecture (AMBA) has always been MCA. In this light, the greater flexibility was not considered to outweigh the complexities that non-MCA added in ARMv8.

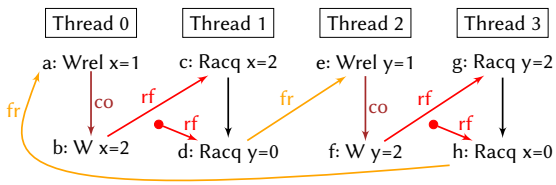
The rest of this section describes some of the issues we encountered during work to establish formal models for non-MCA ARMv8: refining the Flowing and POP models of Flur et al. [2016b, 2017], developing axiomatic models (within ARM), and trying to establish the correspondence between the two. The complexity this uncovered was part of the motivation for the shift to the new MCA ARMv8 model we address in the rest of the paper, where these issues disappear. The discussion may also be instructive for those developing other memory models.

MCA of store-release. The non-MCA ARMv8 documentation stated [ARM Ltd. 2016, B2.7.3] *"A Store-Release instruction is multicopy atomic when observed with a Load-Acquire instruction"*. When this line was discussed with ARM architects they explained that this would be achieved in practice by allowing a load-acquire read to be satisfied from a store-release write only after that write has propagated to all cores. In the Flowing model this is expressed by forbidding load-acquire reads from reading from store-release writes in a queue; they can only read from such writes in memory.

As intended, this forbids a version of IRIW where all the stores are store-release and all the loads are load-acquire (Test `IRIW+poaas+LL`). However, it unintentionally also forbids the variant where all the stores are store-release and just the first load of each thread is a load-acquire (`IRIW+poaps+LL`), which the original architectural intent would allow (as the MCA of the store-releases is not guaranteed for the regular loads). Our discussions did not identify a reasonable non-MCA microarchitectural mechanism that would forbid `IRIW+poaas+LL` and still allow `IRIW+poaps+LL`. This issue does not arise for MCA ARMv8: the architecture is now defined to be MCA for all memory accesses, irrespective of whether they are release/acquire or not, which forbids both these litmus tests.

Vanishing reads and acquire tokens. In actual implementations, a read request leaves no trace in the caches or buffers after it is satisfied. This could be matched in Flowing by removing the

read request from the queues. By itself, however, this would lead to behaviour that is unsound w.r.t. C/C++11. Consider a C/C++11 version of `IRIW` where all the loads are sequentially consistent (SC) and each writing thread performs two stores, the first one being SC and the second one relaxed (to the same location), with the loads reading from the latter. As those relaxed stores are the immediate modification-order successors of the SC store from the same thread, they form a release sequence [Batty et al. 2011; Becker 2011]. C/C++11 guarantees SC behaviour in this case, and therefore this execution is forbidden. The mapping of this litmus test to ARMv8 is below, using store-release and load-acquire instructions (the C and ARM versions are named `WW+RR+WW+RR+wsiscrlx+poscsc+wsiscrlx+poscsc.c` and `WW+RR+WW+RR+wsilp+poaa+wsilp+poaa`, in the supplementary material). As with the §2 example, the first reads of each thread can read from the intermediate-level queues. If those reads were removed at this point, nothing would prevent the following reads from flowing all the way to memory and reading from the initial state, making this behaviour allowed. To avoid this, the flowing model introduced "acquire tokens": it kept the satisfied read in the queue, but below the write it was satisfied from [Flur et al. 2016a, Satisfy read from segment]. This gave the extensional behaviour originally intended by the architects, but it did not abstract from any known or imagined microarchitecturally reasonable mechanism, reducing confidence in the model, and highlighting the difficulty of establishing good architecture models unguided by plausible implementation practice.



Cumulativity for dmb ld and dmb st. A similar vanishing-reads issue arose for dmb ld. The dmb sy barrier orders arbitrary memory actions that are thread-locally before and after it, and it also has cumulativity properties. Among other things, it orders stores that the barrier thread reads from, before the dmb sy, w.r.t. stores that it performs after (A-cumulativity), and orders stores (from the barrier thread) that occur before the barrier w.r.t. stores (from any thread) that appear logically after the barrier (B-cumulativity). For regular accesses, this worked straightforwardly in the non-MCA architecture.

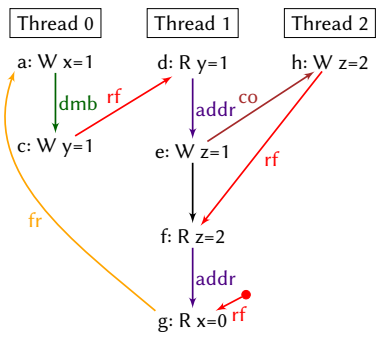
For dmb ld, on the other hand, it was unclear from the ARM documentation which cumulative properties dmb ld enforced. Following discussion, ARM decided (for the earlier non-MCA architecture) that dmb ld should not be cumulative in any way. Instead, it was made similar to an address dependency, which is reflected in Flowing by making dmb ld handled entirely in the thread semantics (not submitted to the queues), thus eliminating the issue of vanishing reads for dmb ld. In the non-MCA architecture, this weakening would make dmb ld unsound as a mapping of the C11 fence-acquire. Initially, ARM believed that this should be fixed by changing either the mapping or the semantics of the C/C++11 fence-acquire. Instead, shifting to MCA for all memory accesses strengthens dmb ld and makes the mapping sound.

For dmb st, the ARM conclusion for the non-MCA architecture was to make dmb st be only B-cumulative, again following extended discussion. That was straightforward to express in Flowing. Again, in the MCA model the question is moot.

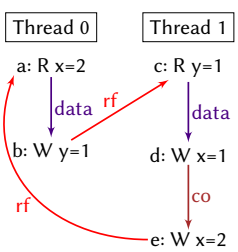
Partially satisfying reads. Mixed-size accesses also present unique challenges in the non-MCA context. To guarantee single-copy atomicity in the Flowing model (a read that reads from any byte of a write cannot also read from any coherence-hidden byte of another write), when a read is partially satisfied by a write with a smaller memory footprint, the unsatisfied part of the read and the write are reordered and never allowed to be reordered with each other again. This reordering has no analogy in real microarchitecture, as current implementations typically do not allow partial

satisfaction of reads, yet the architecture intentionally did not forbid it. In fact, one of the processor implementation errata mentioned by Flur et al. [2017], #855830 of ARM Ltd. [2017], was found by noticing the issue for a previous version of the Flowing model, conjecturing that it would be a likely hardware bug, and trying the corresponding test on hardware.

Big detour. One can create order between memory accesses from the same thread using intervening memory accesses from another thread. For example, the litmus test on the left is a version of MP with the order in Thread 0 enforced by a barrier and the order in Thread 1 enforced by a combination of address dependencies and a "detour" [Alglave et al. 2014] via Thread 2. In the general case the detour might involve dependencies or other means of creating order in the third thread, including nested detours. 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: it meant that one could not easily incrementally build up a picture of the relations; instead one had to iterate to a (global) fixed point.



Write subsumption. Another key difficulty in axiomatic modelling for the non-MCA ARMv8 architecture was handling *write subsumption*. In that architecture, and in the Flowing and POP models of Flur et al. [2016b, 2017], a write was allowed to propagate to other threads before program-order preceding writes to the same location were committed: the earlier write could be *subsumed* by the later ones. For example, the illustrated variant of `LB`, with data dependencies and with a second non-dependent write on the right thread (`LB+data+data-wsi`), was allowed. This was handled in the axiomatic model with a variant of the usual coherence order, but again proved challenging to explain.



4 THE MCA ARMv8 ARCHITECTURE

The most notable change in the revised ARMv8 architecture is the switch to an MCA model.[2] This makes tests such as `IRIW+addrs` and `WRC+addrs` forbidden, together with all but the last test named in §3, and means that the concept of barrier cumulativity is no longer required. The remaining relaxed-memory effects (which are still subtle) are all due to thread-local out-of-order and speculative execution and thread-local buffering. Writes can still be visible to program-order later reads on the same thread (even on speculative paths, as in `PPOCA`) before becoming visible to other threads.

The second change is a strengthening of the order preserved in the threads: the revised ARMv8 memory model forbids write subsumption, and thereby creates order from a data-dependent write to program-order successor writes to the same address, forbidding also the last test from §3.

[2]Terminology: the ARM documentation calls this an "other multicopy-atomic" model (to distinguish from the original definition of multicopy atomicity by Collier [1992], which required atomicity w.r.t. reads by all threads, including the writer thread), while here we simply say "multicopy-atomic", for consistency with the more recent normal usage.

The third change is concerned with the definition of dependencies (read-to-read address and control+isb/isync dependencies[3], 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 optimisations in value computations (such as x AND $0 = 0$): *"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."* [ARM Ltd. 2016, 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 nondeterministic semantics. Recent work on C/C++ concurrency illustrates the difficulties of defining envelopes around such optimisations [Batty et al. 2015; Pichon-Pharabod and Sewell 2016; Kang et al. 2017]. The revised ARMv8 architecture makes no such distinction.

Since no production hardware implementations have exploited the relative weakness of early ARMv8 compared with the revised architecture, all these changes have been "backported" — they apply to already-existing ARMv8 hardware, and the ARMv8.0 architecture has been revised.

5 INTRODUCING THE TWO FORMAL MODELS

We capture the ARM architectural intent for MCA ARMv8 with two formal models, an operational model and an axiomatic model, that are proved equivalent (for the instruction-set features covered by both, and for finite executions). Both have their advantages, as we discuss below, so having such an equivalence result is the ideal situation, not often achieved.

The operational model builds on Flur et al. [2017, 2016b]; Gray et al. [2015]; Sarkar et al. [2012, 2011]. The MCA architecture could be captured using the Flowing model of Flur et al. [2017, 2016b], as recalled in §2, by choosing a flat topology in which all hardware threads are siblings. However, that would have internal redundancy, as some relaxed behaviours could be exhibited either by out-of-order execution within a thread or by reordering in its queue. One of the key observations of the current work is that the necessary relaxations can (with some care) be made admissible in the thread semantics alone. That means the hierarchical interconnect can be replaced by a memory which essentially just records the most recent write to each location. This simplifies both the operational model itself and the relationship between it and the axiomatic model.

The operational model supports incremental construction of arbitrary allowed executions. It is a nondeterministic labelled transition system, of states and transitions between them, expressed as a function that computes the possible transitions of each state. That makes it usable in several ways. Given an initial program, memory, and register state:

(1) one can explore the possible executions interactively;
(2) for small litmus-test examples, one can calculate the set of all allowed outcomes by an exhaustive search; and
(3) for larger examples, e.g. for small instances of concurrent algorithms, one can explore longer randomly-chosen single traces, or sets thereof.

Importantly, the operational model aims to explain and define the envelope of architecturally allowed relaxed-memory behaviour in an "abstract microarchitectural" style, with relaxed-memory behaviour arising in the model in essentially the same way that it does in hardware implementations. (Of course, an actual hardware implementation that extensionally conforms to the model could be less aggressive than the model, or it could be more aggressive internally, as some are, so long as it does not extensionally allow more programmer-visible behaviours.) Accordingly, the model makes out-of-order and speculative execution explicit, as these are essentially what give rise to the relaxed

[3]a control dependency followed by an isb (ARMv8)/isync (POWER) instruction barrier

behaviour allowed by the ARM architecture and observed on ARM hardware. Similarly following conventional hardware practice, the model does not involve value speculation (except of computed branch addresses, where our tool necessarily approximates), or let writes be speculatively visible to other threads.

Given that, the model aims to abstract as much as possible from the lower-level microarchitectural detail of actual implementations, while keeping the intended envelope of programmer-visible behaviour: out-of-order and speculative execution are explicit, but in terms of an abstract tree or list of instruction instances, not a concrete pipeline; the instruction semantics are specified in terms of execution of their instruction-description pseudocode from the ARM manual (manually or automatically translated from the ASL used there into the Sail language of Gray et al. [2015] and [Flur et al. 2016b]), rather than hardware operations; the observable effects of shadow registers and register renaming are included by having instruction instances read from their predecessors rather than having explicit register files and renaming; there are no explicit buffers (of writes, read requests, or barriers); and there is no cache hierarchy or cache protocol.

Supporting incremental construction of executions with explicit speculative execution requires the operational model to roll-back and restart instructions in certain circumstances, and to discard speculative paths that turn out to be mistaken. That adds some complexity, but keeps the clear correspondence with microarchitecture. (One could define an operational model that just deadlocks in such cases, effectively a roll-back to the initial state; that would support exhaustive search but be awkward in interactive exploration.) The fact that the operational model constructs candidate executions incrementally means that it can execute the pseudocode of each instruction instance on concrete values; it does not require symbolic execution of the ASL or Sail metalanguage used for instruction description (modulo calculation of some register and memory footprint information).

Following Flur et al. [2017], the operational model supports mixed-size and misaligned accesses: aligned memory accesses of different sizes (e.g. 1, 2, 4, 8, or 16 bytes) can overlap, single load and store instructions can (e.g. if misaligned) give rise to many memory read and write accesses, and register accesses can touch different parts of registers. This also adds complexity, but it is necessary for defining the semantics of real code, rather than just non-mixed-size litmus tests.

The abstract-microarchitectural nature of the model has made it malleable in the face of architectural changes over time, and the fact that we have to define the model behaviour in any model state has helped identify previously unconsidered subtleties, including some of those of §3.

The axiomatic model, on the other hand, is expressed as a predicate that defines which candidate complete executions are permitted, where a candidate execution is a graph of memory-access and barrier events, related by program-order, dependency, coherence, and atomicity relations. This model is expressed using the `herd` tool of Alglave and Maranget [2017]; Alglave et al. [2014].

The axiomatic model does not describe the incremental construction of partial executions; only the successful complete executions. For small litmus-test examples, one can calculate the set of all allowed outcomes using `herd`, analogous to (2) above, but it does not support the interactive or single-random-trace exploration of (1) and (3). Historically the axiomatic computation of allowed outcomes has been considerably faster than the exhaustive memoised search used for operational models, which is especially combinatorially challenging for non-MCA models. The shift to MCA has made this less of an issue, as both computations are fast enough to allow easy iteration during model development: the operational model 55 minutes while `herd` takes 7 minutes, for our entire non-mixed test suite of around 9 000 tests. In contrast, the non-MCA Flowing and POP models took days, with some tests still not terminating.

The two models differ in their temporal granularity: the axiomatic model has a single event for each memory access instruction, while in the operational model each instruction gives rise

to many transitions. For example, for a load instruction there are transitions for the instruction fetch/decode, for register reads and writes, for instruction-internal steps of the ASL/Sail pseudocode, for announcing the memory footprint (when that becomes provisionally known), for each individual read being satisfied (from memory or by forwarding), for the point at which all reads are satisfied, and for the point when the instruction is finished and cannot be restarted. One can single-step through each of these, if desired, but all the transitions except those for read-satisfaction can be taken eagerly, as soon as they are enabled, without excluding other behaviour; this is important for performance of the exhaustive search.

This structure of the operational model is desirable in several ways. It arises from the choice of an abstract microarchitectural style, which gives a clear operational intuition and correspondence to plausible hardware implementations; from incremental construction of partial possibly-speculative executions; from the handling of mixed-size; and from the integration with the instruction-description ASL/Sail semantics. But it does introduce complexity with respect to the axiomatic model, which defines the thread-local memory ordering between (non-mixed-size) memory accesses in a more direct way. Axiomatic modelling is much simplified in the MCA setting, just as operational modelling is: those thread-local ordering properties are now the only things the axiomatic model has to handle, without non-MCA propagation and cumulativity effects. Pleasingly, the main part of the axiomatic model fits on a single page. A priori one may find it surprising that one can define a satisfactory axiomatic model in terms of just a single event for each memory access instruction; this paper may help explain why that is so, by showing how certain causal dependencies that involve many steps in the operational model are collapsed to access-to-access relational constraints in the axiomatic model.

Note also that the axiomatic model relies on primitive notions of address, data, and control dependencies (hard-coded into the `herd` implementation), while in the operational model these arise from the instruction semantics and from the thread-semantics treatment of register accesses and conditional branches (taking them as primitive simplifies the concurrency model, but is less foundational). The axiomatic model can then be expressed using just inductive definitions of binary relations, e.g. as in the `herd` model definition language, whereas the enumeration of operational model transitions requires a more expressive metalanguage — it is essentially a typed pure terminating functional program (or higher-order-logic definition), in `lem` [Mulligan et al. 2014].

The two models differ in the features they cover, for historical reasons: they both cover memory accesses (regular, release/acquire, and exclusives) and barriers (dmb sy, dmb st, and dmb ld), but the axiomatic model does not cover mixed-size accesses, while the operational model does not cover the more relaxed form of load-acquire (LDAPR) introduced in ARMv8.3, or the atomic read-modify-write instructions introduced in ARMv8.1. Neither covers load/store-pair instructions, exceptions, interrupts, floating-point, vector instructions, MMU behaviour, self-modifying code, or other systems aspects. The operational model is integrated with a more authoritative instruction semantics, derived from the ARM-internal specification, for a larger fragment of the ISA (including, e.g., all the instructions used in compiling simple C code), while the axiomatic model currently relies on the smaller and more ad hoc instruction semantics built into `herd`. Currently, `herd`'s instruction semantics differs from that of the operational model in that the success bit of a successful store-exclusive introduces address/data/control dependencies from the load-exclusive it is paired with. This does not match the intention of the ARMv8 architecture where the success bit is not supposed to introduce any dependencies.

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

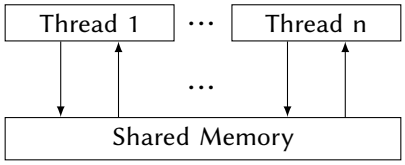
6 AN OPERATIONAL MODEL FOR MCA ARMv8

In the following we give a prose description of our formal operational model for multicopy-atomic ARMv8, the *Flat model*. For brevity we omit load/store-exclusive instructions; the full description, including those, is in the supplementary material, as is the formal model. The full description also includes remarks about the difficulties in modelling ARMv8 exclusive instructions operationally (due to the architecture's intention to allow store exclusives to promise success/failure very early).

To see the model's behaviour on example programs, using the web interface, we give a step-by-step execution of how the model allows the `PPOCA` test as an example (press 'Enter' repeatedly to follow the steps): https://is.gd/qSJM6H.

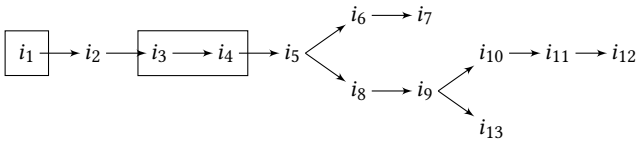
The operational model is expressed as a state machine, with states that are an abstract representation of hardware machine states. We first introduce the model states and transitions informally.

Model states. A model state consists just of a shared memory and a tuple of thread model states:



The shared memory state effectively just records the most recent write to each location, together with some additional data for exclusives.

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. For example, below we show a thread model state with instruction instances $i_1, \ldots, i_{13}$, and the program-order-successor relation between them. Three of those (i_1, i_3, and i_4, boxed) have been finished; the remainder are non-finished.



Non-finished instruction instances can be subject to restart, e.g. if they depend on an out-of-order or speculative read that turns out to be unsound. The finished instances are not necessarily contiguous: in the example, i_3 and i_4 are finished even though i_2 is not, which can only happen if they are sufficiently independent. Instruction instances i_5 and i_9 are conditional branches for which the thread has fetched multiple possible successors. When a conditional branch is finished, any untaken alternative paths are discarded, and instruction instances that follow (in program order) a non-finished conditional branch cannot be finished until that conditional branch is. One can choose whether or not to allow simultaneous exploration of multiple successors of a conditional branch (as shown above); this does not affect the set of allowed outcomes.

The intra-instruction behaviour of a single instruction can largely be treated as sequential (but not atomic) execution of its ASL/Sail pseudocode. Each instruction instance state includes a pseudocode execution state, which one can think of as a representation of the pseudocode control state, pseudocode call stack, and local variable values. 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. 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 §6.4, 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 instance that writes to that register.
- ∘ Register write
- ∘ Pseudocode internal step: this covers ASL/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 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 (when all its reads are entirely satisfied): 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, with their footprints: 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 (when its writes are all propagated): At this point all writes have been propagated to memory, and the instruction pseudocode can continue execution.

Barrier instructions:

- ∘ Commit barrier

6.1 Intra-instruction Pseudocode Execution

To link the model transitions introduced above to the execution of the instructions an interface is needed between Sail and the rest of the concurrency model. For each instruction instance this

intra-instruction semantics is expressed as a state machine, essentially running the instruction pseudocode, where each pseudocode execution state is a request of one of the following forms:

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

Each of these states is a suspended computation with a request for an action or input from the concurrency model and, except in the case of Done, a continuation for the remaining execution.

Here memory values are lists of bytes, addresses are 64-bit numbers, read and write kinds identify whether they are regular, exclusive, and/or release/acquire operations, register names identify a register and slice thereof (start and end bit indices), and the continuations describe how the instruction instance will continue for any value that might be provided by the surrounding memory model. This largely follows Gray et al. [2015, §2.2], except that memory writes are split into two steps, Write_ea and Write_memv. We ensure these are paired in the pseudocode, but there may be other steps between them: it is observable that the Write_ea can occur before the value to be written is determined, because the potential memory footprint of the instruction becomes provisionally known then.

We ensure 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. The semantics therefore has to know the register write footprint of each instruction instance, which it calculates when the instruction instance is created. We ensure in the pseudocode that each instruction does exactly one register write to each bit of its register footprint, and also that instructions do not do register reads from their own register writes. In some cases, but not in the fragment of ARM that we cover 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). This has to be carefully handled in order not to create unintentional strength. First, for some instructions we need to ensure that the pseudocode is in the maximally liberal order, e.g. to allow early computed-address register writebacks before the corresponding memory write. Leaving load-pair aside (which we do not cover), and the treatment of the multiple reads or writes that can be associated with a single load or store instruction (which we do), we have not so far needed other intra-instruction concurrency. Second, the model has to be able to know when a register read value can no longer change (i.e. due to instruction restart). We approximate 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 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 we cover this approximation is exact.

We express the pseudocode execution semantics in two ways: a definitional interpreter for Sail [Gray et al. 2015], with an exhaustive symbolic mode to (re)calculate an instruction's memory and register footprints, and as a shallow embedding, translating Sail into directly executable code, with separate hand-written definitions of the footprint functions. The two are essentially equivalent: the first lets one small-step through the pseudocode interactively, while the second is more efficient and should be more convenient for proof.

6.2 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 branch; or a 'simple' instruction.
- *regs_in*, the set of input *reg_name*s, as statically determined;
- *regs_out*, the output *reg_name*s, 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.
- 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 we refer to a write or read request without mentioning the kind of request we mean 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*.

6.3 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.

6.4 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*:
  (a) for a non-branch/jump instruction, the successor instruction address (*i.program_loc+4*);
  (b) for an instruction that has performed a write to the program counter register (_PC), the value that was written;
  (c) for a conditional branch, either the successor address or the branch target address[4]; or
  (d) for a jump to an address which is not yet determined, any address (this is approximated in our 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 initialized 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 we assume a fixed program rather than modelling fetches with memory reads; we do not model 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. . .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) $[^{\text{dmb ld/}}_{\text{dmb st}}]$ all po-previous dmb ld instructions are finished;
(3) $[^{\text{release/}}_{\text{acquire}}]$ if *i* is a load-acquire, all po-previous store-releases are finished; and
(4) $[^{\text{release/}}_{\text{acquire}}]$ all non-finished po-previous load-acquire instructions are entirely satisfied.

Let *wss* be the maximal set of unpropagated write slices from store instruction instances po-before *i*, that overlap with the unsatisfied slices of *r*, and which are not superseded by intervening stores that are either propagated or read from by this thread. 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.

[4]In AArch64, all the conditional branch instructions have statically determined addresses.

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.

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.

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 (i.e., 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*.

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. . .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 initiate 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 §6.5);
(2) all po-previous conditional branch instructions are finished;

(3) all po-previous dmb sy and isb instructions are finished;
(4) [dmb ld/ dmb st] all po-previous dmb ld instructions are finished;
(5) [release/ acquire] all po-previous load-acquire instructions are finished;
(6) all po-previous store instructions have initiated and so have non-empty *mem_writes*;
(7) [release/ acquire] if i is a store-release, all po-previous memory access instructions are finished;
(8) [dmb ld/ dmb st] all po-previous dmb st instructions are finished;
(9) all po-previous memory access instructions have a fully determined memory footprint; and
(10) all po-previous load instructions have initiated and so have non-empty *mem_reads*.

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) all memory writes of po-previous store instructions that overlap w have already propagated
(2) all read requests of po-previous load instructions that overlap with w have already been satisfied, and the load instruction is non-restartable (see §6.5); and
(3) all read requests satisfied by forwarding w are entirely satisfied.

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; and
(3) update the memory with w.

Complete store instruction (when its writes are all propagated). A store instruction i in 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 state *Plain next_state* where *next_state* is *Barrier(barrier_kind, next_state′)* can be committed if:

(1) all po-previous conditional branch instructions are finished;
(2) [dmb ld/ dmb st] if i is a dmb ld instruction, all po-previous load instructions are finished;
(3) [dmb ld/ dmb st] 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: there, barriers committed in program-order and potentially re-ordered in the storage subsystem. Here the thread subsystem is weakened to subsume the re-ordering of Flowing's (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:
  (a) all po-previous dmb sy and isb instructions are finished;
  (b) [dmb ld/ dmb st] all po-previous dmb ld instructions are finished;
  (c) [release/ acquire] all po-previous load-acquire instructions are finished;
  (d) 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 *cfp′* be the complement of *cfp* in the memory footprint of *i*. If *cfp′* is not empty:
    (i) *i′* has a fully determined memory footprint;
    (ii) *i′* has no unpropagated memory write that overlaps with *cfp′*; and
    (iii) If *i′* is a load with a memory footprint that overlaps with *cfp′*, then all the read requests of *i′* that overlap with *cfp′* are satisfied and *i′* can not be restarted (see §6.5).
    Here a memory write is called fixed if it is the write of a store instruction that has fully determined data.
  (e) [release/ acquire] 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 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*.

6.5 Auxiliary Definitions

Fully determined. 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 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 we use 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 *s* 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).

7 AN AXIOMATIC MODEL FOR MCA ARMv8

Unlike the operational model, where the behaviour of a program is an emergent property of the model's execution, the axiomatic model is a predicate over candidate complete executions, each consisting of its memory and barrier events and binary relations over them, largely as in Alglave et al. [2010, 2014]: *program order* (`po`), *reads-from* (`rf`), *coherence order* (`co`), and *read-modify-write* (`rmw`). Here `rmw` contains the successful read/write-exclusive pairs. The `rf`, `co`, and derived `fr` relations are each subdivided into their "internal" (same-thread) and "external" (different-thread) parts, suffixed `i` and `e` respectively. A candidate execution also specifies the address (`addr`), data (`data`), and control (`ctrl`) dependency relations.

The ARMv8 architecture reference manual defines (in text) additional relations in terms of these and uses them to define a predicate capturing when a given candidate execution is architecturally permitted. The main part of the corresponding `herd` model, by Deacon [2016], is below.

```
let ca = fr | co                                  (* Coherence-after *)
let obs = rfe | fre | coe                         (* Observed-by *)
let dob = addr | data                             (* Dependency-ordered-before *)
      | ctrl; [W]
      | (ctrl | (addr; po)); [ISB]; po; [R]
      | addr; po; [W]
      | (ctrl | data); coi
      | (addr | data); rfi
let aob = rmw                                     (* Atomic-ordered-before *)
      | [range(rmw)]; rfi; [A | Q]
let bob = po; [dmb.full]; po                      (* Barrier-ordered-before *)
      | [L]; po; [A]
      | [R]; po; [dmb.ld]; po
      | [A | Q]; po
      | [W]; po; [dmb.st]; po; [W]
      | po; [L]
      | po; [L]; coi
let ob = (obs | dob | aob | bob)+                 (* Ordered-before *)

acyclic po-loc | ca | rf    as internal
```

```
irreflexive ob                as external
empty rmw & (fre; coe)        as atomic
```

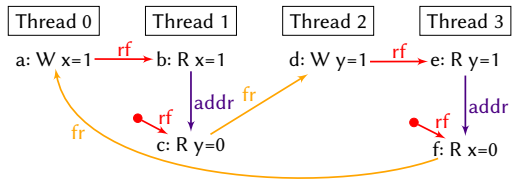
Here `|`, `&`, `;`, and `+` are relational union, intersection, composition, and transitive closure, `[W]`, `[R]`, `[L]`, `[A]` and `[Q]` are the identity relations over all write, read, release, acquire and acquirePC events respectively, and `[ISB]`, `[dmb.full]`, `[dmb.st]`, `[dmb.ld]` are the singleton identity on the corresponding barrier events; `po-loc` relates two same-address memory accesses in program order.

Per-location relations. The *coherence-after* relation is the union of the *from-reads* and *coherence-order* relations and represents the order in which writes must propagate with respect to other accesses to the same location as a result of coherence.

The *observed-by* relation is the union of the external subsets of the *reads-from*, *from-reads* and *coherence order* relations and represents inter-thread communication through shared variables. It is therefore not possible for an access to be *observed-by* another access from the same thread.

Intra-thread ordering relations. Intra-thread order can be established using instruction dependencies, atomic instructions or barrier instructions (including load-acquire/store-release instructions). These are described by the *dependency-ordered-before* (`dob`), *atomic-ordered-before* (`aob`) and *barrier-ordered-before* (`bob`) relations respectively, which together form an externally-ordered subset of program order. For example, the `[L];po;[A]` case of the `bob` relation describes a store-release appearing in program order before a load-acquire, which is required to be observed in order by all threads as a result of the sequentially-consistent semantics of ARMv8's release/acquire instructions.

Relational constraints. The ARMv8 architecture places two constraints on the relations of a candidate execution: the *internal visibility requirement* requires uniprocessor ("SC per location") semantics and the *external visibility requirement* requires that intra-thread ordering respects multi-copy atomicity. The axiomatic model expresses the former by prohibiting cycles in the transitive closure of program order, coherence-after and reads-from for a given location, and the latter by requiring that the transitive closure of the intra-thread ordering relations and observed-by forms a partial order, known as *ordered-before* (`ob`). There is also an additional constraint to ensure that the atomicity of a successful load/store-exclusive pair or an ARMv8.1 atomic RMW instruction (e.g. cas) is enforced by requiring that the atomic read is not related to the atomic write by an external *from-reads*; *coherence-order* sequence, although this is handled separately by the architecture. To illustrate the definition, recall the previous `IRIW+addrs` example. This non-multicopy-atomic behaviour exhibits a cycle of the shape `rfe;addr;fre;rfe;addr;fre`, which can be expressed as a cycle in `ob`: (a) is *observed-by* (b); (b) is *dependency-ordered-before* (c); (c) is *observed-by* (d); (d) is *observed-by* (e); (e) is *dependency-ordered-before* (f) and (f) is *observed-by* (a). The `external` axiom requires `ob`'s acyclicity, and so this candidate execution is forbidden.



8 PROOF OF EQUIVALENCE

The basic problem in relating the Flat operational model of §6 and the MCA ARMv8 axiomatic model of §7 [ARM Ltd. 2017; Deacon 2016] (ARMv8-ax for short) is the mismatch between the events the two models refer to: where 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 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 we have to define 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, we 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.

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

- $(E, E') \in \mathtt{po}_t$ for two events (E, E') if E is before E' in the instruction tree after t (in a final state all instructions are finished, and since finished branch instructions have only one successor in the tree, the tree is a linear order of instructions)
- $(W, W') \in \mathtt{co}_t$ for two writes W and W' if W propagates to memory before W' in t
- $(W, R) \in \mathtt{rf}_t$ for a write W and a read R if, in the final satisfy-read transition of R in t, the read R is satisfied by W (we say "final" since if the trace t involves a restart of R it might be satisfied multiple times)
- $(RE, WE) \in \mathtt{rmw}_t$ for a read-exclusive RE and write-exclusive WE if in t the write-exclusive WE is successfully paired with the read-exclusive RE.

Given this definition we can state:

Theorem 8.1. *Let $x = (po, co, rf, rmw)$ be a finite candidate execution of ARMv8-axiomatic for a given program P. The execution x is valid under ARMv8-axiomatic if and only if there exists a valid finite trace t of Flat-operational for the program P such that $(po_t, co_t, rf_t, rmw_t) = (po, co, rf, rmw)$.*

We now give the high-level ideas of the proof for both directions of the implication.

8.1 "If": Flat Operational Behaviour Included in ARMv8 Axiomatic

To show that the candidate execution $x = (\mathtt{po}_t, \mathtt{co}_t, \mathtt{rf}_t, \mathtt{rmw}_t)$ induced by an arbitrary valid Flat trace t is allowed by ARMv8-ax we need to prove that any such x satisfies the three axioms: `external`, `internal`, and `atomic`. We first consider `external` (which can be regarded as the main axiom). We assume that for any trace of Flat there is an equivalent one that has no restarts, which intuitively holds because restarts are down-closed with respect to the information flow in the model, and hence, without loss of generality, that t involves no restarts.

The basic idea of this direction of the proof is that, interpreting `ob` as a relation between Flat transitions using the correspondence given above, we show that t (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 t must be acyclic as well. To illustrate the proof of `ob` $\subseteq t$ consider an edge $e \in$ `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 t 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 t, 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 t has no restarts there can be no such coherence violation.

8.2 "Only If": ARMv8 Axiomatic Behaviour Included in Flat Operational

The other direction of the proof is more difficult. We have to show that, given a candidate execution x allowed by ARMv8-ax, there exists a Flat trace t that induces x. We would like to do this by defining t 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 t: we have constructed the trace t' for a prefix of S and need to extend it for the next element B in S. The trace t' 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 t'.

To address this problem we bridge the gap between the Flat operational and ARMv8-ax models 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 t 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, we have:

(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 t' we know all such R and R' are satisfied; and in combination with other edges of `Order` we 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 `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` is also one in `ob`.

9 EXPERIMENTAL VALIDATION

We experimentally validate that the two models include the (non-errata) behaviour observed on certain production ARMv8 hardware, and that the two models allow the same behaviour as each other — giving additional evidence for the above theorem. This uses a test suite of 11310 litmus tests, consisting mostly of families of tests systematically generated using `diy` [Alglave and Maranget 2017], together with some hand-written tests; it includes the tests used in Flur et al. [2016b, 2017]. Of these, 2369 are mixed-size tests and so are only used for the operational 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 which is interpreted inconsistently by the tools. The experimentally observed behaviour on hardware for each test is produced using the `litmus` tool [Alglave et al. 2011], running on ARMv8 implementations including: LG H955 phone (Qualcomm Snapdragon810 SoC, ARM Cortex-A57/A53 CPU, quad+quad core — using the A53 cores); iPad Air 2 (Apple A8X SoC/CPU, three-core); iPhone 7 (Apple A10 SoC/CPU, dual+dual core); Google Nexus 9 tablet (Nvidia Tegra K1 SoC, Nvidia Denver CPU, dual-core); Open-Q 820 development kit (Qualcomm Snapdragon 820 SoC, Qualcomm Krait CPU, quad-core); ODROID-C2 development board (Amlogic S905 SoC, ARM Cortex-A53 CPU, quad-core); Samsung Galaxy S8 phone (Exynos 9 (8895) SoC, Exynos M2/Cortex-A53 CPU, quad+quad core) (not all tests have been run on all implementations). We run both formal models on the same tests to compute the set of

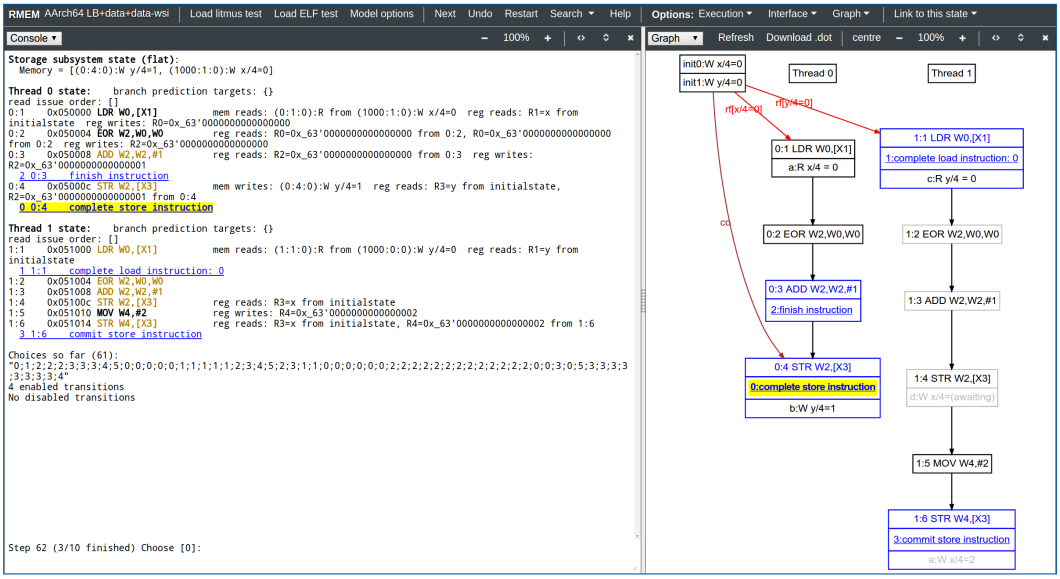


Fig. 1. An `rmem` web interface screenshot, running the §3 test `LB+data+data-wsi` interactively. This shows a Flat operational model state, in text and graphically, with clickable transitions in blue.

model-allowed final states for each test: using `rmem` of §10 for Flat and `herd` for ARMv8-axiomatic. All hardware-allowed behaviours are allowed by ARMv8-axiomatic and Flat. As usual, not all model-allowed behaviours are observed, as particular processor implementations typically do not exploit all of the architecturally allowed looseness. For all tests the two models allow the same set of final states, with the exception of six tests due to the aforementioned handling of dependencies from the result register of a store exclusive in `herd` (§5). For the tests that both models run on, on a 80 core POWER8 (TN71-BP012) machine, `rmem` (Flat-operational) terminates in less than 55 minutes (cumulative runtime of 12 hours), and `herd` (ARMv8-axiomatic) terminates in less than 7 minutes (cumulative runtime of 27 minutes).

Additionally, we use the `memalloy` tool of Wickerson et al. [2017] to automatically compare ARMv8-axiomatic with the Flat-axiomatic model used in the equivalence proof: running `memalloy` for 120 hours finds no mismatches between the two axiomatic models of up to 12 events, also in agreement with the proof of equivalence. At an earlier stage Flat-axiomatic's structure was different: to closely match Flat-operational it distinguished between reads satisfied by forwarding and by storage, and existentially quantified over a partition of reads into those sets; John Wickerson adapted `memalloy` to support this quantification in the tool. Earlier versions of Flat-axiomatic were also defined mutually recursively. Since the `alloy` framework that `memalloy` builds on cannot handle recursive definitions, `memalloy` reported some false positives due to insufficient loop-unrolling.

10 THE RMEM TOOL

We integrated the Flat operational semantics into an exploration tool `rmem`, building on the `ppcmem` tool [Sarkar et al. 2011; Gray et al. 2015; Flur et al. 2016b, 2017]. It is available at http://www.cl.cam.ac.uk/~pes20/rmem/. This supports all three modes of use mentioned in §5: interactive exploration, exhaustive memoised search, and (new) random trace search.

We have significantly improved tool usability with a new web interface, shown in Fig. 1: the tool can show the current state and the enabled transitions of the operational model, in textual and

graphical form, and one can interactively click on transitions in either. The tool can be run entirely in the browser, for convenience. For this, the operational model is compiled from Lem to OCaml and thence to JavaScript, using `js_of_ocaml` [Vouillon and Balat 2014], and renders the interactive graphs using Viz.js [Daines 2017], a packaging of Graphviz [Gansner and North 2000] compiled to JavaScript using Emscripten [Zakai 2011]. The tool can also be run from the command line, for best performance. A library of litmus and ELF binary tests is provided, including those mentioned in this paper, as well as a "link to this state" function for easy sharing of examples (e.g. https://is.gd/Jd1kjR, the state shown in Fig. 1). To make it feasible to explore the behaviour of larger examples, e.g. concurrent C code compiled to standalone ELF binaries, we added basic debugger functionality: breakpoints, watchpoints, and support for some DWARF debug information, e.g. to show the names of local variables and the C source lines corresponding to machine instructions. All this required considerable UI engineering.

The tool lets one explore behaviour at many different granularities: one can single-step through (and display) the ASL/Sail pseudocode of an individual instruction instance, or one can take the transitions marked "eager" in §5 automatically as soon as they are enabled, or many points inbetween (e.g. one can choose whether register reads/writes should be taken automatically). Some transitions cannot be taken eagerly in general but it is sound not to explore arbitrary interleaving of these with other transitions. We optionally enable this optimisation, as well as optionally discarding traces which include restarted instructions (from incorrectly speculated reads) or discarded instructions (from incorrectly speculated branch targets).

These pruning options are also important for performance in exhaustive search.

As a further optimisation to prune away parts of the state space which do not contain "interesting" concurrent behaviour, we optionally record the memory locations which are accessed by each thread and derive from this an approximation of the set of program locations which access memory which is accessed by more than one thread. This allows us to also eagerly take memory transitions of instructions that only touch thread-local memory. When searching in this mode, we start with an empty approximation and repeat the search until a fixed point is reached, after which (in exhaustive mode) we can be sure to have explored all observable behaviours.

Finally, we implemented a simple loop count stopping condition based on hit counts of (branch, destination) pairs to allow searching on larger programs with potentially unbounded loops, without requiring manual unrolling, easing its potential application to unmodified real-world algorithms.

11 APPLYING RMEM TO A LINUX SPINLOCK

To demonstrate `rmem` in use on a production concurrent algorithms, not just artificial litmus tests, we tried it on a Linux kernel spinlock. [Linux contributors 2014; Howells et al. 2016]. Flur et al. [2016b] did some interactive exploration of a test with two threads contending for the same lock as a proof of concept, verifying the detectability of injected bugs, but were unable to feasibly perform automated analysis. Using `rmem` it is now straightforward to do exhaustive analysis of a finitised version of the same two-thread test case, taking 15 seconds, 3 minutes, and 18 minutes for one, two and three (manual) unrolls respectively to verify correctness (on a 2.60GHz Intel Core i5-3230M). We also re-verified the detection of bug-injected versions. Interactive exploration using the graphical UI, which now allows direct selection of transitions on the graph, is also much easier.

We also considered a previously discovered Linux kernel bug in a different part of the spinlock API, `spin_unlock_wait()`. A simplified litmus version of the test case is shown for brevity. This code uses mixed-size accesses, load-acquire-exclusive (`LDAXR`), load-acquire-exclusive halfword (`LDAXRH`) and store-exclusive (`STXR`) instructions, and is an example of a real-world bug previously discovered in the wild, making it a perfect test case for our exploration tool. The relevant memory

```
uint32_t lock = 0; uint64_t obj = 0, ptr = obj;
0:X0 = lock; 0:X10 = ptr; 0:X11 = obj;
1:X0 = lock; 1:X10 = ptr;
```

```
P0                                  P1
(* ptr = 0; smp_mb (); *)           (* spin_lock(&lock); *)
MOV X1, #0                          enq:
STR X1, [X10]     a: W ptr=0          LDAXR W1, [X0]      d:Re-a lock=0
DMB SY                                ADD   W2, W1, #16, LSL #12
                                      STXR  W3, W2, [X0]  e:Wexc lock=...
(* spin_unlock_wait(&lock); *)         CBNZ  W3, enq
chk:                                  EOR   W2, W1, W1, ROR #16
  LDR  W1, [X0]   b: R lock=0         CBZ   W2, outl
  AND  W2, W1, #0xffff                spinl:
  CMP  W2, W1, LSR #16                  LDAXRH W3, [X0]
  B.NE chk                              EOR    W2, W3, W1, LSR #16
                                        CBNZ   W2, spinl
(* smp_mb (); obj = 1; *)             outl:
DMB SY                                (* if (ptr) BUG_ON( *ptr ); *)
MOV X0, #1                            LDR X0, [X10]       f:R ptr=obj)
STR X0, [X11]     c: W obj=1          MOV X1, #0
                                      CBZ X0, end
                                      LDR X1, [X0]        g:R obj=1
end:                                  end:
```

accesses are bold, barriers are green, and branches creating control dependencies are purple. A lock `lock` protects a shared pointer `ptr` to a shared object `obj`. Thread `P0` will clear the pointer, wait for the lock to be free, and then process the object, which we represent by storing `1` to it. Thread `P1` is intended to observe `ptr` $\neq 0 \implies$ `obj` $= 0$, but the ARM architecture allows load `(f)` to be satisfied early, from the initial state, so that the branch around `(g)` will not be taken, regardless of what `(b)` reads, and so whether or not `(c)` commits, allowing `P1` to observe `ptr` $\neq 0 \wedge$ `obj` $= 1$ by `c` $\xrightarrow{\text{rf}}$ `g`.

Exhaustive exploration discovers the bug in an unrolled version in seconds (3 seconds, 67 seconds and 18 minutes for one to three unrolls), producing a sample counterexample execution that one can then replay interactively. Random single-trace execution of the non-unrolled code also readily finds the bug in approximately 0.07% of traces (perhaps explaining why it took so long to surface in the wild) which is repeatably reproducible by a 100000-trace search taking about 2 minutes. These times are all for the command-line version; in the browser the one-unroll exhaustive search takes 19s as opposed to the 3s above. One can run `rmem` to a state of the erroneous execution, just before `g` is satisfied from either the initial state or `c`, from https://is.gd/eDKHP8.

To remove the undesired reordering, one could insert a memory barrier into thread `P1` to enforce ordering between instructions `(e)` and `(f)`, but this has performance implications, preventing desirable speculation into the 'lock acquired' case. Deacon [2015] originally fixed the bug by replacing the regular load `b` with a load-acquire/store-release pair similar to instructions `(d)` and `(e)`, which suffices to remove the erroneous behaviour by properly allowing only one thread to successfully write-exclusive to `lock`, forcing a discard of any out-of-order satisfaction of `(f)` in executions which might otherwise allow `(f)` to read from the initial state and `c` $\xrightarrow{\text{rf}}$ `g`. We verified the correctness of loop-unrolled versions of the fixed implementation through exhaustive search.

We were also able to perform the same exhaustive and random searches, producing the same results in similar timeframes, on ELF binaries compiled from the Linux C/embedded assembly source with only minor modifications. This reduces both the effort required to adapt test cases for analysis and the potential for errors introduced in transcribing to litmus tests. (Independently, due to lack of clarity of the intended semantics for `spin_unlock_wait()`, McKenney [2017] is in the process of removing this API, in favour of simple taking of the lock or full read-copy-update.)

12 RELATED WORK

The most closely related work has already been mentioned. We point out in particular the axiomatic `herd` model for POWER and ARMv7 of Alglave et al. [2014]. Since their model has to cope with a non-MCA concurrency semantics, it is considerably more complex than ARMv8-axiomatic and requires three axioms in addition to the standard thread-coherence axiom. In the non-MCA setting,

unlike ARMv8-axiomatic, `hb` cannot include the edges `fre` and `coe`, whose composition with thread-local order must necessarily take partial propagation of writes into account. Flat-axiomatic's relations distinguish between read satisfaction and commitment in a similar way to theirs. Arvind and Maessen [2006] develop a procedure for enumerating possible outcomes of multicopy-atomic relaxed-memory programs that is parametric in some ordering choices.

In addition, several papers have proved the equivalence of operational and axiomatic relaxed memory models in one form or another. Focussing just on hardware models: Ahamad et al. [1995] axiomatically specify Causal memory and proves that an operational implementation thereof satisfies the axioms. Higham et al. [1998] formalise SPARC and a number of simpler memory models in both axiomatic and operational style, with proof of equivalence. Owens et al. [2009] define and prove equivalent an operational and an axiomatic concurrency model for x86-TSO; similarly Burckhardt and Musuvathi [2008, App. A] for TSO. Finally, Alglave et al. [2014] define axiomatic concurrency models for ARM and POWER in `herd`, and prove that the operational POWER model of Sarkar et al. [2011] satisfies the conditions of that axiomatic model, including an equivalence proof of the axiomatic POWER model and an operationalised version thereof.

Some care is needed with the terms "axiomatic" and "operational" here. The two kinds of model in the literature are typically at broadly similar levels of abstraction as those we consider here, with axiomatic models having one event per instruction and operational models supporting incremental execution in an abstract microarchitectural style, but neither property is necessarily true in general. One can have axiomatic models with finer-grain events, e.g. Mador-Haim et al. [2012] and Manerkar et al. [2015], and one can have operational models that are not microarchitectural in style or do not support effective incremental execution.

Other related work explores tool support for axiomatic memory models and their design: Mador-Haim et al. [2010], Wickerson et al. [2017], and Bornholt and Torlak [2017] develop tools to automatically find litmus tests that distinguish between memory models, with the last also targetting synthesis of models from a sketch and a set of litmus tests. Lustig et al. [2017] also synthesise litmus tests suites from model definitions, in the Alloy framework.

13 CONCLUSION

We have established operational and axiomatic models for the revised ARMv8 model, provably equivalent and with the latter adopted as part of the ARM architecture definition. Together with the substantial simplifications of the revised ARMv8 model (multicopy atomicity, absence of write subsumption, and clear dependency definition), and the ability to use the `rmem` tool to explore the behaviour of real software with respect to the full envelope of behaviour allowed by the architecture (not just that of particular hardware implementations), this puts ARM concurrency on a more solid and straightforward foundation. As the caveats in the Introduction indicate, this is surely not the last word on the ARMv8 model, but it should aid hardware designers and concurrent programmers. It should also ease future work on reasoning and verification for ARM, as these become considerably simpler in the context of the revised architecture and our formal models thereof.

ACKNOWLEDGMENTS

We thank Richard Grisenthwaite for discussions of the ARM architecture, Luc Maranget for assistance with tests and testing, Alastair Reid for comments on a draft, Alan Stern for interesting discussions, and John Wickerson for assistance with `memalloy`. This work was partly funded by the EPSRC Programme Grant *REMS: Rigorous Engineering for Mainstream Systems*, EP/K008528/1, EPSRC grant *C3: Scalable & Verified Shared Memory via Consistency-directed Cache Coherence* EP/M027317/1 (Sarkar), an ARM iCASE award (Pulte), and an EPSRC IAA KTF (Flur).

REFERENCES

Allon Adir, Hagit Attiya, and Gil Shurek. 2003. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Trans. Parallel Distrib. Syst.* 14, 5 (2003), 502–515. https://doi.org/10.1109/TPDS.2003.1199067

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241

Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proc. DAMP 2009*.

Jade Alglave and Luc Maranget. 2017. http://diy.inria.fr/doc/index.html. (April 2017).

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Proc. CAV*.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: running tests against hardware. In *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, Berlin, Heidelberg, 41–44. http://dl.acm.org/citation.cfm?id=1987389.1987395

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM TOPLAS* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

ARM Ltd. 2016. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd. ARM DDI 0487A.k_iss10775 (ID092916).

ARM Ltd. 2017. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd. ARM DDI 0487B.a (ID033117).

ARM Ltd. 2017. ARM Processor Cortex-A53 MPCore Product Revision r0 Software Developers Errata Notice. http://infocenter.arm.com/help/topic/com.arm.doc.epm048406/Cortex_A53_MPCore_Software_Developers_Errata_Notice.pdf. (May 2017).

Arvind Arvind and Jan-Willem Maessen. 2006. Memory Model = Instruction Reordering + Store Atomicity. *SIGARCH Comput. Archit. News* 34, 2 (May 2006), 29–40. https://doi.org/10.1145/1150019.1136489

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. 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*. 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proc. POPL*.

Pete Becker (Ed.). 2011. *Programming Languages — C++*. ISO/IEC 14882:2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

James Bornholt and Emina Torlak. 2017. 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)*. ACM, New York, NY, USA, 467–481. https://doi.org/10.1145/3062341.3062353

Sebastian Burckhardt and Madanlal Musuvathi. 2008. *Effective Program Verification for Relaxed Memory Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 107–120. https://doi.org/10.1007/978-3-540-70545-1_12

Nathan Chong and Samin Ishtiaq. 2008. Reasoning about the ARM weakly consistent memory model. In *MSPC*.

William W. Collier. 1992. *Reasoning about parallel architectures*. Prentice Hall, Englewood Cliffs. http://opac.inria.fr/record=b1105256

F. Corella, J. M. Stone, and C. M. Barton. 1993. *A formal specification of the PowerPC shared memory architecture*. Technical Report RC18638. IBM.

Mike Daines. 2017. Viz.js, a hack to put Graphviz on the web. https://github.com/mdaines/viz.js/. (2017).

Will Deacon. 2015. Linux commit 'arm64: spinlock: serialise spin_unlock_wait against concurrent lockers'. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d86b8da04dfa. (2015).

Will Deacon. 2016. The ARMv8 Application Level Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat. (2016).

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016a. The Flowing and POP Models (supplementary material for Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA). (2016). http://www.cl.cam.ac.uk/~sf502/popl16/model_full.pdf.

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016b. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. In *POPL 2017: The 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*. https://doi.org/10.1145/2535838.2535841

Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience* 30, 11 (2000), 1203–1233. http://www.graphviz.org/.

Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture*.

Lisa Higham, Jalal Kawash, and Nathaly Verwaal. 1998. *Weak Memory Consistency Models. Part I: Definitions and Comparisons*. Technical Report. Department of Computer Science, The University of Calgary. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.2126.

David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. 2016. Documentation/memory-barriers.txt. https://www.kernel.org/doc/Documentation/memory-barriers.txt. (2016).

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

Linux contributors. 2014. Documentation/locking/spinlocks.txt. https://www.kernel.org/doc/Documentation/locking/spinlocks.txt. (2014).

Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. 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 '17)*. ACM, New York, NY, USA, 661–675. https://doi.org/10.1145/3037697.3037723

Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. 2010. Generating Litmus Tests for Contrasting Memory Consistency Models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 273–287. https://doi.org/10.1007/978-3-642-14295-6_26

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Proceedings of CAV 2012: the 24th International Conference on Computer Aided Verification*. 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: Using μHb Graphs to Verify the Coherence-consistency Interface. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/2830772.2830782

Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. Draft available from http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf. (2012).

Paul E McKenney. 2017. Remove spin_unlock_wait(). (Jun 2017). https://lkml.org/lkml/2017/6/29/967

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*. 175–188. https://doi.org/10.1145/2628136.2628143

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*. 391–407.

Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of POPL*.

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*. 311–322. https://doi.org/10.1145/2254064.2254102

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*. 175–186. https://doi.org/10.1145/1993498.1993520

Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. 2009. The Semantics of x86-CC Multiprocessor Machine Code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 379–391. https://doi.org/10.1145/1594834.1480929

Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972. https://doi.org/10.1002/spe.2187 https://ocsigen.org/js_of_ocaml/.

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 190–204. https://doi.org/10.1145/3009837.3009838

Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/2048147.2048224 https://github.com/kripken/emscripten.