Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory

Jade Alglave, Luc Maranget, Michael Tautschnig

Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory

JADE ALGLAVE, University College London
LUC MARANGET, INRIA
MICHAEL TAUTSCHNIG, Queen Mary University of London

7

We propose an axiomatic generic framework for modelling weak memory. We show how to instantiate this framework for Sequential Consistency (SC), Total Store Order (TSO), C++ restricted to release-acquire atomics, and Power. For Power, we compare our model to a preceding operational model in which we found a flaw. To do so, we define an operational model that we show equivalent to our axiomatic model.

We also propose a model for ARM. Our testing on this architecture revealed a behaviour later acknowledged as a bug by ARM, and more recently, 31 additional anomalies.

We offer a new simulation tool, called herd, which allows the user to specify the model of his choice in a concise way. Given a specification of a model, the tool becomes a simulator for that model. The tool relies on an axiomatic description; this choice allows us to outperform all previous simulation tools. Additionally, we confirm that verification time is vastly improved, in the case of bounded model checking.

Finally, we put our models in perspective, in the light of empirical data obtained by analysing the C and C++ code of a Debian Linux distribution. We present our new analysis tool, called mole, which explores a piece of code to find the weak memory idioms that it uses.

Categories and Subject Descriptors: B.3.2 [**Shared Memory**]; C.0 [**Hardware/Software Interfaces**]

General Terms: Theory, Experimentation, Verification

Additional Key Words and Phrases: Concurrency, weak memory models, software verification

ACM Reference Format:
Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36, 2, Article 7 (June 2014), 74 pages.
DOI: http://dx.doi.org/10.1145/2627752

1. INTRODUCTION

There is a joke where a physicist and a mathematician are asked to herd cats. The physicist starts with an infinitely large pen, which he reduces until it is of reasonable diameter yet contains all of the cats. The mathematician builds a fence around himself and declares the outside to be the inside. Defining memory models is akin to herding cats: both the physicist's and mathematician's attitudes are tempting, but neither can go without the other.

Recent years have seen many formalisations of memory models emerge both on the hardware and software sides (e.g., see; Adir et al. [2003], Arvind and Maessen [2006],

Authors' addresses: J. Alglave, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, United Kingdom; email: J.Alglave@cs.ucl.ac.uk; L. Maranget, INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay France; email: Luc.Maranget@inria.fr; M. Tautschnig, School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, London E1 4NS, United Kingdom; email: michael.tautschnig@qmul.ac.uk.

DOI: http://dx.doi.org/10.1145/2627752

Boehm and Adve [2008], Chong and Ishtiaq [2008], Boudol and Petri [2009], Sarkar et al. [2009, 2011, 2012], Alglave et al. [2009, 2010, 2012], Batty et al. [2011], Alglave [2012], Mador-Haim et al. [2012], and Boudol et al. [2012]). Yet we feel the need for more work in the area of *defining* models. There are several reasons for this.

On the hardware side, all existing models of Power (some of which we list in Table I) have some flaws (see Section 2). This calls for reinvestigating the model for the sake of repairing it of course, but for several other reasons as well, which we explain below.

One particularly important reason is that Power underpins C++'s *atomic* concurrency features [Boehm and Adve 2008; Batty et al. 2011; Sarkar et al. 2012]: implementability on Power has had a major influence on the design of the C++ model. Thus, modelling flaws in Power could affect C++.

Another important reason is that, at present, the code in the wild (see our experiments in Section 9 on release 7.1 of the Debian Linux distribution) still does not use the C++ atomics. Thus, we believe that programmers have to rely on what the hardware does, which requires descriptive models of the hardware.

On the software side, recent work shows that the C++ model allows behaviours that break modular reasoning (see the *satisfaction cycles* issue in Batty et al. [2013]), whereas Power does not, since it prevents *out of thin air* values (see Section 4). Moreover, C++ requires the irreflexivity of a certain relation (see the HBvsMO axiom in Batty et al. [2013]), whereas Power offers a stronger acyclicity guarantee, as we show in this article.

Ideally, we believe that these models would benefit from stating principles that underpin weak memory as a whole, not just one particular architecture or language. Not only would it be aesthetically pleasing, but it would allow more informed decisions on the design of high-level memory models, ease the conception and proofs of compilation schemes, and allow the reusability of simulation and verification techniques from one model to another.

Models roughly fall into two classes: *operational* and *axiomatic*. Operational models, such as the Power model of Sarkar et al. [2011], are abstractions of actual machines, composed of idealised hardware components such as buffers and queues. Axiomatic models, such as the C++ model of Batty et al. [2011], distinguish allowed behaviours from forbidden behaviours, usually by constraining various relations on memory accesses.

We now list a few criteria that we believe our models should meet; we do not claim to be exhaustive, nor do we claim that the present work fully meets all of them, although we discuss in the conclusion of this article to what extent it does. Rather, we see this list as enunciating some wishes for works on weak memory (including ours of course), and more generally weak consistency, as can be found for example in distributed systems:

—*Stylistic proximity of models*, whether hardware (e.g., x86, Power, or ARM) or software (e.g., C++), would permit the statement of general principles spanning several models of weak memory. It should be easier to find principles common to Power and C++, amongst others, if their respective models were described in the same terms.
—*Rigour* is not our only criterion: for example, all recent Power models enjoy a considerable amount of rigour yet are still somewhat flawed.
—*Concision* of the model seems crucial to us: we want to specify a model concisely to grasp it and modify it rapidly, without needing to dive into a staggering number of definitions.

One could tend towards photorealistic models and account for each and every detail of the machine. We find that operational models in general have such traits, although some do more than others. For example, we find that the work of Sarkar et al. [2011, 2012] is too close to the hardware, and, perhaps paradoxically, too precise, to be easily amenable to pedagogy, automated reasoning, and verification. Although we do

recognise the effort and the value of this work, without which we would not have been able to build the present work, we believe that we need models that are more *rationally descriptive* (as coined by Richard Bornat). We go back to what we mean by "rational" at the end of our introduction.

—*Efficient simulation and verification* have not been the focus of previous modelling work, except for Mador-Haim et al. [2010] and Alglave et al. [2013a, 2013b]. These works show that simulation [Mador-Haim et al. 2010] and verification [Alglave et al. 2013b] (for bounded model checking) can be orders of magnitude faster when it relies on axiomatic models rather than operational ones.

Yet operational models are often considered more intuitive than axiomatic models. Perhaps the only tenable solution to this dilemma is to propose both styles in tandem and show their equivalence. As exposed in Hoare and Lauer [1974], "a single formal definition is unlikely to be equally acceptable to both implementor and user, and [. . .] at least two definitions are required, a constructive one [. . .] for the implementor, and an implicit one for the user [. . .]."

—*Soundness w.r.t. hardware* is mandatory regardless of the modelling style. Ideally, one would prove the soundness of a model w.r.t. the formal description of the hardware, such as at RTL [Gordon 2002]. However, we do not have access to these data because of commercial confidentiality. To overcome this issue, some previous work has involved experimental testing of hardware (e.g., see Collier [1992], Sarkar et al. [2009, 2011], Alglave et al. [2012], and Mador-Haim et al. [2012]), with increasing thoroughness over time.[1]

A credible model cannot forbid behaviours exhibited on hardware, unless the hardware itself is flawed. Thus, models should be extensively tested against hardware, and retested regularly: this is how we found flaws in the model of Sarkar et al. [2011] (see Section 2). Yet, we find the experimental flaws themselves to be less of an issue than the fact that the model does not seem to be easily fixable.

—*Adaptability* of the model—that is, setting the model in a flexible formalism—seems crucial if we want stable models. By stable, we mean that even though we might need to change parameters to account for an experimental flaw, the general shape of the model—its principles—should not need to change.

Testing (as extensive as it may be) cannot be sufficient, as it cannot guarantee that a behaviour that was not observed yet might not be triggered in the future. Thus, one needs some guarantee of *completeness* of the model.

—*Being in accord with the architectural intent* might give some guarantee of completeness. We should try to create models that respect or take inspiration from the architects' intents. This is one of the great strengths of the model of Sarkar et al. [2011]. However, this cannot be the only criterion, as the experimental flaws of [Sarkar et al. 2011] show. Indeed, the architects' intents might not be as formal as one might need in a model, two intents might be contradictory, or an architect might not realise all the consequences of a given design.

—*Accounting for what programmers do* seems a sensible criterion. One cannot derive a model from programming patterns, as some of these patterns might rely on erroneous understanding of the hardware. Yet to some extent, these patterns should reflect part of the architectural intent, considering that systems programmers or compiler writers communicate relatively closely with hardware designers.

Crucially, we have access to open source code, as opposed to the chips' designs. Thus, we can analyse the code and derive some common programming patterns from it.

[1]Throughout this article, we refer to online material to justify our experimental claims; the reader should take these as bibliography items and refer to them when details are needed.

Rational models is what we advocate here. We believe that a model should allow a rational explanation of what programmers can rely on. We believe that by balancing all of the preceding criteria, one can provide such a model. This is what we set out to do in this article.

By rational, we mean the following: we think that we, as architects, semanticists, programmers, and compiler writers, are to understand concurrent programs. Moreover, we find that to do so, we are to understand some particular patterns (e.g., the message passing pattern given in Figure 1, the very classical store buffering pattern given in Figure 14, or the controversial load buffering pattern given in Figure 7). We believe that by being able to explain a handful of patterns, one should be able to generalise the explanation and thus be able to understand a great deal of weak memory.

To make this claim formal and precise, we propose a generic model of weak memory, in axiomatic style. Each of our four axioms has a few canonical examples, which should be enough to understand the full generality of the axiom. For example, we believe that our NO THIN AIR axiom is fully explained by the load buffering pattern of Figure 7. Similarly, our OBSERVATION axiom is fully explained by the message passing, write to read causality, and Power ISA2 patterns of Figures 8, 11, and 12, respectively.

On the modelling front, our main stylistic choices and contributions are as follows: to model the propagation of a given store instruction to several different threads, we use only one memory event per instruction (see Section 4), instead of several subevents (e.g., one per thread, as one would do in Itanium [Intel Corp. 2002] or in the Power model of Mador-Haim et al. [2012]). We observe that this choice makes simulation much faster (see Section 8).

To account for the complexity of write propagation, we introduce the novel notion of *propagation order*. This notion is instrumental in describing the semantics of fences, for instance, as well as the subtle interplay between fences and coherence (see Section 4).

We deliberately try to keep our models concise, as we aim at describing them as simple text files that one can use as input to an automated tool (e.g., a simulation tool or a verification tool). We note that we are able to describe IBM Power in less than a page (see Figure 38).

Outline. We present related works in Section 2. After a tutorial on axiomatic models (Section 3), we describe our new generic model of weak memory in Section 4 and show how to instantiate it to describe Sequential Consistency (SC) [Lamport 1979], Total Store Order (TSO) (used in Sparc [SPARC International Inc. 1994] and x86's [Owens et al. 2009] architectures), and C++ restricted to release-acquire atomics.

Section 5 presents examples of the semantics of instructions, which are necessary to understand Section 6, where we explain how to instantiate our model to describe Power and ARMv7. We compare formally our Power model to the one of Sarkar et al. [2011] in Section 7. To do so, we define an operational model that we show equivalent to our axiomatic model.

We then present our experiments on Power and ARM hardware in Section 8, detailing the anomalies that we observed on ARM hardware. We also describe our new herd simulator, which allows the user to specify the model of his choice in a concise way. Given a specification of a model, the tool becomes a simulator for that model.

Additionally, we demonstrate in the same section that our model is suited for verification by implementing it in the bounded model checker cbmc [Clarke et al. 2004] and comparing it with the previously implemented models of Alglave et al. [2012] and Mador-Haim et al. [2012].

In Section 9, we present our analysis tool mole, which explores a piece of code to find the weak memory behaviours that it contains. We detail the data gathered by mole by analysing the C and C++ code in a Debian Linux distribution. This gives us a pragmatic

perspective on the models that we present in Section 4. Additionally, mole may be used by programmers to identify areas of their code that may be (unwittingly) affected by weak memory, or by static analysis tools to identify areas where more fine-grained analysis may be required.

Online Companion Material. We provide the source and documentation of herd at `http://diy.inria.fr/herd`. We provide all our experimental reports w.r.t. hardware at `http://diy.inria.fr/cats`. We provide our Coq scripts at `http://www0.cs.ucl.ac.uk/staff/j.alglave/cats/`. We provide the source and documentation of mole at `http://diy.inria.fr/mole`, as well as our experimental reports w.r.t. release 7.1 of the Debian Linux distribution.

2. RELATED WORK

Our introduction echoes position papers by Burckhardt and Musuvathi [2008], Zappa Nardelli et al. [2009], Adve and Boehm [2010], and Boehm and Adve [2012], which all formulate criteria, prescriptions, or wishes as to how memory models should be defined.

Looking for general principles of weak memory, one might look at the hardware documentation: we cite Alpha [Compaq Computer Corp. 2002], ARM [ARM Ltd. 2010], Intel [Intel Corp. 2009], Itanium [Intel Corp. 2002], IBM Power [IBM Corp. 2009], and Sparc [SPARC International Inc. 1992, 1994]. Ancestors of our SC PER LOCATION and NO THIN AIR axioms (see Section 4) appear notably in Sparc's and Alpha's documentations.

We also refer the reader to work on modelling particular instances of weak memory, such as ARM [Chong and Ishtiaq 2008], TSO [Boudol and Petri 2009] or x86 [Sarkar et al. 2009; Owens et al. 2009] (which indeed happens to implement a TSO model; see Owens et al. [2009]), C++ [Boehm and Adve 2008; Batty et al. 2011], or Java [Manson et al. 2005; Cenciarelli et al. 2007]. We revisit Power at the end of this section.

In the rest of this article, we write TSO for Total Store Order, implemented in Sparc TSO [SPARC International Inc. 1994] and Intel x86 [Owens et al. 2009]. We write PSO for Partial Store Order and RMO for Relaxed Memory Order, two other Sparc execution models. We write Power for IBM Power [IBM Corp. 2009].

Collier [1992], Neiger [2000], and Adve and Gharachorloo [1996] have provided general overviews of weak memory, but in a less formal style than one might prefer.

Steinke and Nutt [2004] provide a unified framework to describe consistency models. They choose to express their models in terms of the view order of each processor and describe instances of their framework, amongst them several classical models such as PRAM [Lipton and Sandberg 1988] and Cache Consistency [Goodman 1989].

Rational models appear in Arvind and Maessen's work [2006], aiming at weak memory in general but applied only to TSO, and in that of Batty et al. [2013] for C++. Interestingly, the work of Burckhardt et al. [2013, 2014] on distributed systems follows this trend.

Some works on weak memory provide simulation tools: the ppcmem tool of Sarkar et al. [2011] and that of Boudol et al. [2012] implement their respective operational model of Power, whilst the cppmem tool of Batty et al. [2011] enumerates the axiomatic executions of the associated C++ model. The tool of Mador-Haim et al. [2012] does the same for their axiomatic model of Power. MemSAT has an emphasis towards the Java memory model [Torlak et al. 2010], whilst Nemos focuses on classical models such as SC or causal consistency [Yang et al. 2004], and TSOTool handles TSO [Hangal et al. 2004].

To some extent, decidability and verification papers [Gopalakrishnan et al. 2004; Burckhardt et al. 2007; Atig et al. 2010, 2011, 2012; Bouajjani et al. 2011, 2013; Kuperstein et al. 2010, 2011; Liu et al. 2012; Abdulla et al. 2012, 2013] do provide some general principles about weak memory, although we find them less directly applicable

Table I. A Decade of Power Models in Order of Publication

model	style	comments
Adir et al. [2003]	axiomatic	based on discussion with IBM architects; precumulative barriers
Alglave et al. [2009]	axiomatic	based on documentation; not tested on h/w
Alglave et al. [2012] and Alglave and Maranget [2011]	single-event axiomatic	based on extensive testing; semantics of lwsync stronger than Sarkar et al. [2011] on r+lwsync+sync, weaker on mp+lwsync+addr
Sarkar et al. [2011, 2012]	operational	based on discussion with IBM architects and extensive testing; flawed w.r.t. Power h/w on, e.g., mp+lwsync+addr-po-detour (see Fig. 36 and `http://diy.inria.fr/cats/pldi-power/#lessvs`) and ARM h/w on, e.g., mp+dmb+fri-rfi-ctrlisb (see `http://diy.inria.fr/cats/pldi-arm/#lessvs`)
Mador-Haim et al. [2012]	multi-event axiomatic	thought to be equivalent to Sarkar et al. [2011] but not experimentally on, e.g., mp+lwsync+addr-po-detour (see `http://diy.inria.fr/cats/cav-power`)
Boudol et al. [2012]	operational	semantics of lwsync stronger than Sarkar et al. [2011] on, e.g., r+lwsync+sync
Alglave et al. [2013a]	operational	equivalent to Alglave et al. [2012]

to programming than semantics work. Unlike our work, most of them are restricted to TSO, or its siblings PSO and RMO, or theoretical models.

Notable exceptions are Alglave et al. [2013a, 2013b], which use the generic model of Alglave et al. [2012]. The present article inherits some of the concepts developed in Alglave et al. [2012]: it adopts the same style in describing executions of programs and pursues the same goal of defining a generic model of weak memory. Moreover, we adapt the cbmc tool of Alglave et al. [2013b] to our new models and reuse the diy testing tool of Alglave et al. [2012] to conduct our experiments against hardware.

Yet we emphasise that the model that we present here is quite different from the model of Alglave et al. [2012], despite the stylistic similarities: in particular Alglave et al. [2012] did not have a distinction between the OBSERVATION and PROPAGATION axioms (see Section 4), which were somewhat merged into the *global happens-before* notion of Alglave et al. [2012].

A decade of Power models is presented in Table I. Earlier work (omitted for brevity) accounted for outdated versions of the architecture. For example in 2003, Adir et al. described an axiomatic model [Adir et al. 2003], "developed through [. . .] discussions with the PowerPC architects," with outdated noncumulative barriers, following the pre-PPC 1.09 PowerPC architecture.

Later, we refer to particular weak memory behaviours that serve as test cases for distinguishing different memory architectures. These behaviours are embodied by litmus tests, with standardised names in the style of Sarkar et al. [2011], for example, mp+lwsync+addr. All of these behaviours will appear in the rest of the article so that the novice reader can refer to them after a first read-through. We explain the naming convention in Section 4.

In 2009, Alglave et al. proposed an axiomatic model [Alglave et al. 2009], but this was not compared to actual hardware. In 2010, they provided another axiomatic model [Alglave et al. 2010, 2012] as part of a generic framework. This model is based on extensive and systematic testing. It appears to be sound w.r.t. Power hardware, but its semantics for lwsync cannot guarantee the mp+lwsync+addr behaviour (see Figure 8) and allows the r+lwsync+sync behaviour (see Figure 16), both of which clearly go

against the architectural intent (see Sarkar et al. [2011]). This model (and the generic framework to which it belongs) has a provably equivalent operational counterpart [Alglave et al. 2013a].

In 2011, Sarkar et al. [2011, 2012] proposed an operational model in collaboration with an IBM designer, which might be taken to account for the architectural intent. Yet, we found this model to forbid several behaviours observed on Power hardware (e.g., mp+lwsync+addr-po-detour, see Figure 36 and http://diy.inria.fr/cats/pldi-power/#lessvs). Moreover, although this model was not presented as a model for ARM, it was thought to be a suitable candidate. Yet, it forbids behaviours (e.g., mp+dmb+fri-rfi-ctrlisb, see Figure 32 and `http://diy.inria.fr/cats/pldi-arm/#lessvs`) that are observable on ARM machines and are claimed to be desirable features by ARM designers.

In 2012, Mador-Haim et al. [2012] proposed an axiomatic model, thought to be equivalent to the one of Sarkar et al. [2011]. Yet, this model does not forbid the behaviour of mp+lwsync+addr-po-detour (see `http://diy.inria.fr/cats/cav-power`), which is a counterexample to the proof of equivalence appearing in Mador-Haim et al. [2012]. The model of Mador-Haim et al. [2012] also suffers from the same experimental flaw w.r.t. ARM hardware as the model of Sarkar et al. [2011].

More fundamentally, the model of Mador-Haim et al. [2012] uses several write events to represent the propagation of one memory store to several different threads, which in effect mimics the operational transitions of the model of Sarkar et al. [2011]. We refer to this style as multievent axiomatic, as opposed to single-event axiomatic (e.g., as in Alglave et al. [2010, 2012]), where there is only one event to represent the propagation of a given store instruction. Our experiments (see Section 8) show that this choice impairs the simulation time by up to a factor of 10.

Later in 2012, Boudol et al. [2012] proposed an operational model where the semantics of lwsync is stronger than the architectural intent on, for example, r+lwsync+sync (like that of Alglave et al. [2010]).

3. PREAMBLE ON AXIOMATIC MODELS

Next, we give a brief presentation of axiomatic models in general. The expert reader might want to skip this section.

Axiomatic models are usually defined in three stages. First, an *instruction semantics* maps each instruction to some mathematical objects. This allows us to define the control-flow semantics of a multithreaded program. Second, we build a set of *candidate executions* from this control-flow semantics: each candidate execution represents one particular dataflow of the program—that is, which communications might happen between the different threads of our program. Third, a *constraint specification* decides which candidate executions are valid or not.

We now explain these concepts in a way that we hope to be intuitive. Later in this article, we give the constraint specification part of our model in Section 4 and an outline of the instruction semantics in Section 5.

Multithreaded Programs. Multithreaded programs, such as the one given in Figure 1, give one sequence of instructions per thread. Instructions can come from a given assembly language instruction set, such as Power ISA, or be pseudocode instructions, as is the case in Figure 1.

In Figure 1, we have two threads, T_0 and T_1, in parallel. These two threads communicate via the two memory locations x and y, which hold the value 0 initially. On T_0, we have a store of value 1 into memory location x, followed in program order by a store of value 1 into memory location y. On T_1, we have a load of the contents of memory location y into register r_1, followed in program order by a load of the contents of memory

mp

initially x=0; y=0

T_0	T_1
(a) x ← 1	(c) r1 ← y
(b) y ← 1	(d) r2 ← x

Fig. 1. A multithreaded program implementing a message passing pattern.

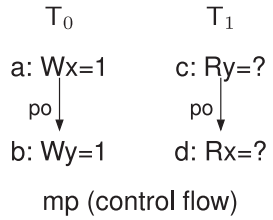


Fig. 2. Control-flow semantics for the message passing pattern of Figure 1.

location x into register r_2. Memory locations, for example x and y, are shared by the two threads, whereas the registers are private to the thread holding them, here T_1.

The snippet in Figure 1 is at the heart of a message passing (mp) pattern, where T_0 would write some data into memory location x, then set a flag in y. T_1 would then check if it has the flag, then read the data in x.

Control-Flow Semantics. The instruction semantics, in our case, translates instructions into events, which represent, for example, memory or register accesses (i.e., reads and writes from and to memory or registers), branching decisions, or fences.

Consider Figure 2: we give a possible control-flow semantics to the program in Figure 1. To do so, we proceed as follows: each store instruction (e.g., x ← 1 on T_0) corresponds to a write event specifying a memory location and a value (e.g., Wx=1). Each load instruction (e.g., r1 ← y on T_1) corresponds to a read event specifying a memory location and a undetermined value (e.g., Ry=?). Note that the memory locations of the events are determined by the program text, as well as the values of the writes. For reads, the values will be determined in the next stage.

Additionally, we also have implicit write events Wx=0 and Wy=0 representing the initial state of x and y that we do not depict here.

The instruction semantics also defines relations over these events, representing, for example, the program order within a thread, or address, data, or control dependencies from one memory access to the other, via computations over register values.

Thus, in Figure 3, we also give the program order relation, written po, which lifts the order in which instructions have been written to the level of events. For example, the two stores on T_0 in Figure 1 have been written in program order, thus their corresponding events Wx=1 and Wy=1 are related by po in Figure 2.

We are now at a stage where we have, given a program such as the one in Figure 1, several event graphs, such as the one in Figure 2. Each graph gives a set of events representing accesses to memory and registers; the program order between these events, including branching decisions; and the dependencies.

Data-Flow Semantics. The purpose of introducing dataflow is to define which communications, or interferences, might happen between the different threads of our program. To do so, we need to define two relations over memory events: the read-from relation rf, and the coherence order co.

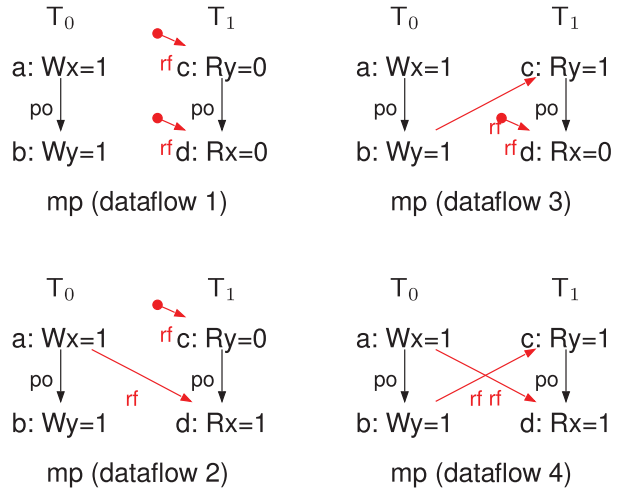


Fig. 3. One possible dataflow semantics for the control-flow semantics given in Figure 2.

The read-from relation rf describes, for any given read, from which write this read could have taken its value. A read-from arrow with no source, as in the top left of Figure 3, corresponds to reading from the initial state.

For example, in Figure 3, consider the drawing at the bottom left-most corner. The read c from y takes its value from the initial state, and hence reads the value 0. The read d from x takes its value from the update a of x by T_0, and hence reads the value 1.

The coherence order gives the order in which all of the memory writes to a given location have hit that location in memory. For example, in Figure 3, the initial write to x (not depicted) hits the memory before the write a on T_0, by convention, and hence the two writes are ordered in coherence.

We are now at a stage where we have, given a program such as the one in Figure 1, an event graph as given by the control-flow semantics (see Figure 2), and several read-from relations and coherence orders describing possible communications across threads (see Figure 3). In Figure 3, we do not display any coherence order, because there is only one write per location.

Note that for a given control-flow semantics, there could be several suitable dataflow semantics, if, for example, there were several writes to x with value 1 in our example: in that case, there would be two possible read-froms to give a value to the read of x on T_1.

Each such object (see Figure 3), which gathers events, program order, dependencies, read-from, and coherence, is called a *candidate execution*. As one can see in Figure 3, there can be more than one candidate execution for a given program.

Constraint Specification. For each candidate execution, the constraint specification part of our model decides whether this candidate represents a valid execution or not.

Traditionally, such specifications are in terms of acyclicity or irreflexivity of various combinations of the relations over events given by the candidate execution. This means that the model would reject a candidate execution if this candidate contains a cycle amongst a certain relation defined in the constraint specification.

For example, in Figure 3, the constraints for describing Lamport's SC [Lamport 1979] (see also Section 4.7) would rule out the right top-most candidate execution because the read from x on T_1 reads from the initial state, whereas the read of y on T_1 has observed the update of y by T_0.

4. A MODEL OF WEAK MEMORY

We present our axiomatic model and show its SC and TSO instances. We also explain how to instantiate our model to produce C++ R-A—that is, the fragment of C++ restricted to the use of release-acquire atomics. Section 6 presents Power.

The inputs to our model are candidate executions of a given multithreaded program. Candidate executions can be ruled out by the four axioms of our model, given later in Figure 5: SC PER LOCATION, NO THIN AIR, OBSERVATION, and PROPAGATION.

4.1. Preliminaries

Before explaining each of these axioms, we define a few preliminary notions.

Conventions. In this article, we use several notations that rely on relations and orders. We denote the transitive (resp. reflexive-transitive) closure of a relation r as r^+ (resp. r^*). We write $\mathsf{r}_1;\mathsf{r}_2$ for the sequential composition of two relations r_1 and r_2 (i.e., $(x,y) \in (\mathsf{r}_1;\mathsf{r}_2) \triangleq \exists z.(x,z) \in \mathsf{r}_1 \wedge (z,y) \in \mathsf{r}_2$). We write irreflexive($\mathsf{r}$) to express the irreflexivity of r (i.e., $\neg(\exists x.(x,x) \in \mathsf{r})$). We write acyclic($\mathsf{r}$) to express its acyclicity (i.e., the irreflexivity of its transitive closure: $\neg(\exists x.(x,x) \in \mathsf{r}^+)$).

A *partial order* is a relation r that is transitive (i.e., $\mathsf{r} = \mathsf{r}^+$) and irreflexive. Note that this entails that r is also acyclic. A *total order* is a partial order r defined over a set $\mathbb{S}$ that enjoys the totality property: $\forall x \neq y \in \mathbb{S}.(x,y) \in \mathsf{r} \vee (y,x) \in \mathsf{r}$.

Executions are tuples $(\mathbb{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co})$, which consist of a set of events $\mathbb{E}$, giving a semantics to the instructions, and three relations over events: po, rf, and co that we explain in this section.

Events consist of a unique identifier (in this article, we use lowercase letters, such as a), the thread holding the corresponding instruction (e.g., T_0), the line number or program counter of the instruction, and an action.

Actions are of several kinds, which we detail in the course of this article. For now, we only consider read and write events relative to memory locations. For example, for the location x, we can have a read of the value 0 noted $\mathrm{R}x = 0$, or a write of the value 1, noted $\mathrm{W}x = 1$. We write proc(e) for the thread holding the event e and addr(e) for its address, or memory location.

Given a candidate execution, the events are determined by the program's instruction semantics—we give examples in Section 5.

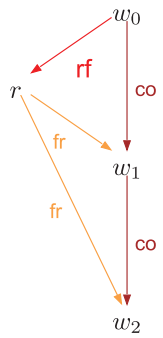
Given a set of events, we write $\mathsf{WR}, \mathsf{WW}, \mathsf{RR}, \mathsf{RW}$ for the set of write-read, write-write, read-read, and read-write pairs, respectively. For example, $(w,r) \in \mathsf{WR}$ means that w is a write and r a read. We write $\mathsf{po} \cap \mathsf{WR}$ for the write-read pairs in program order, and $\mathsf{po} \backslash \mathsf{WR}$ for all pairs in program order except the write-read pairs.

Relations over Events. The program order po lifts the order in which instructions have been written in the program to the level of events. The program order is a total order over the memory events of a given thread, but does not order events from different threads. Note that program order unrolls loops and determines the branches taken.

The read-from rf links a read from a register or a memory location to a unique write to the same register or location. The value of the read must be equal to the one of the write. We write rfe (external read-from) when the events related by rf belong to distinct threads—that is, $(w,r) \in \mathsf{rfe} \triangleq (w,r) \in \mathsf{rf} \wedge \mathrm{proc}(w) \neq \mathrm{proc}(r)$. We write rfi for internal read-from, when the events belong to the same thread.

The coherence order co totally orders writes to the same memory location. We write coi (resp. coe) for internal (resp. external) coherence.

We derive the from-read fr from the read-from rf and the coherence co, as follows:



That is, a read r is in fr with a write w_1 (resp. w_2) if r reads from a write w_0 such that w_0 is in the coherence order before w_1 (resp. w_2). We write fri (resp. fre) for the internal (resp. external) from-read.

We gather all communications in com $\triangleq$ co $\cup$ rf $\cup$ fr. We give a glossary of all the relations that we describe in this section in Table II. For each relation, we give its notation, its name in English, the directions (i.e., write W or read R) of the source, and target of the relation (column "dirns"), where to find it in the text (column "reference"), and an informal prose description. Additionally, in the column "nature," we give a taxonomy of our relations: are they fundamental execution relations (e.g., po, rf), architectural relations (e.g., ppo), or derived (e.g., fr, hb)?

Reading Notes. We refer to orderings of events w.r.t. several relations. To avoid ambiguity, given a relation r, we say that an event e_1 is *r-before* another event e_2 (or e_1 is an *r-predecessor* of e_1, or e_2 is *r-after* e_1, or e_2 is *r-subsequent*, etc.) when $(e_1, e_2) \in$ r.

Next, we present several examples of executions in the style of Sarkar et al. [2011]. We depict the events of a given thread vertically to represent the program order and the communications by arrows labelled with the corresponding relation. Figure 4 shows a classic message passing (mp) example.

This example is a communication pattern involving two memory locations x and y: x is a message, and y a flag to signal to the other thread that it can access the message.

T_0 writes the value 1 to memory at location x (see the event a). In program order after a (hence the po arrow between a and b), we have a write of value 1 to memory at location y. T_1 reads from y (see the event c). In the particular execution shown here, this read takes its value from the write b by T_0, hence the rf arrow between b and c. In program order after c, we have a read from location x. In this execution, we suppose that this event d reads from the initial state (not depicted), which by convention sets the values in all memory locations and registers to 0. This is the reason why the read d has the value 0. This initial write to x is, by convention, co-before the write a of x by T_0, and therefore we have an fr arrow between d and a.

Note that in the following discussion, even if we do not always depict all of the program order, a program order edge is always implied between each pair of events ordered vertically below a thread id (e.g., T_0).

Convention for Naming Tests. We refer to tests following the same convention as in Sarkar et al. [2011]. We roughly have two flavours of names: classical names, which are abbreviations of classical litmus test names appearing in the literature, and systematic names, which describe the accesses occurring on each thread of a test.

Table II. Glossary of Relations

notation	name	nature	dirns	reference	description
po	program order	execution	any, any	§Relations over events	instruction order lifted to events
rf	read-from	execution	WR	§Relations over events	links a write w to a read r taking its value from w
co	coherence	execution	WW	§Relations over events	total order over writes to the same memory location
ppo	preserved program order	architecture	any, any	§Architectures	program order maintained by the architecture
ffence, ff	full fence	architecture	any, any	§Architectures	e.g., syncon Power, dmb and dsb on ARM
lwfence, lwf	lightweight fence	architecture	any, any	§Architectures	e.g., lwsyncon Power
cfence	control fence	architecture	any, any	§Architectures	e.g., isync on Power, isb on ARM
fences	fences	architecture	any, any	§Architectures	architecture-dependent subset of the fence relations (ffence, lwfence, cfence)
prop	propagation	architecture	WW	§Architectures	order in which writes propagate, typically enforced by fences
po-loc	program order restricted to the same memory location	derived	any, any	§SC PER LOCATION	$\{(x, y) \mid (x, y) \in \text{po} \wedge \text{addr}(x) = \text{addr}(y)\}$
com	communications	derived	any, any	§Relations over events	co ∪ rf ∪ fr
fr	from-read	derived	RW	§Relations over events	links a read r to a write w' co-after the write w from which r takes its value
hb	happens before	derived	any, any	§NO THIN AIR	ppo ∪ fences ∪ rfe
rdw	read different writes	derived	RR	Fig. 27	two threads: first thread holds a write, second thread holds two reads
detour	detour	derived	WR	Fig. 28	two threads: first thread holds a write, second threads hold a write and a read

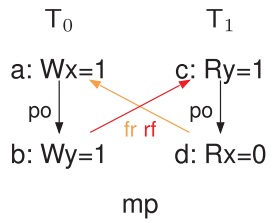


Fig. 4. Message passing pattern.

Classical patterns, such as the preceding message passing pattern, have an abbreviated name: mp stands for "message passing," sb for "store buffering," lb for "load buffering," wrc for "write-to-read causality," and rwc for "read-to-write causality."

When a pattern does not have a classical name from the literature, we give it a name that simply describes which accesses occur: for example, 2+2w means that the test is made of two threads holding two writes each, and w+rw+2w means that we have three

Table III. Glossary of Litmus Test Names

classic	systematic	diagram	description
coXY		Fig. 6	coherence test involving an access of kind X and an access of kind Y; X and Y can be either R (read) or W (write)
lb	rw+rw	Fig. 7	load buffering (i.e., two threads each holding a read then a write)
mp	ww+rr	Fig. 8	message passing (i.e., two threads: first thread holds two writes, second thread holds two reads)
wrc	w+rw+rr	Fig. 11	write to read causality (i.e., three threads: first thread holds a write, second thread holds a read then a write, third thread holds two reads)
isa2	ww+rw+rr	Fig. 12	one of the tests appearing in the Power ISA documentation [IBM Corp. 2009] (i.e., write to read causality prefixed by a write, meaning that the first thread holds two writes instead of just one, as in the wrc case)
2+2w	ww+ww	Fig. 13(a)	two threads holding two writes each
	w+rw+2w	Fig. 13(b)	three threads: first thread holds a write, second thread holds a read then a write, third thread holds two writes
sb	wr+wr	Fig. 14	store buffering (i.e., two threads each holding a write then a read)
rwc	w+rr+wr	Fig. 15	read to write causality three threads: first thread holds a write, second thread holds two reads, third thread holds a write then a read
r	ww+wr	Fig. 16	two threads: first thread holds two writes, second thread holds a write and a read
s	ww+rw	Fig. 16	two threads: first thread holds two writes, second thread holds a read and a write
w+rwc	ww+rr+wr	Fig. 19	read to write causality pattern rwc, prefixed by a write (i.e., the first thread holds two writes instead of just one, as in the rwc case)
iriw	w+rr+w+rr	Fig. 20	independent reads of independent writes (i.e., four threads: first thread holds a write, second holds two reads, third holds a write, fourth holds two reads)

threads: a write on a first thread, a read followed by a write on a second thread, and then two writes on another thread.

Finally, when we extend a test (e.g., rwc, "read-to-write causality") with an access (e.g., a write) on an extra thread, we extend the name appropriately: w+rwc (see Figure 19). We give a glossary of the test names presented in this article in Table III, in the order in which they appear; for each test, we give its systematic name, as well as its classic name (i.e., borrowed from previous works) when there is one.

Note that in every test, we choose the locations so that we can form a cycle in the relations of our model: for example, 2+2w has two threads with two writes each, such that the first one accesses, for example, the locations x and y and the second one accesses y and x. This precludes having the first thread accessing x and y and the second one z and y, because we could not link the locations to form a cycle.

Given a pattern such as mp, shown earlier,, we write mp+lwfence+ppo for the same underlying pattern where in addition the first thread has a lightweight fence lwfence between the two writes and the second thread maintains its two accesses in order thanks to some preserved program order mechanism (ppo, see later discussion). We write mp+lwfences for the mp pattern with two lightweight fences, one on each thread. We sometimes specialise the naming to certain architectures and mechanisms, as in mp+lwsync+addr, where lwsync refers to Power's lightweight fence and addr denotes an address dependency—a particular way of preserving program order on Power.

Architectures. Architectures are instances of our model. An *architecture* is a triple of functions (ppo, fences, prop) that specifies the preserved program order ppo, the fences fences and the propagation order prop.

The preserved program order gathers the set of pairs of events that are guaranteed not to be reordered w.r.t. the order in which the corresponding instructions occur in the program text. For example, on TSO, only write-read pairs can be reordered, so the preserved program order for TSO is po \ WR. On weaker models, such as Power or ARM, the preserved program order merely includes dependencies, for example, address dependencies, when the address of a memory access is determined by the value read by a preceding load. We detail these notions, and the preserved program order for Power and ARM, in Section 6.

The function ppo, given an execution $(\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$, returns the preserved program order. For example, consider the execution of the message passing example given later in Figure 8. Assume that there is an address dependency between the two reads on T_1. As such a dependency constitutes a preserved program order relation on Power, the ppo function would return the pair (c, d) for this particular execution.

Fences (or *memory barriers*) are special instructions that prevent certain behaviours. On Power and ARM (see Section 6), we distinguish between control fence (which we write cfence), lightweight fence (lwfence), and full fence (ffence). On x86, implementing TSO, there is only one fence, called mfence.

In this article, we use the same names for the fence instructions and the relations that they induce over events. For example, consider the execution of the message passing example given later in Figure 1. Assume that there is a lightweight Power fence lwsync between the two writes a and b on T_0. In this case, we would have $(a, b) \in \mathsf{lwsync}$.[2]

The function fences returns the pairs of events in program order that are separated by a fence, when given an execution. For example, consider the execution of the message passing example given in Figure 8. Assume that there is a lightweight Power fence lwsync between the two writes on T_0. On Power, the fences function would thus return the pair (a, b) for this particular execution.

The propagation order constrains the order in which writes are propagated to the memory system. This order is a partial order between writes (not necessarily to the same location), which can be enforced by using fences. For example, on Power, two writes in program order separated by an lwsync barrier (see Figure 8) will be ordered the same way in the propagation order.

We note that the propagation order is distinct from the coherence order co: indeed, co only orders writes to the same location, whereas the propagation order can relate writes with different locations through the use of fences. However, both orders have to be compatible, as expressed by our PROPAGATION axiom, which we explain next (see Figures 5 and 13(a)).

The function prop returns the pairs of writes ordered by the propagation order, given an execution. For example, consider the execution of the message passing example given later in Figure 8. Assume that there is a lightweight Power fence lwsync between the two writes on T_0. On Power, the presence of this fence forces the two writes to propagate in the order in which they are written on T_0. The function prop would thus return the pair (a, b) for this particular execution.

[2]Note that if there is a fence "fence" between two events e_1 and e_2 in program order, the pair of events (e_1, e_2) belongs to the eponymous relation "fence" (i.e., $(e_1, e_2) \in$ fence), regardless of whether the particular fence "fence" actually orders these two accesses. For example, on Power, the lightweight fence lwsync does not order write-read pairs in program order. Now consider the execution of the store buffering pattern in Figure 14, and assume that there is an lwsync between the write a and the read b on T_0. In this case, we have $(a.b) \in$ lwsync. However, the pair (a, b) would not be maintained in that order by the barrier, which we model by excluding write-read pairs separated by an lwsync from the propagation order on Power (see Figures 17 and 18: the propagation order prop contains lwfence, which on Power is defined as lwsync \ WR only, not lwsync).

$$\text{Input data: } (\mathsf{ppo}, \mathsf{fences}, \mathsf{prop}) \text{ and } (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$$

$$\text{(SC PER LOCATION) } \text{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{com}) \text{ with}$$
$$\mathsf{po\text{-}loc} \triangleq \{(x, y) \in \mathsf{po} \wedge \text{addr}(x) = \text{addr}(y)\}$$
$$\mathsf{fr} \triangleq \{(r, w_1) \mid \exists w_0.(w_0, r) \in \mathsf{rf} \wedge (w_0, w_1) \in \mathsf{co}\}$$
$$\mathsf{com} \triangleq \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$$
$$\text{(NO THIN AIR) } \text{acyclic}(\mathsf{hb}) \text{ with}$$
$$\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}$$
$$\text{(OBSERVATION) } \text{irreflexive}(\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*)$$
$$\text{(PROPAGATION) } \text{acyclic}(\mathsf{co} \cup \mathsf{prop})$$

Fig. 5. A model of weak memory.

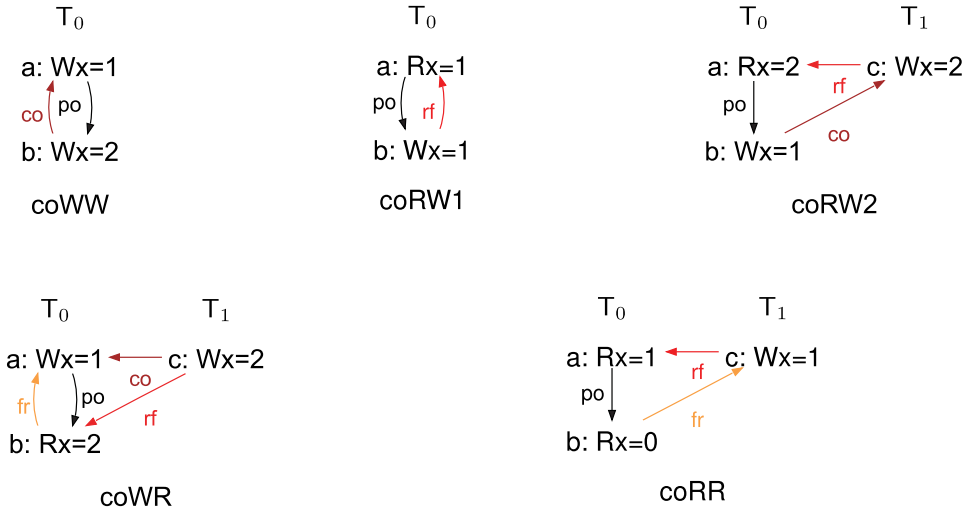


Fig. 6. The five patterns forbidden by SC PER LOCATION.

We can now explain the axioms of our model (see Figure 5). For each example execution that we present in this section, we write in the caption of the corresponding figure whether it is allowed or forbidden by our model.

4.2. SC PER LOCATION

SC PER LOCATION ensures that the communications com cannot contradict po-loc (program order between events relative to the same memory location)—that is, acyclic(po-loc ∪ com). This requirement forbids exactly the five patterns (as shown in Alglave [2010, A.3, p. 184]) given in Figure 6.

The pattern coWW forces two writes to the same memory location x in program order to be in the same order in the coherence order co. The pattern coRW1 forbids a read from x to read from a po-subsequent write. The pattern coRW2 forbids the read a to read from a write c that is co-after a write b, if b is po-after a. The pattern coWR forbids a read b to read from a write c which is co-before a previous write a in program order. The pattern coRR imposes that if a read a reads from a write c, all subsequent reads in program order from the same location (e.g., the read b) read from c or a co-successor write.

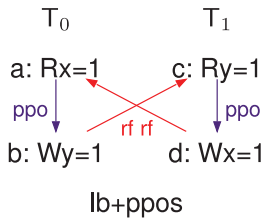


Fig. 7. The load buffering pattern lb with ppo on both sides (forbidden).

4.3. NO THIN AIR

NO THIN AIR defines a *happens-before* relation, written hb, defined as ppo ∪ fences—that is, an event e_1 happens before another event e_2 if they are in preserved program order, or there is a fence in program order between them, or e_2 reads from e_1.

NO THIN AIR requires the happens-before relation to be acyclic, which prevents values from appearing out of thin air. Consider the load buffering pattern (lb+ppos) in Figure 7. T_0 reads from x and writes to y, imposing (for example) an address dependency between the two accesses so that they cannot be reordered. Similarly, T_1 reads from y and (dependently) writes to x. NO THIN AIR ensures that the two threads cannot communicate in a manner that creates a happens-before cycle, with the read from x on T_0 reading from the write to x on T_1, whilst T_1 reads from T_0's write to y.

In the terminology of Sarkar et al. [2011], a read is *satisfied* when it binds its value; note that the value is not yet irrevocable. It becomes irrevocable when the read is *committed*. We say that a write is committed when it makes its value available to other threads. Section 6 provides further discussion of these steps.

Our happens-before relation orders the point in time where a read is satisfied and the point in time where a write is committed.

The preceding pattern shows that the ordering due to happens-before applies to any architecture that does not speculate values read from memory (i.e., values written by external threads as opposed to the same thread as the reading thread), nor allows speculative writes (e.g., in a branch) to send their value to other threads.

Indeed, in such a setting, a read such as the read a on T_0 can be satisfied from an external write (e.g., the write d on T_1) only after this external write has made its value available to other threads, and its value is irrevocable.

Our axiom forbids the lb+o pattern regardless of the method o chosen to maintain the order between the read and the write on each thread: address dependencies on both (lb+addrs), a lightweight fence on the first and an address dependency on the second (lb+lwfence+addr), and two full fences (lb+ffences). If, however, one or both pairs are not maintained, the pattern can happen.

4.4. OBSERVATION

OBSERVATION constrains the value of reads. If a write a to x and a write b to y are ordered by the propagation order prop, and if a read c reads from the write of y, then any read d from x that happens after c (i.e., $(c, d) \in$ hb) cannot read from a write that is co-before the write a (i.e., $(d, a) \notin$ fr).

4.4.1. Message Passing (mp). A typical instance of this pattern is the message passing pattern (mp+lwfence+ppo) given in Figure 8.

T_0 writes a message in x, then sets up a flag in y, so that when T_1 sees the flag (via its read from y), it can read the message in x. For this pattern to behave as intended, following the message passing protocol described earlier, the write to x needs to be propagated to the reading thread before the write to y.

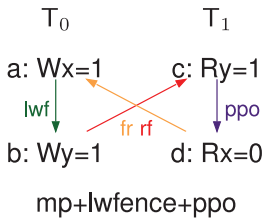


Fig. 8. The message passing pattern mp with lightweight fence and ppo (forbidden).

TSO guarantees it, but Power or ARM need at least a lightweight fence (e.g., lwsync on Power) between the writes. We also need to ensure that the two reads on T_1 stay in the order in which they have been written, such as with an address dependency.

The protocol would also be ensured with a full fence on the writing thread (mp+ffence+addr) or with two full fences (mp+ffences).

More precisely, our model assumes that a full fence is at least as powerful as a lightweight fence. Thus, the behaviours forbidden by the means of a lightweight fence are also forbidden by the means of a full fence. We insist on this point since, as we shall see, our ARM model does not feature any lightweight fence. Thus, when reading an example such as the message passing one in Figure 8, the "lwf" arrow should be interpreted as any device that has at least the same power as a lightweight fence. In the case of ARM, this means a full fence (i.e., dmb or dsb).

From a microarchitectural standpoint, fences order the propagation of writes to other threads. A very naive implementation can be by waiting until any write instructions in flight complete, and all writes are propagated to the complete systems before completing a fence instruction; modern systems feature more sophisticated protocols for ordering write propagation.

By virtue of the fence, the writes to x and y on T_0 should propagate to T_1 in the order in which they are written on T_0. Since the reads c and d on T_1 are ordered by ppo (e.g., an address dependency), they should be satisfied in the order in which they are written on T_1. In the scenario depicted in Figure 8, T_1 reads the value 1 in y (see the read event c) and the value 0 in x (see the read event d). Thus, T_1 observes the write a to x after the write b to y. This contradicts the propagation order of the writes a and b enforced by the fence on T_0.

We note that the Alpha architecture [Compaq Computer Corp. 2002] allows the pattern mp+fence+addr (a specialisation of Figure 8). Indeed, some implementations feature more than one memory port per processor, such as by the means of a split cache [Howells and McKenney 2013]. Thus, in our mp pattern shown earlier, the values written on x and y by T_0 could reach T_1 on different ports. As a result, although the address dependency forces the reads c and d to be satisfied in order, the second read may read a stale value of x, whereas the current value of x is waiting in some queue. One could counter this effect by synchronising memory ports.

4.4.2. Cumulativity. To explain a certain number of the following patterns, we need to introduce the concept of *cumulativity*.

We consider that a fence has a cumulative effect when it ensures a propagation order not only between writes on the fencing thread (i.e., the thread executing the fence) but also between certain writes coming from threads other than the fencing thread.

A-cumulativity. More precisely, consider a situation as shown on the left of Figure 9. We have a write a to x on T_0, which is read by the read b of x on T_1. On T_1 still, we have an access in program order after b. This access could be either a read or a write event; it is a write c in Figure 9. Note that b and c are separated by a fence.

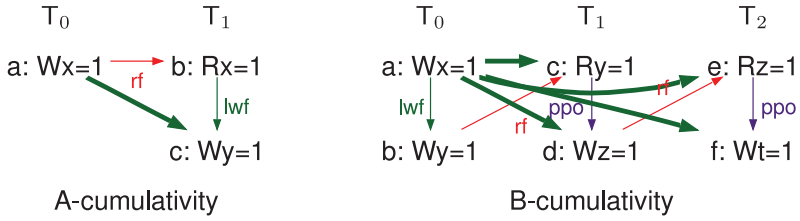


Fig. 9. Cumulativity of fences.

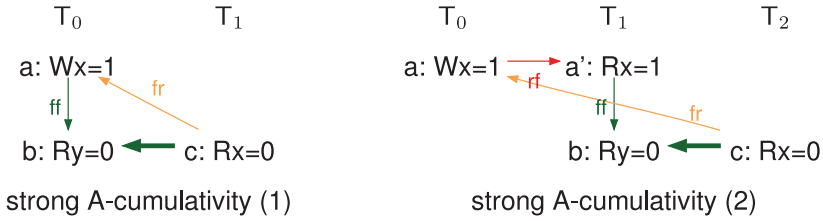


Fig. 10. Strong A-cumulativity of fences.

We say that the fence between b and c on T_1 is *A-cumulative* when it imposes that the read b is satisfied before c is either committed (if it is a write) or satisfied (if it is a read). Note that for b to be satisfied, the write a on T_0 from which b reads must have been propagated to T_1, which enforces an ordering between a and c. We display this ordering in Figure 9 by a thicker arrow from a to c.

The vendors' documentations describe A-cumulativity as follows. On Power, quoting IBM Corp. [2009, Book II, Sec. 1.7.1]: "[the group] A [of T_1] includes all [...] accesses by any [...] processor [...] [e.g., T_0] that have been performed with respect to [T_1] before the memory barrier is created." We interpret "performed with respect to [T_1]" as a write (e.g., the write a on the left of Figure 9) having been propagated to T_1 and read by T_1 such that the read is po-before the barrier.

On ARM, quoting Grisenthwaite [2009, Sec. 4]: "[the] group A [of T_1] contains: All explicit memory accesses [...] from observers [...] [e.g., T_0] that are observed by [T_1] before the dmb instruction." We interpret "observed by [T_1]" on ARM as we interpret "performed with respect to [T_1]" on Power (see preceding paragraph).

Strong A-cumulativity. Interestingly, the ARM documentation does not stop there and includes in group A "[a]ll loads [...] from observers [...] [(e.g., T_1)] that have been observed by any given observer [e.g., T_0], [...] before [T_0] has performed a memory access that is a member of group A."

Consider the situation on the left of Figure 10. We have the read c on T_1, which reads from the initial state for x, and thus is fr-before the write a on T_0. The write a is po-before a read b, such that a and b are separated by a full fence. In that case, we count the read c as part of the group A of T_0. This enforces a (strong) A-cumulativity ordering from c to b, which we depict with a thicker arrow.

Similarly, consider the situation on the right of Figure 10. The only difference with the left of the figure is that we have one more indirection between the read c of x and the read b of y, via the rf between a and a'. In that case as well, we count the read c as part of the group A of T_1. This enforces a (strong) A-cumulativity ordering from c to b, which we depict with a thicker arrow.

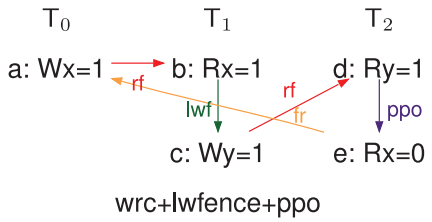


Fig. 11. The write-to-read causality pattern wrc with lightweight fence and ppo (forbidden).

We take *strong* A-cumulativity into account only for full fences (i.e., sync on Power and dmb and dsb on ARM). We reflect this later in Figure 18, in the second disjunct of the definition of the prop relation (com*; prop-base*; ffence; hb*).

B-cumulativity. Consider now the situation shown on the right of Figure 9. On T_0, we have an access a (which could be either a read or a write event) in program order before a write b to y. Note that a and b are separated by a fence. On T_1, we have a read c from y, which reads the value written by b.

We say that the fence between a and b on T_0 is *B-cumulative* when it imposes that a is either committed (if it is a write) or satisfied (if it is a read) before b is committed. Note that the read c on T_1 can be satisfied only after the write b is committed and propagated to T_1, which enforces an ordering from a to c. We display this ordering in Figure 9 by means of a thicker arrow from a to c.

This B-cumulativity ordering extends to all events that happen after c (i.e., are hb-after c), such as the write d on T_1, the read a on T_2, and the write f on T_2. In Figure 9, we display all of these orderings via thicker arrows.

The vendors' documentations describe B-cumulativity as follows. On Power, quoting IBM Corp. [2009, Book II, Sec. 1.7.1]: "[the group] B [of T_0] includes all [...] accesses by any [...] processor [...] that are performed after a load instruction [such as the read c on the right of Figure 9] executed by [T_0] [...] has returned the value stored by a store that is in B [such as the write b on the right of Figure 9]." On the right of Figure 9, this includes, for example, the write d. Then, the write d is itself in the group B, and observed by the read e, which makes the write f part of the group B as well.

On ARM, quoting Grisenthwaite [2009, Sec. 4]: "[the] group B [of T_0] contains [a]ll [...] accesses [...] by [T_0] that occur in program order after the dmb instruction. [...]" On the right of Figure 9, this mean the write b.

Furthermore, ARM's group B contains "[a]ll [...] accesses [...] by any given observer [e.g., T_1] [...] that can only occur after [T_1] has observed a store that is a member of group B." We interpret this bit as we did for Power's group B.

4.4.3. Write to Read Causality (wrc). This pattern (wrc+lwfence+ppo, given in Figure 11) illustrates the A-cumulativity of the lightweight fence on T_1, namely the rfe; fences fragment of the definition illustrated earlier in Figure 9.

There are now two writing threads, T_0 writing to x and T_1 writing to y, after reading the write of T_0. TSO still enforces this pattern without any help. But on Power and ARM, we need to place (at least) a lightweight fence on T_1 between the read of T_0 (the read b from x) and the write c to y. The barrier will force the write of x to propagate before the write of y to the T_2 even if the writes are not on the same thread.

Current multiprocessors may implement A-cumulativity in many different ways. One possible implementation of A-cumulative fences, discussed in Gharachorloo et al. [1990, Sec. 6], is to have the fences not only wait for the previous read (a in Figure 11) to be satisfied but also for all stale values for x to be eradicated from the system (e.g. the value 0 read by e on T_2).

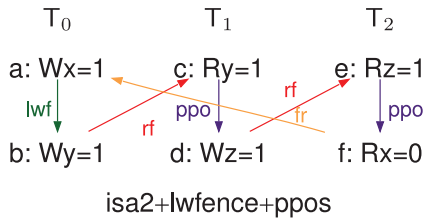
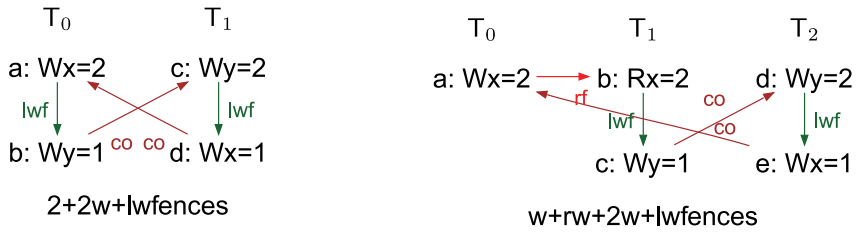


Fig. 12. The pattern isa2 with lighweight fence and ppo twice (forbidden).



(a) The pattern 2+2w

(b) The pattern w+rw+2w

Fig. 13. Two similar patterns with lightweight fences (forbidden).

4.4.4. Power ISA2 (isa2). This pattern (isa2+lwfence+ppos, given in Figure 12) distributes the message passing pattern over three threads like wrc+lwfence+ppo but keeps the writes to x and y on the same thread.

Once again, TSO guarantees it without any help, but Power and ARM need a lightweight fence between the writes, and (for example) an address or data dependency between each pair of reads (i.e., on both T_1 and T_2).

Thus, on Power and ARM, the pattern isa2+lwfence+ppos illustrates the B-cumulativity of the lightweight fence on T_0, namely the fences; hb* fragment of the definition of cumul illustrated in Figure 9.

4.5. PROPAGATION

PROPAGATION constrains the order in which writes to memory are propagated to the other threads so that it does not contradict the coherence order (i.e., acyclic(co $\cup$ prop)).

On Power and ARM, lightweight fences sometimes constrain the propagation of writes, as we have seen in the cases of mp (see Figure 8), wrc (see Figure 11), or isa2 (see Figure 12). They can also, in combination with the coherence order, create new ordering constraints.

The 2+2w+lwsync pattern (given in Figure 13(a)) is for us the archetypal illustration of coherence order and fences interacting to yield new ordering constraints. It came as a surprise when we proposed it to our IBM contact, as he wanted the lightweight fence to be as lightweight as possible (i.e., he wanted 2+2w+lwsync to be allowed) for the sake of efficiency.

However, the pattern is acknowledged to be forbidden. By contrast, and as we shall see later, other patterns (such as the r pattern in Figure 16) that mix the co communication with fr require full fences to be forbidden.

The w+rw+2w+lwfences pattern in Figure 13(b) distributes 2+2w+lwfences over three threads. This pattern is to 2+2w+lwfences what wrc is to mp. Thus, just as in the case of mp and wrc, the lightweight fence plays an A-cumulative role, which ensures that the two patterns 2+2w and w+rw+2w respond to the fence in the same way.

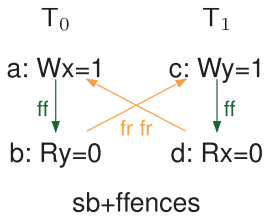


Fig. 14. The store buffering pattern sb with full fences (forbidden).

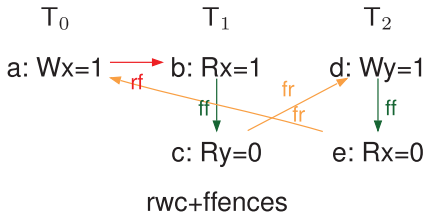


Fig. 15. The read-to-write causality pattern rwc with full fences (forbidden).

On TSO, every relation contributes to the propagation order (see Figure 21), except the write-read pairs in program order, which need to be fenced with a full fence (mfence on TSO).

Consider the store buffering (sb+ffences) pattern given in Figure 14. We need a full fence on each side to prevent the reads b and d from reading the initial state. The pattern sb without fences being allowed, even on TSO, is perhaps one of the most well-known examples of a relaxed behaviour. It can be explained by writes being first placed into a thread-local store buffer and then carried over asynchronously to the memory system. In that context, the effect of a (full) fence can be described as flushing the store buffer. Of course, on architectures more relaxed than TSO, the full fence has more work to do, such as cumulativity duties (as illustrated by the iriw example given later in Figure 20).

On Power, the lightweight fence lwsync does not order write-read pairs in program order. Hence, in particular, it is not enough to prevent the sb pattern; one needs to use a full fence on each side. The sb idiom, and the following rwc idiom, are instances of the strong A-cumulativity of full fences.

The read-to-write causality pattern rwc+ffences (Figure 15) distributes the sb pattern over three threads with a read b from x between the write a of x and the read c of y. It is to sb what wrc is to mp and thus responds to fences in the same way as sb. Hence, it needs two full fences as well. Indeed, on Power, a full fence is required to order the write a and the read c, as lightweight fences do not apply from writes to reads. The full fence on T_1 provides such an order, not only on T_1 but also by (strong) A-cumulativity from the write a on T_1 to the read c on T_1, as the read b on T_1 that reads from the write a po-precedes the fence.

The last two patterns, r+ffences and s+lwfence+ppo (Figure 16), illustrate the complexity of combining coherence order and fences. In both patterns, the thread T_0 writes to x (event a) and then to y (event b). In the first pattern, r+ffences, T_1 writes to y and reads from x. A full fence on each thread forces the write a to x to propagate to T_1 before the write b to y. Thus, if the write b is co-before the write c on T_1, the read d of x on T_1 cannot read from a write that is co-before the write a. By contrast, in the second pattern, s+lwfence+ppo, T_1 reads from y, reading the value stored by the write b and then writes to x. A lightweight fence on T_0 forces the write a to x to propagate

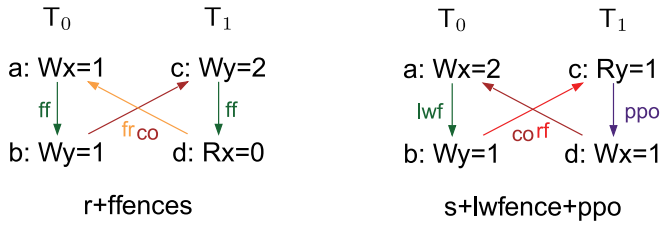


Fig. 16. The patterns r with full fences and s with lightweight fence and ppo (both forbidden).

Power	ffence $\triangleq$ sync	lwfence $\triangleq$ lwsync $\setminus$ WR	cfence $\triangleq$ isync
ARM	ffence $\triangleq$ dmb $\cup$ dsb	lwfence $\triangleq$ $\emptyset$	cfence $\triangleq$ isb

Fig. 17. Fences for Power and ARM.

$$\text{hb} \triangleq \text{ppo} \cup \text{fences} \cup \text{rfe} \qquad \text{fences} \triangleq \text{lwfence} \cup \text{ffence} \qquad \text{A-cumul} \triangleq \text{rfe}; \text{fences}$$
$$\text{prop-base} \triangleq (\text{fences} \cup \text{A-cumul}); \text{hb}^*$$
$$\text{prop} \triangleq (\text{prop-base} \cap \text{WW}) \cup (\text{com}^*; \text{prop-base}^*; \text{ffence}; \text{hb}^*)$$

Fig. 18. Propagation order for Power and ARM.

to T_1 before the write b to y. Thus, as T_1 sees the write b by reading its value (read c) and as the write d is forced to occur by a dependency (ppo) after the read c, that write d cannot co-precede the write a.

Following the architect's intent, inserting a lightweight fence lwsync between the writes a and b does not suffice to forbid the r pattern on Power. It comes in sharp contrast with, for instance, the mp pattern (see Figure 8) and the s pattern, where a lightweight fence suffices. Thus, the interaction between (lightweight) fence order and coherence order is quite subtle, as it does forbid 2+2w and s, but not r.

For completeness, we note that we did not observe the r+lwsync+sync pattern on Power hardware. Therefore, the architect's intent came as a surprise to us, as our first intuition (and indeed our first model [Alglave et al. 2010, 2012]) was to have only a lightweight fence on the first thread and a full fence on the second thread.

4.6. Fences and Propagation on Power and ARM

We summarise the fences and propagation order for Power and ARM in Figures 17 and 18. Note that the control fences (isync for Power and isb for ARM) do not contribute to the propagation order. They contribute to the definition of preserved program order, as explained in Sections 5 and 6.

Fence Placement. Fence placement is the problem of automatically placing fences between events to prevent undesired behaviours. For example, the message passing pattern mp in Figure 8 can be prevented by placing fences between events a and b in T_0 and events c and d in T_1 in order to create a forbidden cycle.

This problem has been studied before for TSO and its siblings PSO and RMO (e.g., see Kuperstein et al. [2010], Bouajjani et al. [2011, 2013], and Liu et al. [2012]), or for previous versions of the Power model [Alglave and Maranget 2011].

We emphasise how easy fence placement becomes, in the light of our new model. We can read them off the definitions of the axioms and of the propagation order. Placing fences essentially amounts to counting the number of communications (i.e., the relations in com) involved in the behaviour that we want to forbid.

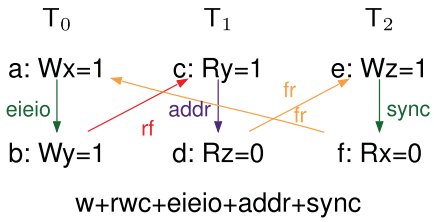


Fig. 19. The pattern w+rwc with eieio, address dependency, and full fence (allowed).

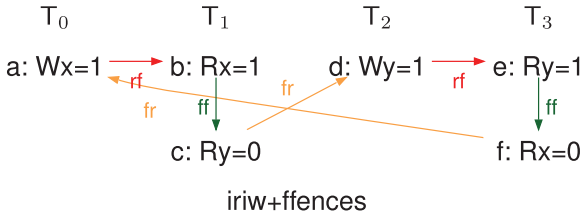


Fig. 20. The independent reads from independent writes pattern iriw with full fences (forbidden).

To forbid a behaviour that involves only read-from (rf) communications, or only *one* from-read (fr) and otherwise rf communications, one can resort to the OBSERVATION axiom, using the prop-base part of the propagation order. Namely, to forbid a behaviour of the mp (see Figure 8), wrc (see Figure 11), or isa2 (see Figure 12) type, one needs a lightweight fence on the first thread, and some preserved program order mechanisms on the remaining threads.

If coherence (co) and read-from (rf) are the only communications involved, one can resort to the PROPAGATION axiom, using the prop-base part of the propagation order. Namely, to forbid a behaviour of the 2+2w (see Figure 13(a)) or w+rw+2w (see Figure 13(b)) type, one needs only lightweight fences on each thread.

If more than one from-read (fr) occurs, or if the behaviour involves both coherence (co) and from-read (fr) communications, one needs to resort to the part of the propagation order that involves the full fence (indeed, that is the only part of the propagation order that involves com). Namely, to forbid behaviours of the type sb (see Figure 14), rwc (see Figure 15), or r (see Figure 16), one needs to use full fences on each thread.

Power's eieio and ARM's dmb.st and dsb.st. We make two additional remarks, not reflected in the figures, about write-write barriers, namely the eieio barrier on Power, and the dmb.st and dsb.st barriers on ARM.

On Power, the eieio barrier only maintains write-write pairs, as far as ordinary memory (*Memory Coherence Required*) memory is concerned [IBM Corp. 2009, p. 702]. We demonstrate (see `http://diy.inria.fr/cats/power-eieio/`) that eieio cannot be a full barrier, as this option is invalidated by hardware. For example, the following w+rwc+eieio+addr+sync pattern (Figure 19) would be forbidden if eieio were a full barrier, yet is observed on Power 6 and Power 7 machines. Indeed, this pattern involves two from-reads (fr), in which case our axioms require us to resort to the full fence part of the propagation order. Thus, we take eieio to be a lightweight barrier that maintains only write-write pairs.

The ARM architecture features the dmb and dsb fences. ARM documentation [Grisenthwaite 2009] forbids iriw+dmb (Figure 20). Hence, dmb is a full fence, as this behaviour involves two from-reads (fr), and thus needs full fences to be forbidden.

The ARM documentation also specifies dsb to behave at least as strongly as dmb [ARM Ltd. 2010, p. A3–49]: "A dsb behaves as a dmb with the same arguments, and

SC: ppo $\triangleq$ po ffence $\triangleq \emptyset$ lwfence $\triangleq \emptyset$ fences $\triangleq$ ffence $\cup$ lwfence
prop $\triangleq$ ppo $\cup$ fences $\cup$ rf $\cup$ fr

TSO: ppo $\triangleq$ po $\setminus$ WR ffence $\triangleq$ mfence lwfence $\triangleq \emptyset$
fences $\triangleq$ ffence $\cup$ lwfence prop $\triangleq$ ppo $\cup$ fences $\cup$ rfe $\cup$ fr

C++ R-A $\approx$: ppo $\triangleq$ sb (see [Batty et al. 2011]) fences $\triangleq \emptyset$ prop $\triangleq$ hb
(we write $\approx$ to indicate that our definition has a discrepancy with the standard)

Fig. 21. Definitions of SC, TSO, and C++ R-A.

also has [. . .] additional properties [. . .].” Thus, we take both dmb and dsb to be full fences. We remark that this observation and our experiments (see Section 8) concern memory litmus tests only; we do not know whether dmb and dsb differ (or indeed should differ) when out-of-memory communication (e.g., interrupts) comes into play.

Finally, the ARM architecture specifies that, when suffixed by .st, ARM fences operate on write-write pairs only. It remains to be decided whether the resulting dmb.st and dsb.st fences are lightweight fences or not. In that aspect, ARM documentation does not help much, nor do experiments on hardware, as all of our experiments are compatible with both alternatives (see `http://diy.inria.fr/cats/arm-st-fences`). Thus, we choose simplicity and consider that .st fences behave as their unsuffixed counterparts but are limited to write-write pairs. In other words, we assume that the ARM architecture makes no provision for some kind of lightweight fence—unlike Power, which provides lwsync.

Formally, to account for .st fences being full fences limited to write-write pairs, we would proceed as follows. In Figure 17 for ARM, we would extend the definition of ffence to dmb $\cup$ dsb $\cup$ (dmb.st $\cap$ WW) $\cup$ (dsb.st $\cap$ WW). Should the option of .st fences being lightweight fences be preferred (i.e., thereby allowing the pattern w+rwc+dmb.st+addr+dmb of Figure 19), one would instead define lwfence as (dmb.st $\cap$ WW) $\cup$ (dsb.st $\cap$ WW) in Figure 17 for ARM.

4.7. Some Instances of Our Framework

We now explain how we instantiate our framework to produce the following models: Lamport’s SC [Lamport 1979]; TSO, used in the Sparc [SPARC International Inc. 1994] and x86 [Owens et al. 2009] architectures; and C++ R-A—that is, C++ restricted to the use of release-acquire atomics [Batty et al. 2011, 2013].

SC and TSO. SC and TSO are described in our terms at the top of Figure 21, which gives their preserved program order, fences, and propagation order. We show that these instances of our model correspond to SC and TSO:

Lemma 4.1. *Our model of SC as given in Figure 21 is equivalent to Lamport’s SC [Lamport 1979]. Our model of TSO as given in Figure 21 is equivalent to Sparc TSO [SPARC International Inc. 1994].*

Proof. An execution ($\mathbb{E}$, po, co, rf) is valid on SC (resp. TSO) iff acyclic(po$\cup$com) (resp. acyclic(ppo $\cup$ co $\cup$ rfe $\cup$ fr $\cup$ fences)) (all relations defined as in Figure 21) by Alglave [2012, Def. 21, p. 203] (resp. Alglave [2012, Def. 23, p. 204].) $\square$

C++ R-A. C++ R-A—that is, C++ restricted to the use of release-acquire atomics—appears at the bottom of Figure 21 in a slightly stronger form than the current standard prescribes, as detailed next.

We take the *sequenced before* relation sb of Batty et al. [2011] to be the preserved program order and the fences to be empty. The happens-before relation hb $\triangleq$ sb $\cup$ rf is

the only relation contributing to propagation (i.e., $\mathsf{prop} = \mathsf{hb}^+$). We take the modification order mo of Batty et al. [2011] to be our coherence order.

The work of Batty et al. [2013] shows that C++ R-A is defined by three axioms: ACYCLICITY (i.e., acyclic(hb)) (which immediately corresponds to our NO THIN AIR axiom), COWR (i.e., $\forall r. \neg(\exists w_1, w_2.\ (w_1, w_2) \in \mathsf{mo} \wedge (w_1, r) \in \mathsf{rf} \wedge (w_2, r) \in \mathsf{hb}^+)$) (which corresponds to our OBSERVATION axiom by definition of fr and since $\mathsf{prop} = \mathsf{hb}^+$ here), and HBVSMO (i.e., irreflexive($\mathsf{hb}^+; \mathsf{mo}$)). Our SC PER LOCATION is implemented by HBVSMO for the coWW case, and the eponymous axioms for the coWR, coRW, coRR cases.

Thus, C++ R-A corresponds to our version, except for the HBVSMO axiom, which requires the irreflexivity of $\mathsf{hb}^+; \mathsf{mo}$, whereas we require its acyclicity via the axiom PROPAGATION. To adapt our framework to C++ R-A, one simply needs to weaken the PROPAGATION axiom to irreflexive(prop; co).

4.8. A Note on the Genericity of Our Framework

We remark that our framework is, as of today, not as generic as it could be, for several reasons that we explain next.

Types of Events. For a given read or write event, we handle only one type of this event. For example, we can express C++ when all reads are acquire and all writes are release, but nothing more. As of today, we could not express a model where some writes can be relaxed writes (in the C++ sense) and others can be release writes. To embrace models with several types of accesses, such as C++ in its entirety, we would need to extend the expressiveness of our framework. We hope to investigate this in future work.

Choice of Axioms. We note that our axiom SC PER LOCATION might be perceived as too strong, as it forbids *load-load hazard* behaviours (see coRR in Figure 6). This pattern was indeed officially allowed by Sparc RMO [SPARC International Inc. 1994] and pre-Power 4 machines [Tendler et al. 2002].

Similarly, the NO THIN AIR axiom might be perceived as controversial, as several software models, such as Java or C++, allow certain lb patterns (see Figure 7).

We also have already discussed, in the preceding section, how the PROPAGATION axiom needs to be weakened to reflect C++ R-A accurately.

Yet we feel that this is much less of an issue than having only one type of events. Indeed, one can very simply disable the NO THIN AIR check, or restrict the SC PER LOCATION check so that it allows load-load hazard (e.g., see Alglave [2012, Sec. 5.1.2]), or weaken the PROPAGATION axiom (as done earlier).

Rather than axioms set in stone and declared as the absolute truth, we present here some basic bricks from which one can build a model at will. Moreover, our new herd simulator (see Section 8.3) allows the user to very simply and quickly modify the axioms of their model, and re-run their tests immediately, without having to dive into the code of the simulator. This reduced the cost of experimenting with several different variants of a model or fine-tuning a model. We give a flavour of such experimentations and fine-tuning in our discussion about the ARM model (e.g., see Table V, later).

5. INSTRUCTION SEMANTICS

In this section, we specialise the discussion to Power, to make the reading easier. Before presenting the preserved program order for Power, given later in Figure 25, we need to define *dependencies*. We borrow the names and notations of Sarkar et al. [2011] for more consistency.

To define dependencies formally, we need to introduce some more possible actions for our events. In addition to the read and write events relative to memory locations, we can now have:

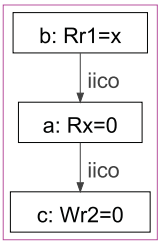
—read and write events relative to registers, (e.g., for a register r, we can have a read of the value 0, noted Rr=0, or a write of the value 1, noted Wr=1);
—branching events, which represent the branching decisions being made; and
—fence events (e.g., lwfence).

Note that in general, a single instruction can involve several accesses, for example, to registers. Events coming from the same instruction can be related by the relation iico (intra-instruction causality order). We give examples of such a situation below.

5.1. Semantics of Instructions

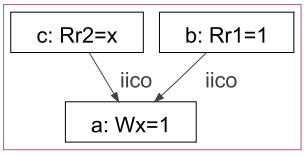
We now give examples of the semantics of instructions. We do not intend to be exhaustive, but rather to give the reader enough understanding of the memory, register, branching, and fence events that an instruction can generate so that we can define dependencies w.r.t. these events in due course. We use Power assembly syntax [IBM Corp. 2009].

Here is the semantics of a load "lwz r2,0(r1)" with r1 holding the address x, assuming that x contains the value 0. The instruction reads the content of the register r1 and finds there the memory address x; then (following iico), it reads from location x and finds there the value 0. Finally, it writes this value into register r2:



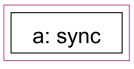
load

Here is the semantics of a store "stw r1,0(r2)" with r1 holding the value 1 and r2 holding the address x. The instruction reads the content of the register r2 and finds there the memory address x. In parallel, it reads the content of the register r1 and finds there the value 1. After (in iico) these two events, it writes the value 1 into memory address x:



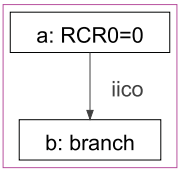
store

Here is the semantics of a "sync" fence, simply an eponymous event:



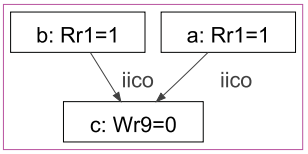
sync fence

Here is the semantics for a branch "bne L0," which branches to the label L0 if the value of a special register (the Power ISA specifies it to be register CR0 [IBM Corp. 2009, Ch. 2, p. 35]) is not equal to 0 (bne stands for "branch if not equal"). Thus, the instruction reads the content of CR0 and emits a branching event. Note that it emits the branching event regardless of the value of CR0, just to signify that a branching decision has been made:



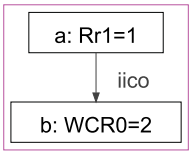
branching

Here is the semantics for a "xor r9,r1,r1," which takes the bitwise xor of the value in r1 with itself and puts the result into the register r9. The instruction reads (twice) the value in the register r1, takes their xor, and writes the result (necessarily 0) in r9:



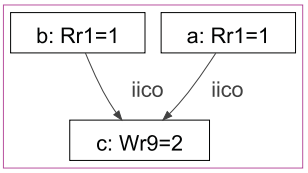
xor

Here is the semantics for a comparison "cmpwi r1, 1," which compares the value in register r1 with 1. The instruction writes the result of the comparison (2 encodes equality) into the special register CR0 (the same that is used by branches to make branching decisions). Thus, the instruction reads the content of r1, then writes to CR0 accordingly:



comparison

Here is the semantics for an addition "add r9,r1,r1," which reads the value in register r1 (twice) and writes the sum into register r9:



addition

5.2. Dependencies

We can now define dependencies formally, in terms of the events generated by the instructions involved in implementing a dependency. We borrow the textual explanations from Sarkar et al. [2011].

In Figure 22, we give the definitions of address (addr), data (data), control (ctrl), and control+cfence (ctrl+cfence) dependencies. Next, we detail and illustrate what they mean.

In Figure 22, we use the notations that we have defined previously (see Section 4 for sets of events). We write "M" for the set of all memory events so that, for example, RM is the set of pairs (r, e), where r is a read and e any memory event (i.e., a write or a read). We write "B" for the set of branch events (coming from a branch instruction); thus, RB is the set of pairs (r, b), where r is a read and b a branch event.

$\mathsf{dd\text{-}reg} = (\mathsf{rf\text{-}reg} \cup \mathsf{iico})^+$	Data dependency over registers
$\mathsf{addr} = \mathsf{dd\text{-}reg} \cap \mathsf{RM}$	The last rf−reg is to the *address entry port* of the target instruction.
$\mathsf{data} = \mathsf{dd\text{-}reg} \cap \mathsf{RW}$	The last rf−reg is to the *value entry port* of the target store instruction.
$\mathsf{ctrl} = (\mathsf{dd\text{-}reg} \cap \mathsf{RB}); \mathsf{po}$	On Power or ARM, control dependencies targetting a read do not belong to ppo.
$\mathsf{ctrl{+}cfence} = (\mathsf{dd\text{-}reg} \cap \mathsf{RB}); \mathsf{cfence}$	On Power or ARM, branches followed by a control fence (isync on Power, isb on ARM) targetting a read belong to ppo.

Fig. 22. Definitions of dependency relations.

Each definition uses the relation dd-reg, defined as $(\mathsf{rf\text{-}reg} \cup \mathsf{iico})^+$. This relation dd-reg builds chains of read-from through registers (rf-reg) and intra-instruction causality order (iico). Thus, it is a special kind of data dependency, over register accesses only.

We find that the formal definitions in Figure 22 make quite evident that all of these dependencies (addr, data, ctrl, and ctrl+cfence) correspond to data dependencies over registers (dd-reg), starting with a read. The key difference between each of these dependencies is the kind of events that they target: whilst addr targets any kind of memory event, data only targets writes. A ctrl dependency only targets branch events; a ctrl+cfence also targets only branch events but requires a control fence cfence to immediately follow the branch event.

5.2.1. Address Dependencies. Address dependencies are gathered in the addr relation. There is an address dependency from a memory read r to a po-subsequent memory event e (either read or write) if there is a dataflow path (i.e., a dd-reg relation) from r to the address of e through registers and arithmetic or logical operations (but not through memory accesses). Note that this notion also includes *false dependencies*—for example, when xor'ing a value with itself and using the result of the xor in an address calculation. For example, in Power (on the left) or ARM assembly (on the right), the following excerpt

```
(1) lwz r2,0(r1)          ldr r2,[r1]
(2) xor r9,r2,r2          eor r9,r2,r2
(3) lwzx r4,r9,r3         ldr r4,[r9,r3]
```

ensures that the load at line (3) cannot be reordered with the load at line (1), despite the result of the xor at line (2) being always 0.

Graphically (see the left diagram of Figure 23), the read a from address x is related by addr to the read b from address y because there is a path of rf and iico (through register events) between them. Notice that the last rf is to the index register (here r9) of the load from y instruction.

5.2.2. Data Dependencies. Data dependencies are gathered in the data relation. There is a data dependency from a memory read r to a po-subsequent memory write w if there is a dataflow path (i.e., a dd-reg relation) from r to the value written by w through registers and arithmetic or logical operations (but not through memory accesses). This also includes false dependencies, as described earlier. For example,

```
(1) lwz r2,0(r1)          ldr r2,[r1]
(2) xor r9,r2,r2          eor r9,r2,r2
(3) stw r9,0(r4)          str r9,[r4]
```

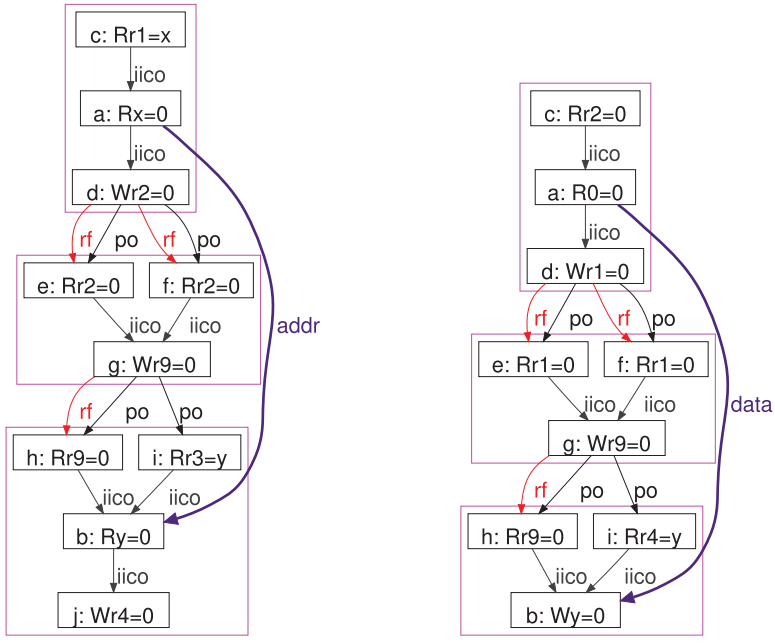


Fig. 23. Dataflow dependencies.

ensures that the store at line (3) cannot be reordered with the load at line (1), despite the result of the xor at line (2) being always 0.

Graphically (see the right diagram of Figure 23), the read a from address x is related by data to the write b to address y because there is a path of rf and iico (through registers) between them, the last rf being to the value entry port of the store.

ARM's Conditional Execution. Our semantics does not account for conditional execution in the ARM sense (see ARM Ltd. [2010, Sec. A8.3.8 "Conditional execution"] and Grisenthwaite [2009, Sec. 6.2.1.2]). Informally, most instructions can be executed or not, depending on condition flags. It is unclear how to handle them in full generality, either both as target of dependencies (conditional load and conditional store instructions) or in the middle of a dependency chain (e.g., conditional move). In the target case, a dependency reaching a conditional memory instruction through its condition flag could act as a control dependency. In the middle case, the conditional move could contribute to the dataflow path that defines address and data dependencies. We emphasise that we have not tested these hypotheses.

5.2.3. Control Dependencies. Control dependencies are gathered in the ctrl relation. There is a control dependency from a memory read r to a po-subsequent memory write w if there is a dataflow path (i.e., a dd-reg relation) from r to the test of a conditional branch that precedes w in po. For example,

(1) lwz r2,0(r1)	ldr r2,[r1]
(2) cmpwi r2,0	cmp r2,#0
(3) bne L0	bne L0
(4) stw r3,0(r4)	str r3,[r4]
(5) L0:	L0:

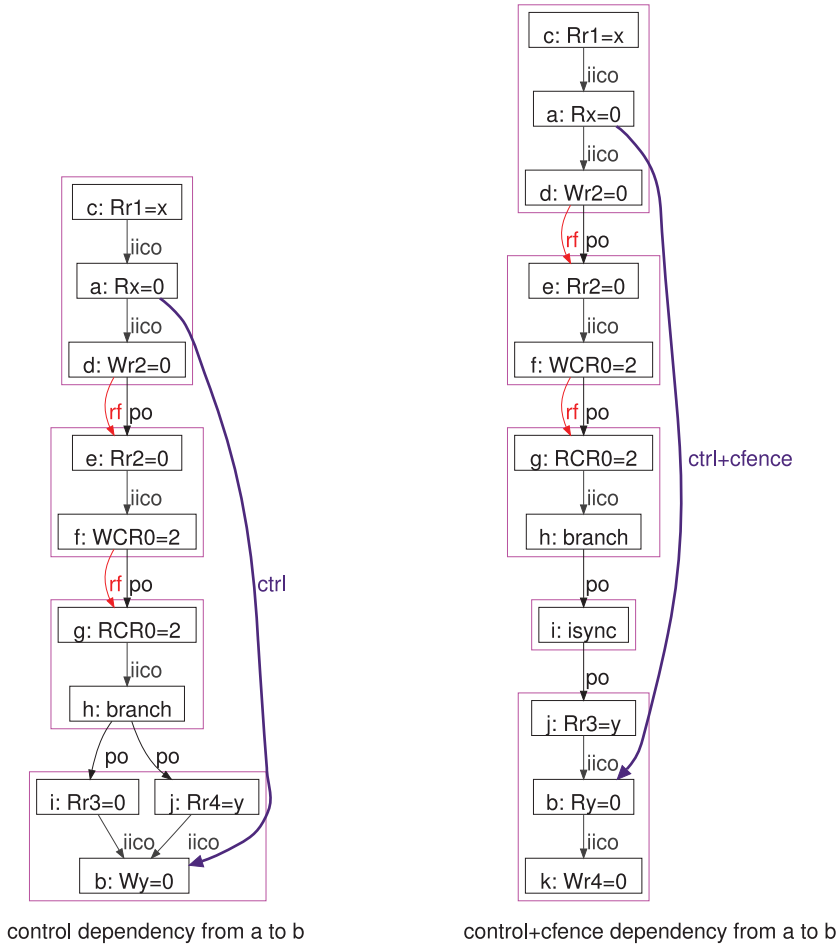


Fig. 24. Control-flow dependencies.

ensures that the store at line (4) cannot be reordered with the load at line (1). We note that there will still be a control dependency from the load to the store, even if the label immediately follows the branch—that is, the label L0 is placed between the conditional branch instruction at line (3) and the store. This property may not hold for all architectures, but it does hold for Power and ARM, as stated in their documentation and as we have checked on numerous implementations. As a side note, the analogue of the construct in a high-level language would be the sequence of an "if" statement, guarding an empty instruction, and a store. Most compilers would optimize the "if" statement away and would thus not preserve the control dependency.

Graphically (see the left diagram[3] of Figure 24), the read a from address x is related by ctrl to the write b to address y because there is a path of rf and iico (through registers) between a and a branch event (h in that case) po-before b (some po edges are omitted for clarity).

[3]The diagram depicts the situation for Power; ARM status flags are handled differently.

$$\mathsf{dp} \triangleq \mathsf{addr} \cup \mathsf{data} \qquad \mathsf{rdw} \triangleq \mathsf{po\text{-}loc} \cap (\mathsf{fre}; \mathsf{rfe}) \qquad \mathsf{detour} \triangleq \mathsf{po\text{-}loc} \cap (\mathsf{coe}; \mathsf{rfe})$$

$$\mathsf{ii}_0 \triangleq \mathsf{dp} \cup \mathsf{rdw} \cup \mathsf{rfi} \qquad \mathsf{ci}_0 \triangleq (\mathsf{ctrl{+}cfence}) \cup \mathsf{detour}$$

$$\mathsf{ic}_0 \triangleq \emptyset \qquad \mathsf{cc}_0 \triangleq \mathsf{dp} \cup \mathsf{po\text{-}loc} \cup \mathsf{ctrl} \cup (\mathsf{addr}; \mathsf{po})$$

$$\mathsf{ii} \triangleq \mathsf{ii}_0 \cup \mathsf{ci} \cup (\mathsf{ic}; \mathsf{ci}) \cup (\mathsf{ii}; \mathsf{ii}) \qquad \mathsf{ci} \triangleq \mathsf{ci}_0 \cup (\mathsf{ci}; \mathsf{ii}) \cup (\mathsf{cc}; \mathsf{ci})$$

$$\mathsf{ic} \triangleq \mathsf{ic}_0 \cup \mathsf{ii} \cup \mathsf{cc} \cup (\mathsf{ic}; \mathsf{cc}) \cup (\mathsf{ii}; \mathsf{ic}) \qquad \mathsf{cc} \triangleq \mathsf{cc}_0 \cup \mathsf{ci} \cup (\mathsf{ci}; \mathsf{ic}) \cup (\mathsf{cc}; \mathsf{cc})$$

$$\mathsf{ppo} \triangleq (\mathsf{ii} \cap \mathsf{RR}) \cup (\mathsf{ic} \cap \mathsf{RW})$$

Fig. 25. Preserved program order for Power.

Table IV. Terminology Correspondence

present paper	Sarkar et al. [2011]	Mador-Haim et al. [2012]
init read i(r)	satisfy read	satisfy read
commit read c(r)	commit read	commit read
init write i(w)	n/a	init write
commit write c(w)	commit write	commit write

Such a dataflow path between two memory reads is not enough to order them in general. To order two memory reads, one needs to place a control fencecfence (isync on Power, isb on ARM, as shown at the top of Figure 25) after the branch, as described next.

*5.2.4. Control+*cfence *Dependencies.* All of the control+cfence dependencies are gathered in the ctrl+cfence relation. There is such a dependency from a memory read r_1 to a po-subsequent memory read r_2 if there is a dataflow path (i.e., a dd-reg relation) from r_1 to the test of a conditional branch that po-precedes a control fence, the fence itself preceding r_2 in po. For example,

(1) lwz r2,0(r1)	ldr r2,[r1]
(2) cmpwi r2,0	cmp r2,#0
(3) bne L0	bne L0
(4) isync	isb
(5) lwz r4,0(r3)	ldr r4,[r3]
(6) L0:	L0:

ensures, thanks to the control fence at line (4), that the load at line (5) cannot be reordered with the load at line (1).

Graphically (see the right diagram[3] of Figure 24), the read a from address x is related by ctrl+cfence to the read b from address y because there is a path of rf and iico (through registers) between a and a branch event (h here) po-before a cfence (i : isync here) po-before b.

6. PRESERVED PROGRAM ORDER FOR POWER

We can now present how to compute the preserved program order for Power, which we give in Figure 25. Some readers might find it easier to read the equivalent specification given later in Figure 38. ARM is broadly similar; we detail it in the next section, in light of our experiments.

To define the preserved program order, we first need to distinguish two parts for each memory event. To name these parts, we again borrow the terminology of the models of Sarkar et al. [2011] and Mador-Haim et al. [2012] for more consistency. We give a table of correspondence between the names of Sarkar et al. [2011] and Mador-Haim et al. [2012] and the present paper in Table IV.

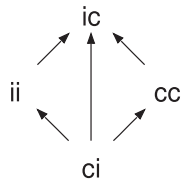


Fig. 26. Inclusions between subevents relations.

A memory read r consists of an *init* $\mathsf{i}(r)$, where it reads its value, and a *commit* part $\mathsf{c}(r)$, where it becomes irrevocable. A memory write w consists of an init part $\mathsf{i}(w)$, where its value becomes available locally, and a commit part $\mathsf{c}(w)$, where the write is ready to be sent out to other threads.

We now describe how the parts of events relate to one another. We do a case disjunction over the part of the events we are concerned with (init or commit).

Thus, we define four relations (see Figure 25): ii relates the init parts of events; ic relates the init part of a first event to the commit part of a second event; ci relates the commit part of a first event to the init part of a second event; and cc relates the commit parts of events. We define these four relations recursively, with a least fixed point semantics; we write r_0 for the base case of the recursive definition of a relation r.

Note that the two parts of an event e are ordered: its init $\mathsf{i}(e)$ precedes its commit $\mathsf{c}(e)$. Thus, for two events e_1 and e_2, if, for example, the commit of e_1 is before the commit of e_2 (i.e., $(e_1, e_2) \in \mathsf{cc}$), then the init of e_1 is before the commit of e_2 (i.e., $(e_1, e_2) \in \mathsf{ic}$). Therefore, we have the following inclusions (see also Figures 25 and 26): ii contains ci; ic contains ii and cc; and cc contains ci.

Moreover, these orderings hold transitively: for three events $e_1, e_2,$ and e_3, if the init of e_1 is before the commit of e_2 (i.e., $(e_1, e_2) \in \mathsf{ic}$), which is itself before the init of e_3 (i.e., $(e_2, e_3) \in \mathsf{ci}$), then the init of e_1 is before the init of e_3 (i.e., $(e_1, e_3) \in \mathsf{ii}$). Therefore, we have the following inclusions (see also Figure 25): ii contains (ic; ci) and (ii; ii); ic contains (ic; cc) and (ii; ic); ci contains (ci; ii) and (cc; ci); and cc contains (ci; ic) and (cc; cc).

We now describe the base case for each of our four relations—that is, ii_0, ic_0, ci_0, and cc_0. We do so by a case disjunction over whether the concerned events are reads or writes. We omit the cases where there is no ordering, such as between two init writes.

Init Reads. Init reads ($\mathsf{ii}_0 \cap \mathsf{RR}$ in Figure 25) are ordered by (i) the addr relation, which is included in the more general "dependencies" (dp) relation that gathers address (addr) and data (data) dependencies as defined previously; (ii) the rdw relation ("read different writes"), defined as po-loc $\cap$ (fre; rfe).

For case (i), if two reads are separated (in program order) by an address dependency, then their init parts are ordered. The microarchitectural intuition is immediate: we simply lift to the model level the constraints induced by dataflow paths in the core. The vendors' documentations for Power [IBM Corp. 2009, Book II, Sec. 1.7.1] and ARM [Grisenthwaite 2009, Sec. 6.2.1.2] support our intuition, as shown next.

Quoting Power's documentation [IBM Corp. 2009, Book II, Sec. 1.7.1]: "[i]f a load instruction depends on the value returned by a preceding load instruction (because the value is used to compute the effective address specified by the second load), the corresponding storage accesses are performed in program order with respect to any processor or mechanism [...]." We interpret this bit as a justification for address dependencies (addr) contributing to the ppo.

Furthermore, Power's documentation adds: "[t]his applies even if the dependency has no effect on program logic (e.g., the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load)." We interpret this as a justification for "false dependencies," as described in Section 5.2.1.

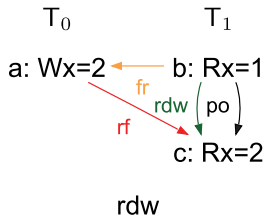


Fig. 27. The read different writes rdw relation.

Quoting ARM's documentation [Grisenthwaite 2009, Sec. 6.2.1.2]: "[if] the value returned by a read is used to compute the virtual address of a subsequent read or write (this is known as an address dependency), then these two memory accesses will be observed in program order. An address dependency exists even if the value read by the first read has no effect in changing the virtual address (as might be the case if the value returned is masked off before it is used, or if it had no effect on changing a predicted address value)." We interpret this ARM rule as we do for the Power equivalent.

We note, however, that the Alpha architecture (see our discussion of mp+fence+addr being allowed on Alpha on p. 17) demonstrates that sophisticated hardware may invalidate the lifting of core constraints to the complete system.

Case (ii) means that the init parts of two reads b and c are ordered when the instructions are in program order, reading from the same location, and the first read b reads from an external write that is co-before the write a from which the second read c reads, as shown in Figure 27.

We may also review this case in the write-propagation model [Sarkar et al. 2011; Mador-Haim et al. 2012]: as the read b reads from some write that is co-before the write a, b is satisfied before the write a is propagated to T_1; furthermore as the read c reads from the write a, it is satisfied after the write a is propagated to T_1. As a result, the read b is satisfied before the read c is.

Init Read and Init Write. Init read and init write ($ii_0 \cap RW$) relate by address and data dependencies—that is, dp. This bit is similar to the ordering between two init reads (see $ii_0 \cap RR$), but here both address and data dependencies are included. Vendors' documentations [IBM Corp. 2009, Book II, Sec. 1.7.1; ARM Ltd. 2010, Sec. A3.8.2] denote address and data dependencies from read to write.

Quoting Power's documentation [IBM Corp. 2009, Book II, Sec. 1.7.1]: "[b]ecause stores cannot be performed "out-of-order" (see Book III), if a store instruction depends on the value returned by a preceding load instruction (because the value returned by the load is used to compute either the effective address specified by the store or the value to be stored), the corresponding storage accesses are performed in program order." We interpret this bit as a justification for lifting both address and data dependencies to the ppo.

Quoting ARM's documentation [ARM Ltd. 2010, Sec. A3.8.2]: "[i]f the value returned by a read access is used as data written by a subsequent write access, then the two memory accesses are observed in program order." We interpret this ARM rule as we did for the preceding Power discussion.

Init Write and Init Read. Init write and init read ($ii_0 \cap WR$) relate by the internal read-from rfi. This ordering constraint stems directly from our interpretation of init subevents (see Table IV): init for a write is the point in time when the value stored by the write becomes available locally, whereas init for a read is the point in time when the read picks its value. Thus, a read can be satisfied from a local write only once the write in question has made its value available locally.

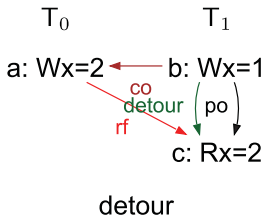


Fig. 28. The detour relation.

Commit Read and Init Read or Write. Commit read and init read or write ($\mathsf{ci}_0 \cap \mathsf{RM}$) relate by ctrl+cfence dependencies. The ctrl+cfence relation models the situation where a first read controls the execution of a branch that contains a control fence that po-precedes the second memory access. An implementation of the control fence could refetch the instructions that po-follows the fence (e.g., see the quote of ARM's documentation [ARM Ltd. 2010, Sec. A3.8.3] that we give later) and prevent any speculative execution of the control fence. As a result, instructions that follow the fence may start only once the branching decision is irrevocable—that is, once the controlling read is irrevocable.

Quoting Power's documentation [IBM Corp. 2009, Sec. 1.7.1]: "[b]ecause an isync instruction prevents the execution of instructions following the isync until instructions preceding the isync have completed, if an isync follows a conditional branch instruction that depends on the value returned by a preceding load instruction, the load on which the branch depends is performed before any loads caused by instructions following the isync." We interpret this bit as saying that a branch followed by an isync orders read-read pairs on Power (read-write pairs only need a branch, without an isync). Furthermore, the documentation adds: "[t]his applies even if the effects of the dependency are independent of the value loaded (e.g., the value is compared to itself and the branch tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction." This means that the dependency induced by the branch and the isync can be a "false dependency."

Quoting ARM's documentation [ARM Ltd. 2010, Sec. A3.8.3]: "[a]n isb instruction flushes the pipeline in the processor, so that all instructions that come after the isb instruction in program order are fetched from cache or memory only after the isb instruction has completed." We interpret this bit as saying that a branch followed by an isb orders read-read and read-write pairs on ARM.

Commit Write and Init Read. Commit write and init read ($\mathsf{ci}_0 \cap \mathsf{WR}$) relate by the relation detour, defined as $\mathsf{po\text{-}loc} \cap (\mathsf{coe}; \mathsf{rfe})$. This means that the commit of a write b to memory location x precedes the init of a read c from x when c reads from an external write a that follows b in coherence, as shown in Figure 28.

This effect is similar to the rdw effect (see Figure 27) but applied to write-read pairs instead of read-read pairs in the case of rdw. As a matter of fact, in the write-propagation model of Sarkar et al. [2011] and Mador-Haim et al. [2012], the propagation of a write w_2 to a thread T before that thread T makes a write w_1 to the same address available to the memory system implies that w_2 co-precedes w_1. Thus, if the local w_1 co-precedes the external w_2, it must be that w_2 propagates to T after w_1 is committed. In turn, this implies that any read that reads from the external w_2 is satisfied after the local w_1 is committed.

Commits. Commits relate read and write events by program order if they access the same memory location—that is, they related by po-loc. Thus, in Figure 25, cc_0 contains po-loc. We inherit this constraint from Mador-Haim et al. [2012]. From the

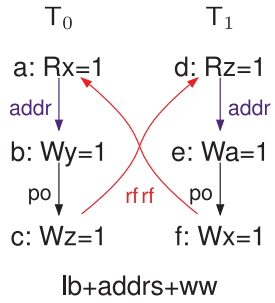


Fig. 29. A variant of the load buffering pattern lb (forbidden, see also Figure 7).

implementation standpoint, the core has to perform some actions to enforce the SC PER LOCATION check—that is, the five patterns of Figure 6. One way to achieve this could be to perform these actions at commit time, which this bit of the ppo represents.

However, note that our model performs the SC PER LOCATION check independently of the definition of the ppo. Thus, the present commit-to-commit ordering constraint is not required to enforce this particular axiom of the model. Nevertheless, if this bit of the ppo definition is adopted, it will yield the specific ordering constraint po-loc $\in$ cc_0. As we shall see in Section 8.1.2, this very constraint needs to be relaxed to account for some behaviours observed on Qualcomm systems.

In addition, commit read and commit read or write ($cc_0 \cap$ RM) relate by address, data, and control dependencies—that is, dp and ctrl.

Here and later (see Figure 29), we express "chains of irreversibility," meaning that when some memory access depends, by whatever means, on a po-preceding read, it can become irrevocable only when the read it depends upon has itself become irrevocable.

Finally, a read r must be committed before the commit of any access e that is program order after any access e' that is addr-after r, as in the lb+addrs+ww[4] pattern in Figure 29.

In the thread T_0 of Figure 29, the write to z cannot be read by another thread before the write b knows its own address, which in turn requires the read a to be committed because of the address dependency between a and b. Indeed, if the address of b was z, the two writes b and c to z could violate the coWW pattern, an instance of our SC PER LOCATION axiom. Hence, the commit of the write c has to be delayed (at least) until after the read a is committed.

Note that the same pattern with data instead of address dependencies is allowed and observed (see `http://diy.inria.fr/cats/data-addr`).

7. OPERATIONAL MODELS

We introduce here an operational equivalent of our model, our *intermediate machine*, inspired by our earlier work [Alglave et al. 2013a] and given in Figure 30, which we then compare to a previous model, the *PLDI machine* of Sarkar et al. [2011].

Outline of This Section. We want to show that a path allowed by the PLDI machine corresponds to an execution valid in our model.

Note that the converse direction (that any execution valid in our model corresponds to a path allowed by the PLDI machine) is not desirable.

[4]We note that this pattern does not quite follow the naming convention that we have outlined in Table III, but we keep this name for more consistency with Sarkar et al. [2011, 2012] and Mador-Haim et al. [2012].

Input data: $(\mathsf{ppo}, \mathsf{fences}, \mathsf{prop})$, $(\mathbb{E}, \mathsf{po})$ and p

Derived from p: $\mathsf{co}(\mathbb{E}, \mathsf{p}) \triangleq \{(w_1, w_2) \mid \mathrm{addr}(w_1) = \mathrm{addr}(w_2) \wedge (\mathrm{cp}(w_1), \mathrm{cp}(w_2)) \in \mathsf{p}\}$

COMMIT WRITE

$$\frac{\begin{array}{cc}(\text{CW: SC PER LOCATION/coWW}) & (\text{CW: PROPAGATION}) \\ \neg(\exists w' \in \mathsf{cw} \text{ s.t. } (w, w') \in \mathsf{po\text{-}loc}) & \neg(\exists w' \in \mathsf{cw} \text{ s.t. } (w, w') \in \mathsf{prop}) \\ \multicolumn{2}{c}{(\text{CW: fences} \cap \text{WR})} \\ \multicolumn{2}{c}{\neg(\exists r \in \mathsf{sr} \text{ s.t. } (w, r) \in \mathsf{fences})}\end{array}}{s \xrightarrow{\mathrm{c}(w)} (\mathsf{cw} \cup \{w\}, \mathsf{cpw}, \mathsf{sr}, \mathsf{cr})}$$

WRITE REACHES COHERENCE POINT

$$\frac{\begin{array}{c}(\text{CPW: WRITE IS COMMITTED}) \\ w \in \mathsf{cw} \\ \begin{array}{cc}(\text{CPW: po-loc AND cpw ARE IN ACCORD}) & (\text{CPW: PROPAGATION}) \\ \neg(\exists w' \in \mathsf{udr}(\mathsf{cpw}) \text{ s.t. } (w, w') \in \mathsf{po\text{-}loc}) & \neg(\exists w' \in \mathsf{udr}(\mathsf{cpw}) \text{ s.t. } (w, w') \in \mathsf{prop})\end{array}\end{array}}{s \xrightarrow{\mathrm{cp}(w)} (\mathsf{cw}, \mathsf{cpw} +\!\!+ [w], \mathsf{sr}, \mathsf{cr})}$$

SATISFY READ

$$\frac{\begin{array}{c}(\text{SR: WRITE IS EITHER LOCAL OR COMMITTED}) \\ (w, r) \in \mathsf{po\text{-}loc} \vee w \in \mathsf{cw} \\ \begin{array}{cc}(\text{SR: PPO/ii}_0 \cap \text{RR}) & (\text{SR: OBSERVATION}) \\ \neg(\exists r' \in \mathsf{sr} \text{ s.t. } (r, r') \in \mathsf{ppo} \cup \mathsf{fences}) & \neg(\exists w' \text{ s.t. } (w, w') \in \mathsf{co} \wedge (w', r) \in \mathsf{prop}; \mathsf{hb}^*)\end{array}\end{array}}{s \xrightarrow{\mathrm{s}(w,r)} (\mathsf{cw}, \mathsf{cpw}, \mathsf{sr} \cup \{r\}, \mathsf{cr})}$$

COMMIT READ

$$\frac{\begin{array}{cc}(\text{CR: READ IS SATISFIED}) & (\text{CR: SC PER LOCATION/coWR, coRW\{1,2\}, coRR}) \\ r \in \mathsf{sr} & \mathrm{visible}(w, r) \\ (\text{CR: PPO/cc}_0 \cap \text{RW}) & (\text{CR: PPO/(ci}_0 \cup \text{cc}_0) \cap \text{RR}) \\ \neg(\exists w' \in \mathsf{cw} \text{ s.t. } (r, w') \in \mathsf{ppo} \cup \mathsf{fences}) & \neg(\exists r' \in \mathsf{sr} \text{ s.t. } (r, r') \in \mathsf{ppo} \cup \mathsf{fences})\end{array}}{s \xrightarrow{\mathrm{c}(w,r)} (\mathsf{cw}, \mathsf{cpw}, \mathsf{sr}, \mathsf{cr} \cup \{r\})}$$

Fig. 30. Intermediate machine.

Indeed, the PLDI machine forbids behaviours observed on Power hardware (see `http://diy.inria.fr/cats/pldi-power/#lessvs`). Moreover, although the PLDI machine was not presented as such, it was thought to be a plausible ARM model. Yet, the PLDI machine forbids behaviours observed on ARM hardware (see `http://diy.inria.fr/cats/pldi-arm/#lessvs`). Our proof has two steps. We first prove our axiomatic model and this intermediate machine equivalent:

THEOREM 7.1. *All behaviours allowed by the axiomatic model are allowed by the intermediate machine and conversely.*

Then we show that any path allowed by the PLDI machine is allowed by our intermediate machine:

LEMMA 7.2. *Our intermediate machine allows all behaviours allowed by the PLDI machine.*

Thus, by Theorems 7.1 and 7.3, a path allowed by the PLDI machine corresponds to an axiomatic execution:

THEOREM 7.3. *Our axiomatic model allows all behaviours allowed by the PLDI machine.*

7.1. Intermediate Machine

Our intermediate machine is simply a reformulation of our axiomatic model as a transition system. We give its formal definition in Figure 30, which we explain later. In this section, we write udr(r) for the union of the domain and the range of the relation r. We write r++[e] to indicate that we append the element e at the end of a total order r.

Like the axiomatic model, our machine takes as input the events $\mathbb{E}$, the program order po, and an architecture (ppo, fences, prop).

It also needs a *path of labels*—that is, a total order over labels; a label triggers a transition of the machine. We borrow the names of the labels from Sarkar et al. [2011, 2012] for consistency.

We build the labels from the events $\mathbb{E}$ as follows: a write w corresponds to a *commit write* label c(w) and a *write reaching coherence point* label cp(w); a read r first guesses from which write w it might read, in an angelic manner; the pair (w, r) then yields two labels: *satisfy read* s(w, r) and *commit read* c(w, r).

For the architecture's functions ppo, fences, and prop to make sense, we need to build a coherence order and a read-from map. We define them from the events $\mathbb{E}$ and the path p as follows:

—co$(\mathbb{E}, \mathsf{p})$ gathers the writes to the same memory location in the order that their corresponding coherence point labels have in p: co$(\mathbb{E}, \mathsf{p}) \triangleq \{(w_1, w_2) \mid \text{addr}(w_1) = \text{addr}(w_2) \wedge (\text{cp}(w_1), \text{cp}(w_2)) \in \mathsf{p}\}$;
—rf$(\mathbb{E}, \mathsf{p})$ gathers the write-read pairs with same location and value that have a commit label in p: rf$(\mathbb{E}, \mathsf{p}) \triangleq \{(w, r) \mid \text{addr}(w) = \text{addr}(r) \wedge \text{val}(w) = \text{val}(r) \wedge \text{c}(w, r) \in \text{udr}(\mathsf{p})\}$.

In the definition of our intermediate machine, we consider the relations defined w.r.t. co and rf in the axiomatic model (e.g., happens-before hb, propagation prop) to be defined w.r.t. the coherence and read-from mentioned earlier.

Now, our machine operates over a state (cw, cpw, sr, cr) composed of

—A set cw ("committed writes") of writes that have been committed;
—A relation cpw over writes having reached coherence point, which is a total order per location;
—A set sr ("satisfied reads") of reads having been satisfied; and
—A set cr ("committed reads") of reads having been committed.

7.1.1. Write Transitions. The order in which writes enter the set cw for a given location corresponds to the coherence order for that location. Thus, a write w cannot enter cw if it contradicts the SC PER LOCATION and PROPAGATION axioms. Formally, a commit write c(w) yields a commit write transition, which appends w at the end of cw for its location if

—(CW: SC PER LOCATION/coWW) there is no po-loc-subsequent write w' that is already committed, which forbids the coWW case of the SC PER LOCATION axiom, and
—(CW: PROPAGATION) there is no prop-subsequent write w' that is already committed, which ensures that the PROPAGATION axiom holds, and
—(CW: fences $\cap$ WR) there is no fences-subsequent read r that has already been satisfied, which contributes to the semantics of the full fence.

A write can reach coherence point (i.e., take its place at the end of the cpw order) if.

—(CPW: WRITE IS COMMITTED) the write has already been committed, and
—(CPW: po-loc AND cpw ARE IN ACCORD) all of the po-loc-previous writes have reached coherence point, and
—(CPW: PROPAGATION) all of the prop-previous writes have reached coherence point.

This ensures that the order in which writes reach coherence point is compatible with the coherence order and the propagation order.

7.1.2. Read Transitions. The read set sr is ruled by the happens-before relation hb. The way reads enter sr ensures NO THIN AIR and OBSERVATION, whilst the way they enter cr ensures parts of SC PER LOCATION and of the preserved program order.

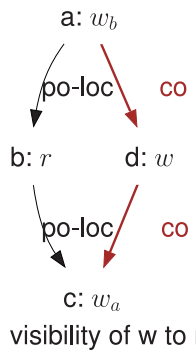
Satisfy Read. The satisfy read transition $\mathsf{s}(w, r)$ places the read r in sr if.

—(SR: WRITE IS EITHER LOCAL OR COMMITTED) the write w from which r reads is either local (i.e., w is po-loc-before r), or has been committed already (i.e., $w \in \mathsf{cw}$), and
—(SR: PPO/$\mathsf{ii}_0 \cap \mathsf{RR}$) there is no $(\mathsf{ppo} \cup \mathsf{fences})$-subsequent read r' that has already been satisfied, which implements the $\mathsf{ii}_0 \cap \mathsf{RR}$ part of the preserved program order, and
—(SR: OBSERVATION) there is no write w' co-after w was[5] s.t. $(w', r) \in \mathsf{prop}; \mathsf{hb}^*$, which ensures OBSERVATION.

Commit Read. To define the commit read transition, we need a preliminary notion. We define a write w to be *visible* to a read r when

—w and r share the same location ℓ;
—w is equal to, or co-after[5], the last write w_b to ℓ that is po-loc-before r; and
—w is po-loc-before r, or co-before the first write w_a to ℓ that is po-loc-after r.

Here, we give an illustration of the case where w is co-after w_b and before w_a:



Now, recall that our read labels contain both a read r and a write w that it might read. The commit read transition $\mathsf{c}(w, r)$ records r in cr when

—(CR: READ IS SATISFIED) r has been satisfied (i.e. is in sr); and
—(CR: SC PER LOCATION/ coWR, coRW{1,2}, coRR) w is *visible* to r, which prevents the coWR, coRW1 and coRW2 cases of SC PER LOCATION; and
—(CR: PPO/$\mathsf{cc}_0 \cap \mathsf{RW}$) there is no $(\mathsf{ppo} \cup \mathsf{fences})$-subsequent write w' that has already been committed, which implements the $\mathsf{cc}_0 \cap \mathsf{RW}$ part of the preserved program order; and

[5]Recall that in the context of our intermediate machine, $(w, w') \in \mathsf{co}$ means that w and w' are relative to the same address and $(\mathsf{cp}(w), \mathsf{cp}(w')) \in \mathsf{p}$.

—(CR: PPO/($\mathsf{ci}_0 \cup \mathsf{cc}_0$)$\cap$RR) there is no (ppo$\cup$fences)-subsequent read r' that has already been satisfied, which implements the $\mathsf{ci}_0 \cap \mathsf{RR}$ and $\mathsf{cc}_0 \cap \mathsf{RR}$ parts of the preserved program order.

To forbid the coRR case of SC PER LOCATION, one needs to (i) make the read set cr record the write from which a read takes its value so that cr is a set of pairs (w, r) and no longer just a set of reads; and (ii) augment the definition of visible(w, r) to require that there is no (w', r') s.t. r' is po-loc-before r yet w' is co-after w. We chose to present the simpler version of the machine to ease the reading.

7.2. Equivalence of the Axiomatic Model and Intermediate Machine (Proof of Theorem 7.1)

We prove Theorem 7.1 in two steps. We first show that given a set of events $\mathbb{E}$, a program order po over these, and a path p over the corresponding labels such that $(\mathbb{E}, \mathsf{po}, \mathsf{p})$ is accepted by our intermediate machine, the axiomatic execution $(\mathbb{E}, \mathsf{po}, \mathsf{co}(\mathbb{E}, \mathsf{p}), \mathsf{rf}(\mathbb{E}, \mathsf{p}))$ is valid in our axiomatic model:

LEMMA 7.4. *All behaviours allowed by the intermediate machine are allowed by the axiomatic model.*

Conversely, from a valid axiomatic execution $(\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$, we build a path accepted by the intermediate machine:

LEMMA 7.5. *All behaviours allowed by the axiomatic model are allowed by the intermediate machine.*

We give below the main arguments for proving our results. For more confidence, we have implemented the equivalence proof between our axiomatic model and our intermediate machine in the Coq proof assistant [Bertot and Casteran 2004]. We provide our proof scripts online: `http://diy.inria.fr/cats/proofs`.

7.2.1. From Intermediate Machine to Axiomatic Model (Proof of Lemma 7.4). We show here that a path of labels p relative to a set of events $\mathbb{E}$ and a program order po accepted by our intermediate machine leads to a valid axiomatic execution. To do so, we show that the execution $(\mathbb{E}, \mathsf{po}, \mathsf{co}(\mathbb{E}, \mathsf{p}), \mathsf{rf}(\mathbb{E}, \mathsf{p}))$ is valid in our axiomatic model.

Well-formedness of $\mathsf{co}(\mathbb{E}, p)$ (i.e., co is a total order on writes to the same location) follows from p being a total order.

Well-formedness of $\mathsf{rf}\ (\mathbb{E}, p)$ (i.e., rf relates a read to a unique write to the same location with same value) follows from the fact that we require that each read r in $\mathbb{E}$ has a unique corresponding write w in $\mathbb{E}$, s.t. r and w have same location and value, and $\mathsf{c}(w, r)$ is a label of p.

The SC PER LOCATION *Axiom Holds*: To prove this, we show that coWW, coRW1, coRW2, coWR, and coRR are forbidden.

coWW is forbidden: Suppose as a contradiction two writes e_1 and e_2 to the same location s.t. $(e_1, e_2) \in \mathsf{po}$ and $(e_2, e_1) \in \mathsf{co}(\mathbb{E}, \mathsf{p})$. The first hypothesis entails that $(\mathsf{cp}(e_1), \mathsf{cp}(e_2)) \in \mathsf{p}$; otherwise, we would contradict the premise (CPW: po-loc AND cpw ARE IN ACCORD) of the WRITE REACHES COHERENCE POINT rule. The second hypothesis means by definition that $(\mathsf{cp}(e_2), \mathsf{cp}(e_1)) \in \mathsf{p}$. This contradicts the acyclicity of p.

coRW1 is forbidden: Suppose as a contradiction a read r and a write w relative to the same location, s.t. $(r, w) \in \mathsf{po}$ and $(w, r) \in \mathsf{rf}\ (\mathbb{E}, \mathsf{p})$. Thus, w cannot be visible to r as it is po-after r. This contradicts the premise (CR: SC PER LOCATION/ coWR, coRW{1,2}, coRR) of the COMMIT READ rule.

coRW2 is forbidden: Suppose as a contradiction a read r and two writes w_1 and w_2 relative to the same location, s.t. $(r, w_2) \in \mathsf{po}$ and $(w_2, w_1) \in \mathsf{co}(\mathbb{E}, \mathsf{p})$ and $(w_1, r) \in \mathsf{rf}(\mathbb{E}, \mathsf{p})$. Thus, w_1 cannot be visible to r as it is co-after w_2, w_2 itself being either equal or co-after the first write w_a in po-loc after r. This contradicts the premise (CR: SC PER LOCATION/ coWR, coRW{1,2}, coRR) of the COMMIT READ rule.

coWR is forbidden: Suppose as a contradiction two writes w_0 and w_1 and a read r relative to the same location, s.t. $(w_1, r) \in \mathsf{po}$, $(w_0, r) \in \mathsf{rf}(\mathbb{E}, \mathsf{p})$, and $(w_0, w_1) \in \mathsf{co}(\mathbb{E}, \mathsf{p})$. Thus, w_0 cannot be visible to r as it is co-before w_1, w_1 being itself either equal or co-before the last write w_b in po-loc before r. This contradicts the premise (CR: SC PER LOCATION/ coWR, coRW{1,2}, coRR) of the COMMIT READ rule.

coRR is forbidden (note that this holds only for the modified version of the machine outlined at the end of Section 7.1): Suppose as a contradiction two writes w_1 and w_1 and two reads r_1 and r_2 relative to the same location, s.t. $(r_1, r_2) \in \mathsf{po}$, $(w_1, r_1) \in \mathsf{rf}$, $(w_2, r_2) \in \mathsf{rf}$, and $(w_2, w_1) \in \mathsf{co}$. Thus, w_2 cannot be visible to r_2 as it is co-before w_1 (following their order in co), and r_1 is po-loc-before r_2. This contradicts the premise (CR: SC PER LOCATION/ coWR, coRW{1,2}, coRR) of the COMMIT READ rule.

The NO THIN AIR *Axiom Holds*: Suppose as a contradiction that there is a cycle in hb^+—that is, there is an event x s.t. $(x, x) \in ((\mathsf{ppo} \cup \mathsf{fences}); \mathsf{rfe})^+$. Thus, there exists y s.t. (i) $(x, y) \in (\mathsf{ppo} \cup \mathsf{fences}); \mathsf{rfe}$ and (ii) $(y, x) \in ((\mathsf{ppo} \cup \mathsf{fences}); \mathsf{rfe})^+$. Note that x and y are reads, since the target of rf is always a read.

We now show that (iii) for two reads r_1 and r_2, having $(r_1, r_2) \in ((\mathsf{ppo} \cup \mathsf{fences}); \mathsf{rfe})^+$ implies $(\mathrm{s}(r_1), \mathrm{s}(r_2)) \in \mathsf{p}$. We are abusing our notations here, writing $\mathrm{s}(r)$ instead of $\mathrm{s}(w, r)$, where w is the write from which r reads. From the fact (iii) and the hypotheses (i) and (ii), we derive a cycle in p, a contradiction since p is an order.

Proof of (iii): Let us show the base case; the inductive case follows immediately. Let us have two reads r_1 and r_2 s.t. $(r_1, r_2) \in (\mathsf{ppo} \cup \mathsf{fences}); \mathsf{rfe}$. Thus, there is w_2 s.t. $(r_1, w_2) \in \mathsf{ppo} \cup \mathsf{fences}$ and $(w_2, r_2) \in \mathsf{rfe}$. Now, note that when we are about to take the SATISFIED READ transition triggered by the label $\mathrm{s}(w_2, r_2)$, we know that the premise (SR: WRITE IS EITHER LOCAL OR COMMITTED) holds. Thus, we know that either w_2 and r_2 belong to the same thread, which immediately contradicts the fact that they are in rfe, or that w_2 has been committed. Therefore, we have (iv) $(\mathrm{c}(w_2), \mathrm{s}(r_2)) \in \mathsf{p}$. Moreover, we can show that (v) $(\mathrm{s}(r_1), \mathrm{c}(w_2)) \in \mathsf{p}$ by the fact that $(r_1, w_2) \in \mathsf{ppo} \cup \mathsf{fences}$. Thus, by (iv) and (v) we have our result.

Proof of (v): Take a read r and a write w s.t. $(r, w) \in \mathsf{ppo} \cup \mathsf{fences}$. We show below that (vi) $(\mathrm{c}(r), \mathrm{c}(w)) \in \mathsf{p}$. Since it is always the case that $(\mathrm{s}(r), \mathrm{c}(r)) \in \mathsf{p}$ (thanks to the fact that a read is satisfied before it is committed, see premise (CR: READ IS SATISFIED) of the COMMIT READ rule), we can conclude. Now for (vi): Since p is total, we have either our result or $(\mathrm{c}(w), \mathrm{c}(r)) \in \mathsf{p}$. Suppose the latter. Then when we are about to take the COMMIT READ transition triggered by $\mathrm{c}(r)$, we contradict the premise (CR: PPO/$\mathsf{cc}_0 \cap \mathsf{RW}$). Indeed, we have $w \in \mathsf{cw}$ by $\mathrm{c}(w)$ preceding $\mathrm{c}(r)$ in p, and $(r, w) \in \mathsf{ppo} \cup \mathsf{fences}$ by hypothesis.

The OBSERVATION *Axiom Holds*: Suppose as a contradiction that $\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*$ is not irreflexive—that is, there are w and r s.t. $(r, w_2) \in \mathsf{fre}$ and $(w_2, r) \in \mathsf{prop}; \mathsf{hb}^*$. Note that $(r, w_2) \in \mathsf{fr}$ implies the existence of a write w_1 s.t. $(w_1, w_2) \in \mathsf{co}$ and $(w_1, r) \in \mathsf{rf}$. Observe that this entails that r, w_1, and w_2 are relative to the same location.

Thus, we take two writes w_1, w_2 and a read r relative to the same location s.t. $(w_1, w_2) \in \mathsf{co}$, $(w_1, r) \in \mathsf{rf}$ and $(w_2, r) \in \mathsf{prop}; \mathsf{hb}^*$ as earlier. This contradicts the (SR: OBSERVATION) hypothesis. Indeed, when we are about to process the transition triggered by the label $\mathrm{s}(r)$, we have $(w_2, r) \in \mathsf{prop}; \mathsf{hb}^*$ by hypothesis, and $(w_1, w_2) \in \mathsf{co}$ by hypothesis.

The PROPAGATION *Axiom Holds*: Suppose as a contradiction that there is a cycle in $(\mathsf{co} \cup \mathsf{prop})^+$—that is, there is an event x s.t. $(x, x) \in (\mathsf{co} \cup \mathsf{prop})^+$. In other terms, there is y s.t. (i) $(x, y) \in \mathsf{co}; \mathsf{prop}$ and (ii) $(y, x) \in (\mathsf{co}; \mathsf{prop})^+$. Note that x and y are writes, since the source of co is always a write.

We now show that (iii) for two writes w_1 and w_2, having $(w_1, w_2) \in (\mathsf{co}; \mathsf{prop})^+$ implies $(\mathrm{cp}(w_1), \mathrm{cp}(w_2)) \in \mathsf{p}^+$. From the fact (iii) and the hypotheses (i) and (ii), we derive a cycle in p, a contradiction since p is an order.

Proof of (iii): Let us show the base case; the inductive case follows immediately. Let us have two writes w_1 and w_2 s.t. $(w_1, w_2) \in \mathsf{co}; \mathsf{prop}$; thus, there is a write w s.t. $(w_1, w) \in \mathsf{co}$ and $(w, w_2) \in \mathsf{prop}$. Since p is total, we have either the result or $(\mathrm{cp}(w_2), \mathrm{cp}(w_1)) \in \mathsf{p}$. Suppose the latter. Thus, we contradict the (CPW: PROPAGATION) hypothesis. Indeed, when we are about to take the WRITE REACHES COHERENCE POINT transition triggered by the label $\mathrm{cp}(w)$, we have $(w, w_2) \in \mathsf{prop}$ by hypothesis, and w_2 already in cpw. The hypothesis $(w_1, w) \in \mathsf{co}$ entails that $(\mathrm{cp}(w_1), \mathrm{cp}(w)) \in \mathsf{p}$, and we also have $(\mathrm{cp}(w_2), \mathrm{cp}(w_1)) \in \mathsf{p}$ by hypothesis. Therefore, when we are about to process $\mathrm{cp}(w)$, we have placed w_2 in cpw by taking the transition $\mathrm{cp}(w_2)$.

7.2.2. From Axiomatic Model to Intermediate Machine (Proof of Lemma 7.5). We show here that an axiomatic execution $(\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ leads to a valid path p of the intermediate machine. To do so, we show that the intermediate machine accepts certain paths[6] that linearise the transitive closure of the relation r defined inductively as follows (we abuse our notations here, writing, for example, $\mathrm{c}(r)$ instead of $\mathrm{c}(w, r)$, where w is the write from which r reads):

—For all $r \in \mathbb{E}$, $(\mathrm{s}(r), \mathrm{c}(r)) \in \mathsf{r}$ (i.e., we satisfy a read before committing it);
—For all $w \in \mathbb{E}$, $(\mathrm{c}(w), \mathrm{cp}(w)) \in \mathsf{r}$ (i.e., we commit a write before it can reach its coherence point);
—For all w and r separated by a fence in program order, $(\mathrm{c}(w), \mathrm{s}(r)) \in \mathsf{r}$ (i.e., we commit the write w before we satisfy the read r);
—For all $(w, r) \in \mathsf{rfe}$, $(\mathrm{c}(w), \mathrm{s}(w, r)) \in \mathsf{r}$ (i.e., we commit a write before reading externally from it);
—For all $(w_1, w_2) \in \mathsf{co}$, $(\mathrm{cp}(w_1), \mathrm{cp}(w_2)) \in \mathsf{r}$ (i.e., cpw and the coherence order are in accord);
—For all $(r, e) \in \mathsf{ppo} \cup \mathsf{fences}$, we commit a read r before processing any other event e (i.e., satisfying e if e is a read, or committing e if e is a write), if r and e are separated, for example, by a dependency or a barrier; and
—For all $(w_1, w_2) \in \mathsf{prop}^+$, $(\mathrm{cp}(w_1), \mathrm{cp}(w_2)) \in \mathsf{r}$ (i.e., cpw and propagation order are in accord).

Since we build p as a linearisation of the relation r defined earlier, we first need to show that we are allowed to linearise r (i.e., that r is acyclic).

Linearisation of r. Suppose as a contradiction that there is a cycle in r—that is, there is a label l s.t. $(l, l) \in \mathsf{r}^+$. Let us write S_1 for the set of commit writes, satisfy reads, and commit reads, and S_2 for the set of writes reaching coherence points. We show by induction that for all pair of labels $(l_1, l_2) \in \mathsf{r}^+$, either:

—l_1 and l_2 are both in S_1, and their corresponding events e_1 and e_2 are ordered by happens-before (i.e., $(e_1, e_2) \in \mathsf{hb}^+$), or

[6]The path has to linearise r so that for all writes w_1 and w_2, if $(\mathrm{cp}(w_1), \mathrm{cp}(w_2)) \in \mathsf{p}$, then $(\mathrm{c}(w_1), \mathrm{c}(w_2)) \in \mathsf{p}$. We refer to this property as "p being fifo." In other words, the linearisation must be such that coherence point labels and commit labels are in accord. Note that this does not affect the generality of Lemma 7.5, as to prove this lemma, we only need to find one valid intermediate machine path for a given axiomatic execution; our path happens to be so that coherence point and commit labels are in accord.

—l_1 and l_2 are both in S_2 and their corresponding events e_1 and e_2 are ordered by $(\mathsf{co} \cup \mathsf{prop})^+$, or
—l_1 is in S_1, l_2 in S_2, and their corresponding events e_1 and e_2 are ordered by happens-before, or
—l_1 is in S_1, l_2 in S_2, and their corresponding events e_1 and e_2 are ordered by $(\mathsf{co} \cup \mathsf{prop})^+$, or
—l_1 is in S_1, l_2 in S_2, and their corresponding events e_1 and e_2 are ordered by $\mathsf{hb}^+; (\mathsf{co} \cup \mathsf{prop})^+$, or
—l_1 is a satisfy read and l_2 the corresponding commit read.
—l_1 is a commit write and l_2 the write reaching coherence point.

Each of these items contradicts the fact that $l_1 = l_2$: the first two resort to the axioms of our model prescribing the acyclicity of hb on the one hand (NO THIN AIR), and co ∪ prop on the second hand (PROPAGATION); all the rest resort to the transitions being different.

We now show that none of the transitions of the machine given in Figure 30 can block.

COMMIT WRITE *Does Not Block*. Suppose as a contradiction a label $\mathsf{c}(w)$ s.t. the transition of the intermediate machine triggered by $\mathsf{c}(w)$ blocks. This means that one of the premises of the COMMIT WRITE rule is not satisfied.

First case: The premise (CW: SC PER LOCATION/coWW) is not satisfied—that is, there exists w' in cw s.t. $(w, w') \in$ po-loc. Since $(w, w') \in$ po-loc, we have $(w, w') \in$ co by SC PER LOCATION. By construction of p, we deduce $(\mathsf{cp}(w), \mathsf{cp}(w')) \in$ p. By p being fifo (see footnote on p. 41), we deduce (i) $(\mathsf{c}(w), \mathsf{c}(w')) \in$ p. Moreover, if w' is in cw when we are about to process $\mathsf{c}(w)$, then w' has been committed before w, hence (ii)$(\mathsf{c}(w'), \mathsf{c}(w)) \in$ p. By (i) and (ii), we derive a cycle in p, a contradiction since p is an order (since we build it as a linearisation of a relation).

Second case: The premise (CW: PROPAGATION) is not satisfied—that is, there exists w' in cw s.t. $(w, w') \in$ prop. Since w' is in cw when we are about to process the label $\mathsf{c}(w)$, we have (i) $(\mathsf{c}(w'), \mathsf{c}(w)) \in$ p. Since $(w, w') \in$ prop, we have (ii) $(\mathsf{cp}(w), \mathsf{cp}(w')) \in$ p (recall that we build p inductively; in particular, the order of the $\mathsf{cp}(w)$ transitions in p respects the order of the corresponding events in prop). Since we build p so that it is fifo, we deduce from (i) the fact (iii) $(\mathsf{c}(w), \mathsf{c}(w')) \in$ p. From (ii) and (iii), we derive a cycle in p, a contradiction since p is an order.

Third case: The premise (CW: fences ∩ WR) is not satisfied—that is, there exists r in sr s.t. $(w, r) \in$ fences. From $r \in$ sr, we deduce $(\mathsf{s}(r), \mathsf{c}(w)) \in$ p. From $(w, r) \in$ fences, we deduce (by construction of p) $(\mathsf{c}(w), \mathsf{s}(r)) \in$ p, which creates a cycle in p.

WRITE REACHES COHERENCE POINT *Does Not Block*. Suppose as a contradiction a label $\mathsf{cp}(w)$ s.t. the transition of the intermediate machine triggered by $\mathsf{cp}(w)$ blocks. This means that one of the premises of the WRITE REACHES COHERENCE POINT rule is not satisfied.

First case: The premise (CPW: WRITE IS COMMITTED) is not satisfied—that is, w has not been committed. This is impossible since $(\mathsf{c}(w), \mathsf{cp}(w)) \in$ p by construction of p.

Second case: The premise (CPW:po-loc AND cpw ARE IN ACCORD) is not satisfied—that is, there is a write w' that has reached coherence point s.t. $(w, w') \in$ po-loc. From $(w, w') \in$ po-loc, we know by SC PER LOCATION that $(w, w') \in$ co. Thus, by construction of p, we know (i) $(\mathsf{cp}(w), \mathsf{cp}(w')) \in$ p. From w' having reached coherence point before w, we know (ii) $(\mathsf{cp}(w'), \mathsf{cp}(w)) \in$ p. By (i) and (ii), we derive a cycle in p, a contradiction since p is an order.

Third case: The premise (CPW: PROPAGATION) is not satisfied—that is, there is a write w' that has reached coherence point s.t. $(w, w') \in$ prop. By construction of p, we deduce (i) $(\mathsf{cp}(w), \mathsf{cp}(w') \in$ p). From w' having reached coherence point before w, we know (ii)

$(\mathrm{cp}(w'), \mathrm{cp}(w)) \in \mathsf{p}$. By (i) and (ii), we derive a cycle in p, a contradiction since p is an order (since we build it as a linearisation of a relation).

SATISFY READ *Does Not Block*. Suppose as a contradiction a label $\mathrm{s}(w, r)$ s.t. the transition of the intermediate machine triggered by $\mathrm{s}(w, r)$ blocks. This means that one of the premises of the SATISFY READ rule is not satisfied. Note that since $\mathrm{s}(w, r)$ is a label of p, we have (i) $(w, r) \in \mathsf{rf}$.

First case: The premise (SR: WRITE IS EITHER LOCAL OR COMMITTED) is not satisfied—that is, w is neither local nor committed. Suppose that w not local (otherwise, we contradict our hypothesis); let us show that it has to be committed. Suppose that it is not; therefore, we have $(\mathrm{s}(r), \mathrm{c}(w)) \in \mathsf{p}$. Since w is not local, we have $(w, r) \in \mathsf{rfe}$, from which we deduce (by construction of p) that $(\mathrm{c}(w), \mathrm{s}(r)) \in \mathsf{p}$; this leads to a cycle in p.

Second case: The premise (SR: PPO/$\mathsf{ii}_0 \cap \mathsf{RR}$) is not satisfied—that is, there is a satisfied read r' s.t. $(r, r') \in \mathsf{ppo} \cup \mathsf{fences}$. From r' being satisfied, we deduce $(\mathrm{s}(r'), \mathrm{s}(r)) \in \mathsf{p}$. From $(r, r') \in \mathsf{ppo} \cup \mathsf{fences}$ and by construction of p, we deduce $(\mathrm{c}(r), \mathrm{s}(r')) \in \mathsf{p}$. Since $\mathrm{s}(r)$ precedes $\mathrm{c}(r)$ in p by construction of p, we derive a cycle in p, a contradiction.

Third case: The premise (SR: OBSERVATION) is not satisfied—that is, there is a write w' s.t. $(w, w') \in \mathsf{co}$ and $(w', r) \in \mathsf{prop}; \mathsf{hb}^*$. Since $(w, w') \in \mathsf{co}$, by (i) $(r, w') \in \mathsf{fr}$. Therefore, we contradict the OBSERVATION axiom.

COMMIT READ *Does Not Block*. Suppose as a contradiction a label $\mathrm{c}(w, r)$ s.t. the transition of the intermediate machine triggered by $\mathrm{c}(w, r)$ blocks. This means that one of the premises of the COMMIT READ rule is not satisfied.

First case: The premise (CR: READ IS SATISFIED) is not satisfied—that is, r is not in sr. This is impossible since we impose $(\mathrm{s}(r), \mathrm{c}(r)) \in \mathsf{p}$ when building p.

Second case: The premise (CR: SC PER LOCATION/ coWR, coRW{1,2}, coRR) is not satisfied—that is, w is not visible to r. This contradicts the SC PER LOCATION axiom, as follows. Recall that the visibility definition introduces w_a as the first write to the same location as r that is po-loc-after r; and w_b as the last write to the same location as r that is po-loc-before r. Now, if w is not visible to r, we have either (i) w is co-before w_b, or (ii) equal or co-after w_a.

Suppose (i), that we have $(w, w_b) \in \mathsf{co}$. Hence, we have $(w, w_b) \in \mathsf{co}$, $(w_b, r) \in \mathsf{po\text{-}loc}$ by definition of w_b, and $(w, r) \in \mathsf{rf}$ by definition of $\mathrm{s}(w, r)$ being in p. Thus, we contradict the coWR case of the SC PER LOCATION axiom. The (ii) case is similar and contradicts (coRW1) if $w = w_a$ or (coRW2) if $(w_a, w) \in \mathsf{cw}$.

Third case: The premise (CR: PPO/$\mathsf{cc}_0 \cap \mathsf{RW}$) is not satisfied—that is, there is a write w' in cw s.t. $(r, w') \in \mathsf{ppo} \cup \mathsf{fences}$. From $w' \in \mathsf{cw}$, we deduce $(\mathrm{c}(w'), \mathrm{c}(r)) \in \mathsf{p}$. From $(r, w') \in \mathsf{ppo} \cup \mathsf{fences}$, we deduce $(\mathrm{c}(r), \mathrm{c}(w')) \in \mathsf{p}$ by construction of p. This leads to a cycle in p, a contradiction.

Fourth case: The premise (CR: PPO/$(\mathsf{ci}_0 \cup \mathsf{cc}_0) \cap \mathsf{RR}$) is not satisfied—that is, there is a read r' in sr s.t. $(r, r') \in \mathsf{ppo} \cup \mathsf{fences}$. From $r' \in \mathsf{sr}$, we deduce $(\mathrm{s}(r'), \mathrm{c}(r)) \in \mathsf{p}$. From $(r, r') \in \mathsf{ppo} \cup \mathsf{fences}$, we deduce $(\mathrm{c}(r), \mathrm{s}(r')) \in \mathsf{p}$ by construction of p. This leads to a cycle in p, a contradiction.

7.3. Comparing Our Model and the PLDI Machine

The PLDI machine is an operational model, which we describe here briefly (see Sarkar et al. [2011] for details). This machine maintains a coherence order (a strict partial order over the writes to the same memory location) and, per thread, a list of the writes and fences that have been propagated to that thread.

A load instruction yields two transitions of this machine (amongst others): a *satisfy read* transition, where the read takes its value, and a *commit read* transition, where

the read becomes irrevocable. A store instruction yields a *commit write* transition, where the write becomes available to be read, several *propagate write* transitions, where the write is sent out to different threads of the system, and a *reaching coherence point* transition, where the write definitely takes place in the coherence order. We summarise the effect of a PLDI transition on a PLDI state in the course of this section.

We show that a valid path of the PLDI machine leads to valid path of our intermediate machine.

First, we show how to relate the two machines.

7.3.1. Mapping PLDI Objects (Labels and States) to Intermediate Objects. We write $\mathsf{pl2l}(l)$ to map a PLDI l to a label of the intermediate machine. For technical convenience, we assume a special noop intermediate label such that from any state s of the intermediate machine, we may perform a transition from s to s via noop .

We can then define $\mathsf{pl2l}(l)$ as being the eponymous label in the intermediate machine if it exists (i.e., for commit write, write reaches coherence point, satisfy read, and commit read), and noop otherwise. We derive from this mapping the set $\mathbb{L}_i$ of intermediate labels composing our intermediate path.

We build a state of our intermediate machine from a PLDI state s and an accepting PLDI path p; we write $\mathsf{ps2s}(\mathsf{p}, s) = (\mathsf{cw}, \mathsf{cpw}, \mathsf{sr}, \mathsf{cr})$ for the intermediate state built as follows:

—For a given location, cw is simply the set of writes to this location that have been committed in s;
—We take cpw to be all the writes having reached coherence point in s, ordered w.r.t. p;
—The set sr gathers the reads that have been satisfied in the state s: we simply add a read to sr if the corresponding satisfy transition appears in p before the transition leading to s;
—The set cr gathers the reads that have been committed in the state s.

7.3.2. Building a Path of the Intermediate Machine from a PLDI Path. A given path p of the PLDI machine entails a run $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \cdots \xrightarrow{l_n} s_n$ such that $(l, l') \in \mathsf{p}$ if and only if there exist i and j such that $i < j$ and $l = l_i$ and $l' = l_j$.

We show that $\mathsf{ps2s}(s_0) \xrightarrow{\mathsf{pl2l}(l_1)} \mathsf{ps2s}(s_1) \xrightarrow{\mathsf{pl2l}(l_2)} \cdots \xrightarrow{\mathsf{pl2l}(l_n)} \mathsf{ps2s}(s_n)$ is a path of our intermediate machine. We proceed by induction for $0 \leq m \leq n$. The base case $m = 0$ is immediately satisfied by the single-state path $\mathsf{ps2s}(s_0)$.

Now, inductively assume that $\mathsf{ps2s}(s_0) \xrightarrow{\mathsf{pl2l}(l_1)} \mathsf{ps2s}(s_1) \xrightarrow{\mathsf{pl2l}(l_2)} \cdots \xrightarrow{\mathsf{pl2l}(l_m)} \mathsf{ps2s}(s_m)$ is a path of the intermediate machine. We prove the case for $m + 1$. Take the transition $s_m \xrightarrow{l_{m+1}} s_{m+1}$ of the PLDI machine. We prove $\mathsf{ps2s}(s_m) \xrightarrow{\mathsf{pl2l}(l_{m+1})} \mathsf{ps2s}(s_{m+1})$ to complete the induction. There are several cases.

When $\mathsf{pl2l}(l_{m+1}) = \mathsf{noop}$, we have that $\mathsf{ps2s}(s_m) = \mathsf{ps2s}(s_{m+1})$ simply because the PLDI labels that have noop as an image by pl2l do not affect the components cw, cpw, sr, and cr of our state.

Only the PLDI transitions that have an eponymous transition in our machine affect our intermediate state. Thus, we list below the corresponding four cases.

Commit Write. In the PLDI machine, a commit transition of a write w makes this write co-after all writes to the same location that have been propagated to its thread. The PLDI transition guarantees that (i) w had not been committed in s_m.

Observe that w takes its place in the set cw, ensuring that we modify the state as prescribed by our COMMIT WRITE rule. Now, we check that we do not contradict the premises of our COMMIT WRITE rule.

First case: Contradict the premise (CW: SC PER LOCATION/ coWW)—that is, take a write w' in cw s.t. $(w, w') \in$ po-loc. In that case, we contradict the fact that the commit order respects po-loc (see Sarkar et al. [2011, p. 7, item 4 of § Commit in-flight instruction]).

Second case: Contradict the premise (CW: PROPAGATION)—that is, take a write w' in cw s.t. $(w, w') \in$ prop. In that case, $(w, w') \in$ prop guarantees that w was propagated to the thread of w' in s_m. Therefore (see Sarkar et al. [2011, p. 6, premise of § Propagate write to another thread]), w was seen in s_m. For the write to be seen, it needs to have been sent in a write request (see Sarkar et al. [2011, p. 6, item 1 of § Accept write request]); for the write request to be sent, the write must have been committed (see Sarkar et al. [2011, p. 8, action 4 of § Commit in-flight instruction]). Thus, we contradict (i).

Third case: Contradict the premise (CW: fences $\cap$ WR)—that is, take a read r in sr s.t. $(w, r) \in$ fences. Since r is in sr, r is satisfied. Note that the write from which r reads can be either local (see Sarkar et al. [2011, p. 8, § Satisfy memory read by forwarding an in-flight write directly to reading instruction]) or committed (see Sarkar et al. [2011, p. 8, § Satisfy memory read from storage subsystem]).

In both cases, the fence between w and r must have been committed (see Sarkar et al. [2011, p. 8, item 2 of § Satisfy memory read by forwarding an in-flight write directly to reading instruction and item 2 of § Satisfy memory read from storage subsystem]). Thus (by Sarkar et al. [2011, p. 7, item 6 of § Commit in-flight instruction]), w has been committed, a contradiction of (i).

Write Reaches Coherence Point. In the PLDI machine, write reaching coherence point transitions order writes following a linearisation of $(\text{co} \cup \text{prop})^+$. Our cw implements that per location, then we make the writes reach coherence point following cw and $(\text{co} \cup \text{prop})^+$.

Observe that a write w reaching coherence point takes its place after all writes having already reached coherence point, ensuring that we modify the intermediate state as prescribed by our WRITE REACHES COHERENCE POINT rule. Now, we check that we do not contradict the premises of WRITE REACHES COHERENCE POINT.

First case: Contradict the premise (CPW: WRITE IS COMMITTED)—that is, suppose that w is not committed. This is impossible as the PLDI machine requires a write to have been seen by the storage subsystem for it to reach coherence point (see Sarkar et al. [2012, p. 3, § Write reaches its coherence point]). For the write to be seen, it needs to have been sent in a write request (see Sarkar et al. [2011, p. 6, item 1 of § Accept write request]); for the write request to be sent, the write must have been committed (see Sarkar et al. [2011, p. 7, action 4 of § Commit in-flight instruction]).

Second case: Contradict the premise (CPW:po-loc AND cpw ARE IN ACCORD)—that is, take a write w' in cpw s.t. $(w, w') \in$ po-loc. This means that (i) w' has reached coherence point before w, despite w preceding w' in po-loc. This is a contradiction, as we now explain. If $(w, w') \in$ po-loc, then w is propagated to its own thread before w' (see Sarkar et al. [2011, p. 7, action 4 of § Commit in-flight instruction]). Since w and w' access the same address, when w' is propagated to its own thread, w' is recorded as being co-after all the writes to the same location already propagated to its thread (see Sarkar et al. [2011, p. 6, item 3 of § Accept write request]), in particular, w. Thus, we have $(w, w') \in$ co. Now, when w' reaches coherence point, all of its coherence predecessors must have reached theirs (see Sarkar et al. [2012, p. 4, item 2 of § Write reaches its coherence point]), in particular, w. Thus, w should reach its coherence point before w', which contradicts (i).

Third case: Contradict the premise (CPW: PROPAGATION)—that is, take a write w' in cpw s.t. $(w, w') \in$ prop. This contradicts the fact that writes cannot reach coherence point in an order that contradicts propagation (see Sarkar et al. [2012, p. 4, item 3 of § Write reaches coherence point]).

Satisfy Read. In the PLDI machine, a satisfy read transition does not modify the state (i.e., $s_1 = s_2$). In the intermediate state, we augment sr with the read that was satisfied. Now, we check that we do not contradict the premises SATISFY READ.

First case: Contradict the (SR: WRITE IS EITHER LOCAL OR COMMITTED) premise—that is, suppose that w is neither local nor committed. Then, we contradict the fact that a read can read either from a local po-loc-previous write (see Sarkar et al. [2011, p. 8, item 1 of § Satisfy memory read by forwarding an in-flight write directly to reading instruction]), or from a write from the storage subsytem—which therefore must have been committed (see Sarkar et al. [2011, p. 8, § Satisfy memory read from storage subsystem]).

Second case: Contradict the (SR: PPO/$\mathsf{ii}_0 \cap \mathsf{RR}$) premise—that is, take a satisfied read r' s.t. $(r, r') \in \mathsf{ppo} \cup \mathsf{fences}$. Then, we contradict the fact that read satisfaction follows the preserved program order and the fences (see Sarkar et al. [2011, p. 8, all items of both § Satisfy memory read by forwarding an in-flight write directly to reading instruction and § Satisfy memory read from storage subsystem]).

Third case: Contradict the (SR: OBSERVATION) premise—that is, take a write w' in co after w s.t. $(w', r) \in \mathsf{prop}; \mathsf{hb}^*$. Since w and w' are related by co, they have the same location. The PLDI transition guarantees that (i) w is the most recent write to addr(r) propagated to the thread of r (see Sarkar et al. [2011, p. 6, § Send a read response to a thread]). Moreover, $(w', r) \in \mathsf{prop}; \mathsf{hb}^*$ ensures that (ii) w' has been propagated to the thread of r, or there exists a write e such that $(w', e) \in \mathsf{co}$ and e is propagated to the thread of r. Therefore, we have $(w', w) \in \mathsf{co}$ (by Sarkar et al. [2011, p. 6, item 2 of § Propagate write to another thread]).

Therefore, by $(w, w') \in \mathsf{co}$, we know that w reaches its coherence point before w'. Yet, w' is a co-predecessor of w, which contradicts (see Sarkar et al. [2012, p. 4, item 2 of § Write reaches coherence point]).

Proof of (ii): We take $(w', r) \in \mathsf{prop}; \mathsf{hb}^*$ as done earlier. This gives us a write w'' s.t. $(w', w'') \in \mathsf{prop}$ and $(w'', r) \in \mathsf{hb}^*$. Note that $(w', w'') \in \mathsf{prop}$ requires the presence of a barrier between w' and w''.

We first remind the reader of a notion from Sarkar et al. [2011], the *group A of a barrier*: the group A of a barrier is the set of all the writes that have been propagated to the thread holding the barrier when the barrier is sent to the system (see Sarkar et al. [2011, p. 5, § Barriers (sync and lwsync) and cumulativity by fiat]). When a barrier is sent to a thread, all writes in its group A must have been propagated to that thread (see Sarkar et al. [2011, p. 6 item 2 of § Propagate barrier to another thread]). Thus, if we show that (a) the barrier between w' and w'' is propagated to the thread of r and (b) w' is in the group A of this barrier, we have our result.

Let us now do a case disjunction over $(w', w'') \in \mathsf{prop}$. When $(w', w'') \in \mathsf{prop\text{-}base}$, we have a barrier b such that (i) $(w', b) \in \mathsf{fences} \cup (\mathsf{rfe}; \mathsf{fences})$ and (ii) $(b, r) \in \mathsf{hb}^*$. Note (i) immediately entails that w' is in the group A of b. For (ii), we reason by induction over $(b, r) \in \mathsf{hb}^*$, the base case being immediate. In the inductive case, we have a write e such that b is propagated to the thread of e before e and e is propagated to the thread of r before r. Thus (by Sarkar et al. [2011, p. 6, item 3 of § Propagate write to another thread]), b is propagated to r.

When $(w', w'') \in \mathsf{com}^*; \mathsf{prop\text{-}base}^*; \mathsf{ffence}; \mathsf{hb}^*$, we have a barrier b (which is a full fence) such that $(w', b) \in \mathsf{com}^*; \mathsf{prop\text{-}base}^*$ and $(b, r) \in \mathsf{hb}^*$. We proceed by reasoning over com^*.

If there is no com step, then we have $(w', b) \in \mathsf{prop\text{-}base}^*$; thus, w' is propagated to the thread of b before b by the earlier prop-base case. Note that this entails that w' is in the group A of b. Since b is a full fence, b propagates to all threads (see Sarkar et al. [2011, premise of § Acknowledge sync barrier]), in particular to the thread of r. Thus (from Sarkar et al. [2011, item 2 of § Propagate barrier to another thread]), it follows that w' is propagated to r.

In the com^+ case (i.e., $(w', b) \in \mathsf{com}^*; \mathsf{prop\text{-}base}^*$), we remark that $\mathsf{com}^+ = \mathsf{com} \cup \mathsf{co}; \mathsf{rf} \cup \mathsf{fr}; \mathsf{rf}$. Thus, since w' is a write, only the cases rf, co and co; rf apply. In the rf case, we have $(w', b) \in \mathsf{prop\text{-}base}^*$, which leads us back to the base case (no com step) shown earlier. In the co and co; rf cases, we have $(w', b) \in \mathsf{co}; \mathsf{prop\text{-}base}^*$—that is, there exists a write e such that $(w', e) \in \mathsf{co}$ and $(e, b) \in \mathsf{prop\text{-}base}^*$ (i.e., our result).

Commit Read. In the PLDI machine, a commit read transition does not modify the state. In the intermediate state, we augment cr with the read that was committed. We now check that we do not contradict the premises of our COMMIT READ rule.

First case: Contradict the premise (CR: READ IS SATISFIED)—that is, suppose that the read that we want to commit is not in sr; this means that this read has not been satisfied. This is impossible since a read must be satisfied before it is committed (see Sarkar et al. [2011, p. 7, item 1 of § Commit in-flight instruction]).

Second case: Contradict the (CR: SC PER LOCATION/ coWR, coRW{1,2}, coRR) premise. This is impossible since the PLDI machine prevents coWR, coRW1, and coRW2) (see Sarkar et al. [2011, p. 3, § Coherence]).

Third case: Contradict the premise (CR: PPO/$\mathsf{cc}_0 \cap \mathsf{RW}$)—that is, take a committed write w' s.t. $(r, w') \in \mathsf{ppo} \cup \mathsf{fences}$. Since r must have been committed before w' (by Sarkar et al. [2011, p. 7, items 2, 3, 4, 5, 7 of § Commit in-flight instruction]), we get a contradiction.

Fourth case: Contradict the premise (CR: PPO/$(\mathsf{ci}_0 \cup \mathsf{cc}_0) \cap \mathsf{RR}$)—that is, take a satisfied read r' s.t. $(r, r') \in \mathsf{ppo} \cup \mathsf{fences}$. Since r must have been committed before r' was satisfied (by Sarkar et al. [2011, p. 8, item 3 of § Satisfy memory read from storage subsystem and item 3 of § Satisfy memory read by forwarding an in-flight write directly to reading instruction]), we get a contradiction.

8. TESTING AND SIMULATION

As usual in this line of work, we developed our model in tandem with extensive experiments on hardware. We report here on our experimental results on Power and ARM hardware. Additionally, we experimentally compared our model to the ones of Sarkar et al. [2011] and Mador-Haim et al. [2012]. Moreover, we developed a new simulation tool called herd[7] and adapted the cbmc tool [Alglave et al. 2013b] to our new models.

8.1. Hardware Testing

We performed our testing on several platforms using the diy tool suite [Alglave et al. 2010, 2011b, 2012]. This tool generates litmus tests—that is, very small programs in x86, Power, or ARM assembly code, with specified initial and final states. It then runs these tests on hardware and collects the memory and register states that it observed during the runs. Most of the time, litmus tests violate SC: if one can observe their final state on a given machine, then this machine exhibits features beyond SC.

We generated 8117 tests for Power and 9761 tests for ARM, illustrating various features of the hardware, such as lb, mp, sb, and their variations with dependencies and barriers, such as lb+addrs, mp+lwsync+addr, sb+syncs.

We tested the model described in Figures 5, 18, and 25 on Power and ARM machines, to check experimentally whether this model was suitable for these two architectures. In the following, we write "Power model" for this model instantiated for Power, and "Power-ARM model" for this model instantiated for ARM. We summarise these experiments in Table V.

[7]We acknowledge that we reused some code written by colleagues, in particular Susmit Sarkar, in an earlier version of the tool.

Table V. Summary of Our Experiments on Power and ARM h/w (w.r.t. Our Power-ARM Model)

	Power	ARM
# tests	8,117	9,761
invalid	0	1,500
unseen	1,182	1,820

For each architecture, the row "unseen" gives the number of tests that our model allows but that the hardware does not exhibit. This can be the case because our model is too coarse (i.e., fails to reflect the architectural intent in forbidding some behaviours), or because the behaviour is intended to be allowed by the architect but is not implemented yet.

The row "invalid" gives the number of tests that our model forbids but that the hardware does exhibit. This can be because our model is too strict and forbids behaviours that are actually implemented, or because the behaviour is a hardware bug.

8.1.1. Power. We tested three generations of machines: Power G5, 6, and 7. The complete logs of our experiments can be found at `http://diy.inria.fr/cats/model-power`.

Our Power model is not invalidated by Power hardware (there is no test in the "invalid" row on Power in Table V). In particular, it allows mp+lwsync+addr-po-detour, which Sarkar et al. [2011] wrongly forbids, as this behaviour is observed on hardware (see `http://diy.inria.fr/cats/pldi-power/#lessvs`).

Our Power model allows some behaviours (see the "unseen" row on Power), such as lb, that are not observed on Power hardware. This is to be expected as the lb pattern is not yet implemented on Power hardware, despite being clearly architecturally allowed [Sarkar et al. 2011].

8.1.2. ARM. We tested several system configurations: NVDIA Tegra 2 and 3; Qualcomm APQ8060 and APQ8064; Apple A5X and A6X; and Samsung Exynos 4412, 5250, and 5410. The complete logs of our experiments can be found at `http://diy.inria.fr/cats/model-arm`. This section about ARM testing is structured as follows:

—we first explain how our Power-ARM model is invalidated by ARM hardware (see the "invalid" row on ARM) by 1,500 tests (see the Our Power-ARM Model Is Invalidated by ARM Hardware section). We detail and document the discussion below at `http://diy.inria.fr/cats/arm-anomalies`.
—we then propose a model for ARM (see the section "Our Proposed ARM Model").
—we then explain how we tested this model on ARM machines and the anomalies that we have found whilst testing (see the section "Testing Our Model").

Our Power-ARM Model Is Invalidated by ARM Hardware. Amongst the tests we have run on ARM hardware, some unveiled a load-load hazard bug in the coherence mechanism of all machines. This bug is a violation of the coRR pattern shown in Section 4 and was later acknowledged as such by ARM, in the context of Cortex-A9 cores (in the note [ARM Ltd. 2011]).[8]

Amongst the machines that we have tested, this note applies directly to Tegra 2 and 3, A5X, and Exynos 4412. Additionally, Qualcomm's APQ8060 is supposed to have many architectural similarities with the ARM Cortex-A9; thus we believe that the note might apply to APQ8060 as well. Moreover, we have observed load-load hazards anomalies on Cortex-A15 based systems (Exynos 5250 and 5410), on the Cortex-A15

[8]We note that violating coRR invalidates the implementation of C++ modification order mo (e.g., the implementation of `memory_order_relaxed`), which explicitly requires the five coherence patterns of Figure 6 to be forbidden [Batty et al. 2011, p. 6, col. 1].

Table VI. Some Counts of Invalid Observations on ARM Machines

	model	machines
coRR	forbidden	allowed, 10M/95G
coRSDWI	forbidden	allowed, 409k/18G
mp+dmb+fri-rfi-ctrlisb	forbidden	allowed, 153k/178G
lb+data+fri-rfi-ctrl	forbidden	allowed, 19k/11G
moredetour0052	forbidden	allowed, 9/17G
mp+dmb+pos-ctrlisb+bis	forbidden	allowed, 81/32G

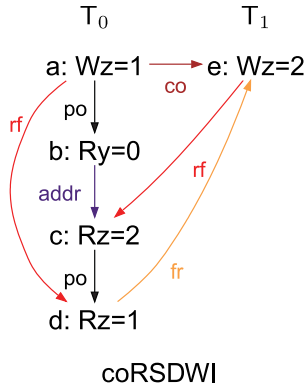


Fig. 31. An observed behaviour that features a coRR violation.

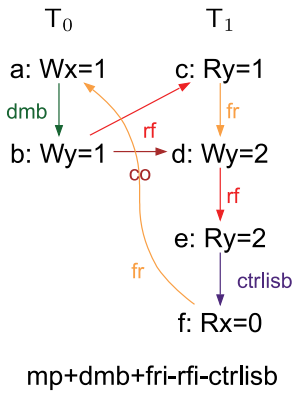


Fig. 32. A feature of some ARM machines.

compatible Apple "Swift" (A6X), and on the Krait-based APQ8064, although much less often than on Cortex-A9 based systems. Note that we observed the violation of coRR itself quite frequently, as illustrated by the first row of Table VI. The second row of Table VI refers to a more sophisticated test, coRSDWI (Figure 31), the executions of which reveal violations of the coRR pattern on location z. Both tests considered, we observed the load-load hazard bug on all the ARM machines that we have tested.

Others, such as the mp+dmb+fri-rfi-ctrlisb behaviour of Figure 32, were claimed to be desirable behaviours by the designers we talked to. This behaviour is a variant of the message passing example, with some more accesses to the flag variable y before the read of the message variable x. We observed this behaviour quite frequently (see the third row of Table VI) albeit on one machine (of type APQ8060) only.

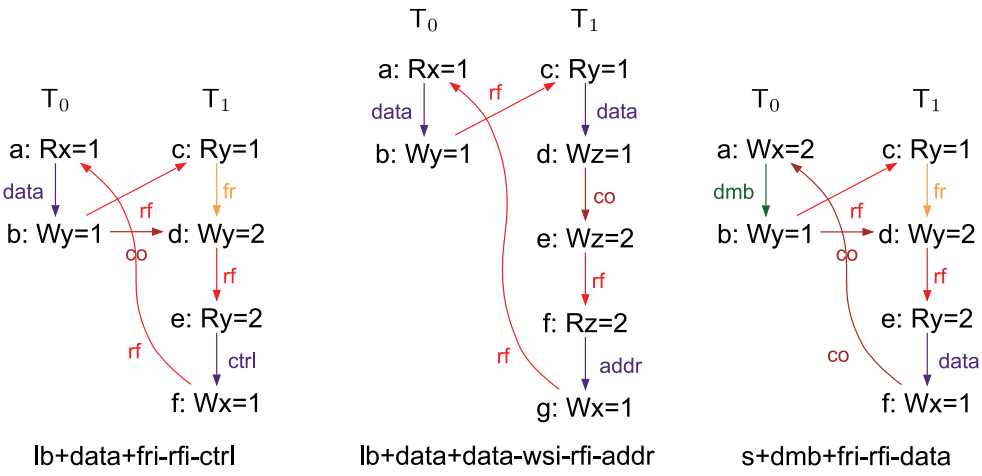


Fig. 33. Putative features of some ARM machines.

Additionally, we observed similar behaviours on APQ8064 (see also http://diy.inria.fr/cats/model-qualcomm/compare.html#apq8064-invalid). We give three examples of these behaviours in Figure 33. The first two are variants of the load buffering example of Figure 7. The last, s+dmb+fri-rfi-data, is a variant of the s pattern shown later in Figure 39. We can only assume that these are as desirable as the behaviour in Figure 32.

For reasons explained in the next paragraph, we gather all such behaviours under the name "early commit behaviours" (see reason (ii) in the section "Our Proposed ARM Model").

Our Proposed ARM Model. Our Power-ARM model rejects the mp+dmb+fri-rfi-ctrlisb behaviour via the OBSERVATION axiom, as the event c is ppo-before the event f. More precisely, from our description of preserved program order (see Figure 25), the order from c to f derives from three reasons: (i) reads are satisfied before they are committed (i(r) precedes c(r)); (ii) instructions that touch the same location commit in order (po-loc is in cc_0); and (iii-a) instructions in a branch that are po-after a control fence (here isb) do not start before the isb executes, and isb does not execute before the branch is settled, which in turn requires the read (e in the diagram) that controls the branch to be committed (ctrl+cfence is in ci_0).

Our Power-ARM model rejects the s+dmb+fri-rfi-data behaviour via the PROPAGATION axiom for the same reason: the event c is ppo-before the event f.

Similarly, our Power-ARM model rejects the lb+data+fri-rfi-ctrl behaviour via the NO THIN AIR axiom, as the event c is ppo-before the event f. Here, still from our description of preserved program order (see Figure 25), the order from c to f derives from three reasons: (i) reads are satisfied before they commit (i(r) precedes c(r)); (ii) instructions that touch the same location commit in order (po-loc is in cc_0); and (iii-b) instructions (in particular store instructions) in a branch do not commit before the branch is settled, which in turn requires the read (e in the diagram) that controls the branch to be committed (ctrl is in cc_0).

Finally, our Power-ARM model rejects lb+data+data-wsi-rfi-addr via the NO THIN AIR axiom, because the event c is ppo-before the event g. Again, from our description of preserved program order (see Figure 25), the order from c to f derives from three reasons: (i) reads are satisfied before they commit (i(r) precedes c(r)); (ii) instructions that touch the same location commit in order (po-loc is in cc_0); and (iii-c) instructions

Table VII. Summary of ARM Models

	Power-ARM	ARM	ARM llh
skeleton	Fig. 5	Fig. 5	Fig. 5 s.t. SC PER LOCATION becomes acyclic(po-loc-llh ∪ com), with po-loc-llh ≜ po-loc \ RR
propagation	Fig. 18	Fig. 18	Fig. 18
ppo	Fig. 25	Fig. 25 s.t. cc_0 becomes dp ∪ ctrl ∪ (addr; po)	Fig. 25 s.t. cc_0 becomes dp ∪ ctrl ∪ (addr; po)

(in particular store instructions) do not commit before a read they depend on (e.g., the read f) is satisfied (addr is in ic_0 because it is in ii_0 and ii_0 is included in ic_0).

The reasons (i), (iii-a), (iii-b), and (iii-c) seem uncontroversial. In particular for (iii-a), if ctrl+cfence is not included in ppo, then neither the compilation scheme from C++ nor the entry barriers of locks would work [Sarkar et al. 2012]. For (iii-b), if ctrl to a write event is not included in ppo, then some instances of the pattern lb+ppos (see Figure 7) such as lb+ctrls would be allowed. From the architecture standpoint, this could be explained by value speculation, an advanced feature that current commodity ARM processors do not implement. A similar argument applies for (iii-c), considering this time that lb+addrs should not be allowed by a hardware model. In any case, allowing such simple instances of the pattern lb+ppos would certainly contradict (low-level) programmer intuition.

As for (ii) however, one could argue that this could be explained by an "early commit" feature. For example, looking at mp+dmb+fri-rfi-ctrlisb in Figure 32 and lb+data+fri-rfi-ctrl in Figure 33, the read e (which is satisfied by forwarding the local write d) could commit without waiting for the satisfying write d to commit, nor for any other write to the same location that is po-before d. Moreover, the read e could commit without waiting for the commit of any read from the same location that is po-before its satisfying write. We believe that this might be plausible from a microarchitecture standpoint: the value read by e cannot be changed by any later observation of incoming writes performed by load instructions po-before the satisfying write d; thus, the value read by e is irrevocable.

In conclusion, to allow the behaviours of Figure 32, we need to weaken the definition of preserved program order of Figure 25. For the sake of simplicity, we chose to remove po-loc altogether from the cc_0 relation, which is the relation ordering the commit parts of events. This means that two accesses relative to the same location and in program order do not have to commit in this order.[9]

Thus, we propose the following model for ARM, which has so far not been invalidated on hardware (barring the load-load hazard behaviours, acknowledged as bugs by ARM [ARM Ltd. 2011] and the other anomalies presented in the section "Testing Our Model", which we take to be undesirable). We go back to the soundness of our ARM model in the section "Remarks on Our Proposed ARM Model".

The general skeleton of our ARM model should be the four axioms given in Figure 5, and the propagation order should be as given in Figure 18. For the preserved program order, we take it to be as the Power one given in Figure 25, except for cc_0, which now excludes po-loc entirely, to account for the early commit behaviours—that is, cc_0 should now be dp ∪ ctrl ∪ (addr; po). Table VII gives a summary of the various ARM models that we consider in this section.

[9]This is the most radical option; one could choose to remove only po-loc ∩ WR and po-loc ∩ RR , as that would be enough to explain the behaviours of Figure 32 and similar others. We detail our experiments with alternative formulations for cc_0 at `http://diy.inria.fr/cats/arm-anomalies/index.html#alternative`. Ultimately, we chose to adopt the weakest model because, as we explain in this section, it still exhibits hardware anomalies.

Table VIII. Classification of Anomalies Observed on ARM Hardware

	ALL	S	T	P	ST	SO	SP	OP	STO	SOP
Power-ARM	37,907	21,333	842	1,133	2,471	1,130	5,561	872	111	4,062
ARM llh	1,121	105	0	0	0	16	10	460	0	530

Testing Our Model. For the purpose of these experiments only, because our machines suffered from the load-load hazard bug, we removed the read-read pairs from the SC PER LOCATION check as well. This allowed us to have results that were not cluttered by this bug.

We call the resulting model "ARM llh" (ARM load-load hazard), that is, the model of Figures 5, 18, and 25—where we remove read-read pairs from po-loc in the SC PER LOCATION axiom to allow load-load hazards, and where cc_0 is dp $\cup$ ctrl $\cup$ (addr; po). Table VII gives a summary of this model, as well as Power-ARM and our proposed ARM model.

We then compared our original Power-ARM model (i.e., taking literally the definitions of Figures 5, 18, and 25) and the ARM llh model with hardware—we omit the ARM model (middle column of Table VII) because of the acknowledged hardware bugs [ARM Ltd. 2011]:

More precisely, we classify the executions of both models: for each model, we count the number of invalid executions (in the sense of Section 4.1). By "invalid," we mean that an execution is forbidden by the model yet observed on hardware. A given test can have several executions, which explains why the numbers in Table VIII are much higher than the number of tests.

Table VIII is organised by sets of axioms of our model (note that these sets are pairwise disjoint): "S" is for SC PER LOCATION, "T" for NO THIN AIR, "O" for OBSERVATION, and "P" is for PROPAGATION. For each set of axioms (column) and model (row), we write the number of executions forbidden by said axiom(s) of said model, yet have been observed on ARM hardware. We omit a column (namely, O, TO, TP, STP, TOP, STOP) if the counts are 0 for both models.

For example, an execution is counted in the column "S" of the row "Power-ARM" if it is forbidden by the SC PER LOCATION check of Figure 5 (and allowed by other checks). The column "S" of the row "ARM llh" is SC PER LOCATION minus the read-read pairs. An execution is counted in the column "OP" of the row "Power-ARM" if it is forbidden by both OBSERVATION and PROPAGATION as defined in Figure 5 (and allowed by other checks); for the row "ARM llh," one needs to take into account the modification to cc_0 mentioned earlier in the definition of the ppo.

Although our original Power-ARM model featured 1,500 tests that exhibited 37,907 invalid executions forbidden by the model yet observed on ARM machines, those numbers drop to 31 tests and 1,121 invalid executions for the ARM llh model (see row "ARM llh," column "ALL"; see also `http://diy.inria.fr/cats/relaxed-classify/index.html`).

An inspection of each of these anomalies revealed what we believe to be more bugs. We consider the violations of SC PER LOCATION to be particularly severe (see all rows mentioning "S"). By contrast, the load-load hazard behaviour could be argued to be desirable, or at least not harmful, and was indeed officially allowed by Sparc RMO [SPARC International Inc. 1994] and pre–Power 4 machines [Tendler et al. 2002], as we mentioned in Section 4.8.

Figure 34 shows a violation of SC PER LOCATION. Despite the apparent complexity of the picture, the violation is quite simple. The violation occurs on T_1, which loads the value 4 from the location y (event f), before writing the value 3 to same location y (event g). However, 4 is the final value of the location y, as the test harness has observed once the test has completed. As a consequence, the event e co-precedes the event f, and we

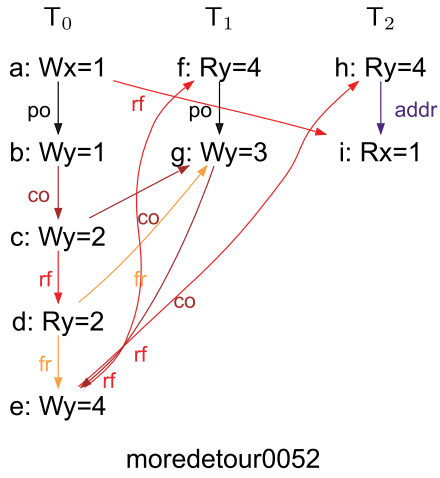


Fig. 34. A violation of SC PER LOCATION observed on ARM hardware.

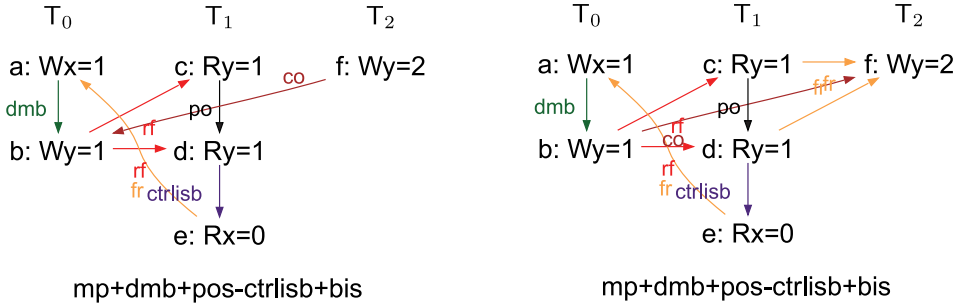


Fig. 35. Two violations of OBSERVATION observed on ARM hardware.

witness a cycle $g \xrightarrow{co} e \xrightarrow{rf} f \xrightarrow{po\text{-}loc} g$. That is, we witness a violation of the coRW2 pattern (see Section 4). Notice that this violation may hinder the C/C++ semantics of low-level atomics. Namely, accesses to atomics performed by using RMO are usually compiled as plain accesses when their size permit, but nevertheless have to be "coherent"—that is, must follow the SC PER LOCATION axiom.

Note that we observed this behaviour rather infrequently, as shown in the fifth row of Table VI. However, the behaviour is observed on two machines of type Tegra 3 and Exynos 4412.

In addition to the violations of SC PER LOCATION shown in Figure 34, we observed the two behaviours of Figure 35 (rather infrequently, as shown on the last row of Table VI, and on one machine only, of type Tegra 3), which violate OBSERVATION.

The test mp+dmb+pos-ctrlisb+bis includes the simpler test mp+dmb+ctrlisb plus one extra read (c on T_1) and one extra write (f on T_2) of the flag variable y. The depicted behaviours are violations of the mp+dmb+ctrlisb pattern, which must uncontroversially be forbidden. Indeed, the only way to allow mp+dmb+ctrlisb is to remove ctrl+cfence from the preserved program order ppo. We have argued earlier that this would, for example, break the compilation scheme from C++ to Power (see Sarkar et al. [2012]).

It is worth noting that we have observed other violations of OBSERVATION on Tegra 3, as one can see at `http://diy.inria.fr/cats/relaxed-classify/OP.html`. For example, we have observed mp+dmb+ctrlisb, mp+dmb+addr, and mp+dmb.st+addr, which should

be uncontroversially forbidden. We tend to classify such observations as bugs of the tested chip. However, since the tested chip exhibits the acknowledged read-after-read hazard bug, the blame can also be put on the impact of this acknowledged bug on our testing infrastructure. Yet this would mean that this impact on our testing infrastructure would show up on Tegra 3 only.

In any case, the interplay between having several consecutive accesses relative to the same location on one thread (e.g., $c, d,$ and e on T_1 in mp+dmb+fri-rfi-ctrlisb—see Figure 32), in particular, two reads (c and e), and the message passing pattern mp, seems to pose implementation difficulties (see the violations of OBSERVATION listed in Table VIII, in the columns containing "O," and the two examples in Figure 35).

Remarks on Our Proposed ARM Model. Given the state of affairs for ARM, we do not claim our model (see model "ARM" in Table V) to be definitive. In particular, we wonder whether the behaviour mp+dmb+fri-rfi-ctrlisb of Figure 32 can only be implemented on a machine with load-load hazards, which ARM acknowledged to be a flaw (see ARM Ltd. [2011]), as it involves two reads from the same address.

Nevertheless, our ARM contacts were fairly positive that they would like this behaviour to be allowed. Thus, we think a good ARM model should account for it. As to the similar early commit behaviours given in Figure 33, we can only assume that they should be allowed as well.

Hence, our ARM model allows such behaviours, by excluding po-loc from the commit order cc_0 (see Figure 25 and Table V). We have performed experiments to compare our ARM model and ARM hardware. To do so, we have excluded the load-load hazard related behaviours.[10]

We give the full comparison table at `http://diy.inria.fr/cats/proposed-arm/`. As one can see, we still have 31 behaviours that our model forbids yet are observed on hardware (on Tegra 2, Tegra 3, and Exynos 4412).

All of them seem to present anomalies, such as the behaviours that we show in Figures 34 and 35. We will consult with our ARM contacts for confirmation.

8.2. Experimental Comparisons of Models

Using the same 8,117 and 9,761 tests that we used to exercise Power and ARM machines, we have experimentally compared our model to the one of Sarkar et al. [2011] and the one of Mador-Haim et al. [2012].

Comparison with the Model of Sarkar et al. [2011]. Our experimental data can be found at `http://diy.inria.fr/cats/pldi-model`. Experimentally, our Power model allows all behaviours that are allowed by the one of Sarkar et al. [2011], which is in line with our proofs of Section 7.

We also observe experimentally that our Power model and the one of Sarkar et al. [2011] differ *only* on the behaviours that Sarkar et al. [2011] wrongly forbids (see `http://diy.inria.fr/cats/pldi-power/#lessvs`). We give one such example in Figure 36: the behaviour mp+lwsync+addr-po-detour is observed on hardware yet forbidden by the model of Sarkar et al. [2011].

We note that some work is ongoing to adapt the model of Sarkar et al. [2011] to allow these tests (see `http://diy.inria.fr/cats/op-model`). This new variant of the model of Sarkar et al. [2011] has so far not been invalidated by the hardware.

[10]More precisely, we have built the model that only allows load-load hazard behaviours. In herd parlance, this is a model that only has one check: reflexive(po-loc; fr; rf). We thus filtered the behaviours observed on hardware by including only the behaviours that are not allowed by this load-load hazard model (i.e., all but load-load hazard behaviours). We then compared these filtered hardware behaviours with the ones allowed by our ARM model.

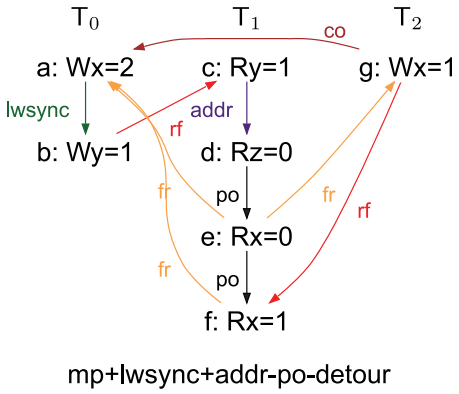


Fig. 36. A behaviour forbidden by the model of Sarkar et al. but observed on Power hardware.

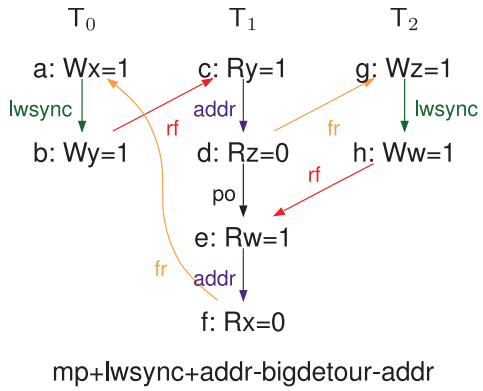


Fig. 37. A behaviour allowed by our model but forbidden by Mador-Haim et al. [2012].

Finally, the model of Sarkar et al. [2011] forbids the ARM "fri-rfi" behaviours such as the ones given in Figure 32. Some work is ongoing to adapt the model of Sarkar et al. [2011] to allow these tests.

Comparison with the Model of Mador-Haim et al. [2012]. Our experimental data can be found at `http://diy.inria.fr/cats/cav-model`. Our Power model and the one of Mador-Haim et al. [2012] are experimentally equivalent on our set of tests, except for a few tests of similar structure. Our model allows them, whereas the model of Mador-Haim et al. [2012] forbids them, and they are not observed on hardware. We give the simplest such test in Figure 37. The test is a refinement of the mp+lwsync+ppo pattern (see Figure 8). The difference of acceptance between the two models can be explained as follows: the model of Mador-Haim et al. [2012] preserves the program order from T_1 initial read c to T_1 final read f, whereas our model does not. More precisely, the issue reduces to reads d and e (on T_1) being ordered or not. And, indeed, the propagation model for writes of Mador-Haim et al. [2012] enforces the order, whereas our definition of ppo does not.

If such a test is intentionally forbidden by the architect, it seems to suggest that one could make the preserved program order of Power (see Figure 25) stronger. Indeed, one could take into account the effect of barriers (such as the one between the two writes g and h on T_2 in Figure 37) within the preserved program order.

Yet, we think that one should tend towards more simplicity in the definition of the preserved program order. It feels slightly at odds with our intuition that the preserved program order should take into account dynamic notions such as the propagation order of the writes g and h. By dynamic notions, we here mean notions which require execution relations, such as rf or prop, in their definition.

As a related side note, although we did include the dynamic relations rdw and detour into the definition of the preserved program order in Figure 25, we would rather prescribe not to include them. This would lead to a weaker notion of preserved program order, but more stand-alone. By this, we mean that the preserved program order would just contain per-thread information (e.g., the presence of a control fence, or a dependency between two accesses), as opposed to external communications such as rfe.

We experimented with a weaker, more static, version of the preserved program order for Power and ARM, where we excluded rdw from ii_0 and detour from ci_0 (see Figure 25). We give the full experiment report at `http://diy.inria.fr/cats/nodetour-model/`. On our set of tests, this leads to only 24 supplementary behaviours allowed on Power and 8 on ARM. We believe that this suggests that it might not be worth complicating the ppo for the sake of only a few behaviours being forbidden. Yet, it remains to be seen whether these patterns are so common that it is important to determine their precise status w.r.t. a given model.

8.3. Model-Level Simulation

Simulation was done using our new herd tool: given a model specified in the terms of Section 4 and a litmus test, herd computes all executions allowed by the model. We distribute our tool, its sources, and documentation at `http://diy.inria.fr/herd`.

Our tool herd understands models specified in the style of Section 4—that is, defined in terms of relations over events, and irreflexivity or acyclicity of these relations. For example, Figure 38 gives the herd model corresponding to our Power model (see Sections 4 and 6). We emphasise the concision of Figure 38, which contains the *entirety* of our Power model.

Language Description. We build definitions with `let`, `let rec`, and `let rec ... and ...` operators. We build unions, intersections, and sequences of relations, with "`|`", "`&`" and "`;`" respectively; transitive closure with "`+`", and transitive and reflexive closure with "`*`". The empty relation is "`0`".

We have some built-in relations, such as po-loc, rf, fr, co, addr, data, and operators to specify whether the source and target events are reads or writes. For example, `RR(r)` gives the relation `r` restricted to both the source and target being reads.

Finally, model checks are implemented by the `acyclic` and `irreflexive` instructions. These are instructions with the following semantics: if the property does not hold, model simulation stops; otherwise, it continues. See herd's documentation for a more thorough description of syntax and semantics.

To some extent, the language that herd takes as input shares some similarities with the much broader Lem project [Owens et al. 2011]. However, we merely intend to have a concise way of defining a variety of memory models, whereas Lem aims at (citing the website, `http://www.cs.kent.ac.uk/people/staff/sao/lem/`) "large scale semantic definitions. It is also intended as an intermediate language for generating definitions from domain-specific tools, and for porting definitions between interactive theorem proving systems."

The alloy tool [Jackson 2002] (see also `http://alloy.mit.edu/alloy`) is closer to herd than Lem. Both alloy and herd allow a concise relational definition of a given system. But whereas alloy is very general, herd is only targeted at memory models definitions.

Thus, one could see herd as a potential front end to alloy. For example, herd provides some built-in objects (e.g., program order, read-from) that spare the user the

```
(* sc per location *) acyclic po-loc|rf|fr|co

(* ppo *)
let dp = addr|data
let rdw = po-loc & (fre;rfe)
let detour = po-loc & (coe;rfe)

let ii0 = dp|rdw|rfi
let ic0 = 0
let ci0 = (ctrl+isync)|detour
let cc0 = dp|po-loc|ctrl|(addr;po)

let rec ii = ii0|ci|(ic;ci)|(ii;ii)
and ic = ic0|ii|cc|(ic;cc)|(ii;ic)
and ci = ci0|(ci;ii)|(cc;ci)
and cc = cc0|ci|(ci;ic)|(cc;cc)
let ppo = RR(ii)|RW(ic)

(* fences *)
let fence = RM(lwsync)|WW(lwsync)|sync

(* no thin air *)
let hb = ppo|fence|rfe
acyclic hb

(* prop *)
let prop-base = (fence|(rfe;fence));hb*
let prop = WW(prop-base)|(com*;prop-base*;sync;hb*)

(* observation *) irreflexive fre;prop;hb*
(* propagation *) acyclic co|prop
```

Fig. 38. herd definition of our Power model.

effort of defining these objects; alloy would need the user to make these definitions explicit.

More precisely, to specify a memory model in alloy, one would need to explicitly define an object "memory event," such as a record with an identifier, a direction (i.e., write or read), a location, and a value, much like we do in our Coq development (see `http://www0.cs.ucl.ac.uk/staff/j.alglave/cats/`).

One would also need to hand-craft relations over events (e.g., the program order po), as well as the well-formedness conditions of these relations (e.g., the program order is total order per thread), using first-order logic. Our tool herd provides all of these basic bricks (events, relations, and their well-formedness conditions) to the user.

Finally, alloy uses a SAT solver as a back end, whereas herd uses a custom solver optimised for the limited constraints that herd supports (namely, acyclicity and irreflexivity of relations).

Efficiency of Simulation. Our axiomatic description underpins herd, which allows for a greater efficiency in the simulation. By contrast, simulation tools based on operational models (e.g., ppcmem Sarkar et al. [2011], or the tool of Boudol et al. [2012][11]) are not

[11]All results relative to the tool of Boudol et al. [2012] are courtesy of Arthur Guillon, who exercised the simulator of Boudol et al. [2012] on a subset of the tests that we used for exercising the other tools.

Table IX. Comparison of Simulation Tools (on Power)

tool	model	style	# of tests	(user) time in *s*
ppcmem	Sarkar et al. [2011]	operational	4,704	14,922,996
herd	Mador-Haim et al. [2012]	multi-event axiomatic	8,117	2,846
—	Boudol et al. [2012]	operational	396	53,100
herd	this model	single-event axiomatic	8,117	321

able to process all tests within the memory bound of 40GB for Sarkar et al. [2011] and 6GB for Boudol et al. [2012]: ppcmem processes 4,704 tests out of 8,117; the tool of Boudol et al. [2012] processes 396 tests out of 518.

Tools based on multi-event axiomatic models (which includes our reimplementation of Mador-Haim et al. [2012] inside herd) are able to process all 8,117 tests but require more than eight times the time that our single-event axiomatic model needs.

Table IX gives a summary of the number of tests that each tool can process, as well as the time needed to do so.

As we have implemented the model of Mador-Haim et al. [2012][12] and the present model inside herd using the same techniques, we claim that the important gain in runtime efficiency originates from reducing the number of events. On a reduced number of events, classical graph algorithms such as acyclicity test and, more significantly, transitive closure and other fixed point calculations run much faster.

We note that simulation based on axiomatic models outperforms simulation based on operational models. This is mostly due to a state explosion issue, which is aggravated by the fact that Power and ARM are very relaxed architectures. Thus, in any given state of the operational machine, there are numerous operational transitions enabled.

We note that ppcmem is not coded as efficiently as it could be. Better implementations are called for, but the distance to herd is considerable: herd is about 45,000 times faster than ppcmem, and ppcmem fails to process about half of the tests.

We remark that our single-event axiomatic model also needs several subevents to describe a given instruction (e.g., see our definition of the preserved program order for Power in Figure 25). Yet, the opposition between multi-event and single-event axiomatic models lies in the number of events needed to describe the propagation of writes to the system. In multi-event models, there is roughly one propagation event per thread, mimicking the transitions of an operational machine. In single-event models, there is only one event to describe the propagation to several different threads; the complexity of the propagation mechanism is captured through our use of the relations (e.g., rf, co, fr, and prop).

We note that single-event axiomatic simulators also suffer from combinatorial explosion. The initial phase computes executions (in the sense of Section 4.1) and thus enumerates all possible rf and co relations. However, as clearly shown in Table IX, the situation is less severe, and we can still process litmus tests of up to four or five threads.

8.4. Verification of C Programs

Although assembly-level litmus tests enable detailed study of correctness of the model, the suitability of our model for the verification of high-level programs remains to be proven. To this effect, we experimented with a modified version of cbmc [Clarke et al. 2004], which is a bounded model checker for C programs. Recent work [Alglave et al. 2013b] has implemented the framework of Alglave et al. [2010, 2012] in cbmc, and

[12]The original implementations tested in Mador-Haim et al. [2012] were, despite using the same model, much less efficient.

Table X. Comparison of Operational Versus Axiomatic Model Implementation

tool	model	# of tests	time in s
goto-instrument+cbmc (SC)	Alglave et al. [2012]	555	2,511.6
cbmc (Power)	Alglave et al. [2012]	555	14.3

Table XI. Comparison of Verification Tools on Litmus Tests

tool	model	# of tests	time in s
cbmc	Mador-Haim et al. [2012]	4,450	1,944
cbmc	present one	4,450	1,041

Table XII. Comparison of Verification Tools on Full-Fledged Examples

tool	model	PgSQL	RCU	Apache
cbmc	Mador-Haim et al. [2012]	1.6	0.5	2.0
cbmc	present one	1.6	0.5	2.0

observed speedups of an order of magnitude w.r.t. other verification tools. cbmc thus features several models, ranging from SC to Power.

In addition, Alglave et al. [2013a] proposes an instrumentation technique, which transforms a concurrent program so that it can be processed by an SC verification tool, such as cbmc in SC mode. This relies on an operational model equivalent to the one of Alglave et al. [2012]; we refer to it in Table X under the name "goto-instrument+tool." The advantage of supporting existing tools in SC mode comes at the price of a considerably slower verification time when compared to the implementation of the equivalent axiomatic model within the verification tool, as Table X shows.

We adapted the encoding of Alglave et al. [2013b] to our present framework and recorded the time needed to verify the reachability of the final state of more than 4,000 litmus tests (translated to C). As a comparison point, we also implemented the model of Mador-Haim et al. [2012] in cbmc and compared the verification times, given in Table XI. We observe some speedup with the present model over the implementation of the model of Mador-Haim et al. [2012].

We also compared the same tools, but on more fully fledged examples, described in detail in Alglave et al. [2013a, 2013b]: PgSQL is an excerpt of the PostgreSQL database server software (see `http://archives.postgresql.org/pgsql-hackers/2011-08/msg00330.php`); RCU is the Read-Copy-Update mechanism of the Linux kernel [McKenney and Walpole 2007], and Apache is a queue mechanism extracted from the Apache HTTP server software. In each of the examples, we added correctness properties, described in Alglave et al. [2013b], as assertions to the original source code. We observed that the verification times of these particular examples are not affected by the choice of either of the two axiomatic models, as shown in Table XII.

9. A PRAGMATIC PERSPECTIVE ON OUR MODELS

To conclude our article, we put our modelling framework into perspective, in light of actual software. Quite pragmatically, we wonder whether it is worth going through the effort of defining, on the one hand, then studying or implementing, on the other hand, complex models such as the Power and ARM models that we present in Section 6. Are there fragments of these models that are simpler to understand and embrace pretty much all the patterns that are used in actual software?

For example, there is a folklore notion that iriw (see Figure 20) is very rarely used in practice. If that is the case, do we need models that can explain iriw?

Conversely, one flaw of the model of Alglave et al. [2012] (and also of the model of Boudol et al. [2012]) is that it forbids the pattern r+lwsync+sync (see Figure 16), against the architect's intent [Sarkar et al. 2011]. While designing the model that we present in the current article, we found that accounting for this pattern increased the complexity of the model. If this pattern is never used in practice, it might not be worth inventing a model that accounts for it, if it makes the model much more complex.

Thus, we ask the following questions: what are the patterns used in modern software? What are their frequencies?

Additionally, we would like to understand whether there are programming patterns used in current software that are not accounted for by our model. Are there programming patterns that are not represented by one of the axioms of our model—that is, SC PER LOCATION, NO THIN AIR, OBSERVATION, or PROPAGATION, as given in Figure 5?

Conversely, can we understand all of the patterns used in current software through the prism of, for example, our OBSERVATION axiom, or is there an actual need for the PROPAGATION axiom too? Finally, we would like to understand to what extent do hardware anomalies, such as the load-load hazard behaviour that we observed on ARM chips (see Section 8) impair the behaviour of actual software.

To answer these questions, we resorted to the largest code base available to us: an entire Linux distribution.

What We Analysed. We picked the current stable release of the Debian Linux distribution (version 7.1, `http://www.debian.org/releases/stable/`), which contains more than 17,000 software packages (including the Linux kernel itself, server software such as Apache or PostgreSQL, but also user-level software, such as Gimp or Vim).

David A. Wheeler's SLOCCount tool (`http://www.dwheeler.com/sloccount/`) reports more than 400 million lines of source code in this distribution. C and C++ are still the predominant languages: we found more than 200 million lines of C and more than 129 million lines of C++.

To search for patterns, we first gathered the packages that possibly make use of concurrency. That is, we selected the packages that make use of either POSIX threads or Linux kernel threads anywhere in their C code. This gave us 1,590 source packages to analyse; this represents 9.3% of the full set of source packages.

The C language [ISO 2011] does not have an explicit notion of shared memory. Therefore, to estimate the number of shared memory interactions, we looked for variables with static storage duration (in the C11 standard sense [ISO 2011, §6.2.4]) that were not marked thread local. We found a total of 2 733 750 such variables. In addition to these, our analysis needs to consider local variables shared through global pointers and objects allocated on the heap to obtain an overapproximation of the set of objects (in the C11 standard sense [ISO 2011, §3.15]) that may be shared between threads.

A Word on C++. The C++ memory model has recently received considerable academic attention (e.g., see Batty et al. [2011, 2013] and Sarkar et al. [2012]). Yet, to date, even a plain text search in all source files for uses of the corresponding `stdatomic.h` and `atomic` header files only reveals occurrences in the source code of compilers, but not in any of the other source packages.

Thus, practical assessment of our subset of the C++ memory model is necessarily left for future work. At the same time, this result reinforces our impression that we need to study hardware models to inform current concurrent programming.

9.1. Static Pattern Search

To look for patterns in Debian 7.1, we implemented a static analysis in a new tool called mole. This means that we are looking for an overapproximation—see later for details—of the patterns used in the program. Building on the tool chain described in

Alglave et al. [2013a], we use the front-end goto-cc and a variant of the goto-instrument tool of Alglave et al. [2013a], with the new option –static-cycles. We distribute our tool mole, along with a documentation, at `http://diy.inria.fr/mole`.

9.1.1. Preamble on the goto- Tools.* goto-cc and goto-instrument are part of the tool chain of cbmc [Clarke et al. 2004], which is widely recognised for its maturity.[13] goto-cc may act as compiler substitute, as it accepts the same set of command line options as several C compilers, such as gcc. Instead of executables, however, goto-cc compiles C programs to an intermediate representation shared by the tool chain around cbmc: *goto-programs*. These goto-programs can be transformed and inspected using goto-instrument. For instance, goto-instrument can be applied to insert assertions of generic invariants such as valid pointer dereferencing or data race checks, or dump goto-programs as C code—and was used in Alglave et al. [2013a] to insert buffers for simulating weak memory models. Consequently, we implemented the search described below in goto-instrument, adding the new option –static-cycles.

9.1.2. Cycles. Note that a pattern like all of the ones that we have presented in this article corresponds to a cycle of the relations of our model. This is simply because our model is defined in terms of irreflexivity and acyclicity checks. Thus, looking for patterns corresponds here to looking for cycles of relations.

Critical Cycles. Previous works [Shasha and Snir 1988; Alglave and Maranget 2011; Bouajjani et al. 2011, 2013] show that a certain kind of cycles, which we call *critical cycles* (following Shasha and Snir [1988]), characterises many weak behaviours. Intuitively, a critical cycle violates SC in a minimal way.

We recall here the definition of a critical cycle (see Shasha and Snir [1988] for more details). Two events x and y are *competing*, written $(x, y) \in \mathsf{cmp}$, if they are from distinct processors, to the same location, and at least one of them is a write (e.g., in iriw, the write a to x on T_0 and the read b from x on T_2). A cycle $\sigma \subseteq (\mathsf{cmp} \cup \mathsf{po})^+$ is critical when it satisfies the following two properties:

—(i) per thread, there are at most two memory accesses involved in the cycle on this thread and these accesses have distinct locations, and
—(ii) for a given memory location ℓ, there are at most three accesses relative to ℓ, and these accesses are from distinct threads ($(w, w') \in \mathsf{cmp}$, $(w, r) \in \mathsf{cmp}$, $(r, w') \in \mathsf{cmp}$, or $\{(r, w), (w, r')\} \subseteq \mathsf{cmp}$).

All of the executions that we give in Section 4 show critical cycles, except for the SC PER LOCATION ones (see Figure 6). Indeed, a critical cycle has to involve more than one memory location by definition.

Static Critical Cycles. More precisely, our tool mole looks for cycles that

—Alternate program order po and competing accesses cmp,
—Traverse a thread only once (see (i) earlier), and
—Involve at most three accesses per memory location (see (ii) earlier).

Observe that the preceding definition is not limited to the well-known patterns that we presented in Section 4. Consider the two executions in Figure 39, both of which match the definition of a critical cycle given earlier.

On the left-hand side, the thread T_1 writes 1 to location x (see event d); the thread T_2 reads this value from x (i.e., $(d, e) \in \mathsf{rf}$), before the thread T_0 writes 2 to x (i.e., $(e, a) \in \mathsf{fr}$). By definition of fr, this means that the write d of value 1 into x by T_1 is co-before the

[13] `http://www.research.ibm.com/haifa/conferences/hvc2011/award.shtml`.

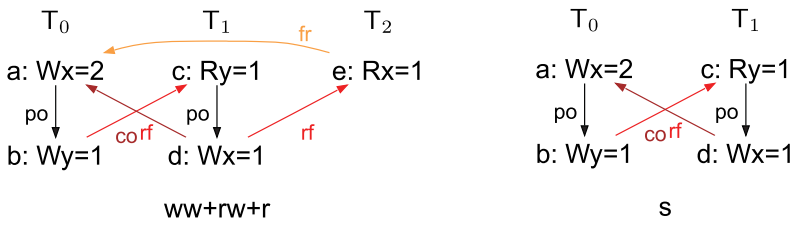


Fig. 39. The pattern s (on the right) and an extended version of it (on the left).

write a of value 2 into x by T_0. This is reflected by the execution on the right-hand side, where we simply omitted the reading thread T_2.

Thus, to obtain our well-known patterns, such as the ones in Section 4 and the s pattern on the right-hand side of Figure 39, we implement the following reduction rules, which we apply to our cycles:

—co; co = co, which means that we only take the extremities of a chain of coherence relations;
—rf; fr = co, which means that we omit the intermediate reading thread in a sequence of read-from and from-read, just like in the earlier s case;
—fr; co = fr, which means that we omit the intermediate writing thread in a sequence of from-read and coherence.

We call the resulting cycles *static critical cycles*. Thus, mole looks for all of the static critical cycles that it can find in the goto-program given as argument. In addition, it also looks for SC PER LOCATION cycles—that is, coWW, coRW1, coRW2, coWR, and coRR as shown in Figure 6.

In the remainder of this section, we simply write *cycles* for the cycles gathered by mole—that is, static critical cycles and SC PER LOCATION cycles.

9.1.3. Static Search. Looking for cycles poses several challenges, which are also pervasive in static data race analysis (see Kahlon et al. [2007]):

—identify program fragments that may be run concurrently, in distinct threads; and
—identify objects that are shared between these threads.

Finding shared objects may be further complicated by the presence of inline assembly. We find inline assembly in 803 of the packages to be analysed. At present, mole only interprets a subset of inline assembly deemed relevant for concurrency, such as memory barriers, but ignores all other inline assembly.

We can now explain how our pattern search works. Note that our approach does not require analysis of whole, linked, programs, which is essential to achieve scalability to a code base this large. Our analysis proceeds as follows:

(1) identify candidate functions that could act as *entry points* for a thread being spawned (an entry point is a function such that its first instruction will be scheduled for execution when creating a thread);
(2) group these candidate entry points, as detailed later, according to shared objects accessed, where we consider an object as *shared* when it either has static storage duration (and is not marked thread local), or is referenced by a shared pointer;
(3) assuming concurrent execution of the threads in each such group, enumerate cycles using the implementation from Alglave et al. [2013a] with a flow-insensitive points-to analysis and, in order to include SC PER LOCATION cycles, weaker restrictions than when exclusively looking for critical cycles;

(4) classify the candidates following their patterns (i.e., using the litmus naming scheme that we outlined in Table III) and the axioms of the model. The categorisation according to axioms proceeds by testing the sequence of relations occurring in a cycle against the axioms of Figure 5; we detail this step below.

Note that our analysis does not take into account program logic (e.g., locks) that may forbid the execution of a given cycle. If no execution of the (concurrent) program includes a certain cycle, we call it a *spurious* cycle, and refer to others as *genuine* cycles. Note that this definition is independent of fences or dependencies that may render a cycle forbidden for a given (weak) memory model. Our notion of genuine simply accounts for the feasibility of a concurrent execution. This overapproximation, and the fact that we also overapproximate on possible entry points, means that any numbers of cycles given in this section cannot be taken as a quantitative analysis of cycles that would be actually executed.

With this approach, instead, we focus on not missing cycles rather than avoiding the detection of spurious cycles. In this sense, the results are best compared to compiler warnings. Performing actual proofs of cycles being either spurious or genuine is an undecidable problem. In principle, we could thus only do so in a best-effort manner, akin to all software verification tools aiming at precise results. In actual practice, however, the concurrent reachability problem to be solved for each such cycle will be a formidable challenge for current software verification tools, including several additional technical difficulties (e.g., devising complex data structures as input values), as we are looking at real-world software rather than stylised benchmarks. With more efficient tools such as the one of Alglave et al. [2013b], we hope to improve on this situation in future work, considering that with the tool of Alglave et al. [2013b], we managed to verify selected real-world concurrent systems code for the first time.

We now explain these steps in further detail and use the Linux Read-Copy-Update (RCU) code [McKenney and Walpole 2007] as an example. Figure 40 shows a code snippet, which was part of the benchmarks that we used in Alglave et al. [2013b], employing RCU. The original code contains several macros, which were expanded using the C preprocessor.

Finding Entry Points. To obtain an overapproximate set of cycles even for, say, library code, which does not have a defined entry point and thus may be used in a concurrent context even when the code parts under scrutiny do contain thread spawn instructions, we consider thread entry points as follows:

—explicit thread entries via POSIX or kernel thread create functions;
—any set of functions $f_1, \ldots, f_n$, provided that f_i is not (transitively) called from another function f_j in this set and f_i has external linkage (see ISO [2011, Sec. 5.1.1.1]); and
—for mutually recursive functions an arbitrary function from this set of recursive functions.

For any function identified as entry point, we create three threads, thereby accounting for multiple concurrent access to shared objects only used by a single function, but also for cases where one of the called functions is running in an additional thread. Note that in the case of analysing library functions, this may violate assumptions about an expected client-side locking discipline expressed in documentation, as is the case with any library not meant to be used in a concurrent context. As discussed earlier, we did restrict our experiments to those software packages that do contain some reference to POSIX threads or Linux kernel threads anywhere in their code—but this is a heuristic filter only.

```
01 struct foo *gbl_foo;
02
03 struct foo foo1, foo2;
04
05 spinlock_t foo_mutex=(spinlock_t){ { .rlock = { .raw_lock = { 0 }, } } };
06
06 void* foo_update_a(void* new_a)
07 {
08   struct foo *new_fp;
09   struct foo *old_fp;
10
11   foo2.a=100;
12   new_fp = &foo2;
13   spin_lock(&foo_mutex);
14   old_fp = gbl_foo;
15   *new_fp = *old_fp;
16   new_fp->a = *(int*)new_a;
16
17   ({ __asm__ __volatile__ ("lwsync" " " : : :"memory");
18      ((gbl_foo)) = (typeof(*(new_fp)) *)((new_fp)); });
19
20   spin_unlock(&foo_mutex);
21   synchronize_rcu();
22   return 0;
23 }
24
25 void* foo_get_a(void* ret)
26 {
26   int retval;
27   rcu_read_lock();
28   retval=({typeof(*(gbl_foo)) *_________p1 =
29            (typeof(*(gbl_foo))*)(*(volatile typeof((gbl_foo))*)&((gbl_foo)));
30            do { } while (0); ; do { } while(0);
31            ((typeof(*(gbl_foo)) *)(_________p1)); })->a;
32   rcu_read_unlock();
33   *(int*)ret=retval;
34   return 0;
35 }
36
36 int main()
37 {
38   foo1.a=1;
39   gbl_foo=&foo1;
40   gbl_foo->a=1;
41
42   int new_val=2;
43   pthread_create(0, 0, foo_update_a, &new_val);
44   static int a_value=1;
45   pthread_create(0, 0, foo_get_a, &a_value);
46
46   assert(a_value==1 || a_value==2);
47 }
```

Fig. 40. Code example from RCU.

For RCU, as shown in Figure 40, we see several functions (or function calls): `main`, `foo_get_a`, `foo_update_a`, `spin_lock`, `spin_unlock`, `synchronize_rcu`, `rcu_read_lock`, and `rcu_read_unlock`. If we discard `main`, we no longer have a defined entry point nor POSIX thread creation through `pthread_create`. In this case, our algorithm would consider `foo_get_a` and `foo_update_a` as the only potential thread entry points, because all other functions are called from one of these two, and there is no recursion.

Finding Threads' Groups. As a next step, we form groups of threads using the identified thread entry points. We group the functions f_i and f_j if and only if the set of objects read or written by f_i or any of the functions (transitively) called by f_i has a nonempty intersection with the set for f_j. Note the transitivity in this requirement: for functions f_i, f_j, f_k with f_i and f_j sharing one object, and f_j and f_k sharing another object, all three functions end up in one group. In general, however, we may obtain several groups of such threads, which are then analysed individually.

When determining shared objects, as noted earlier, pointer dereferencing has to be taken into account. This requires the use of points-to analyses, for which we showed that theoretically they can be sound [Alglave et al. 2011a], even under weak memory models. In practice, however, pointer arithmetic, field-sensitivity, and interprocedural operation require a performance-precision trade-off. In our experiments, we use a flow-insensitive, field-insensitive, and interprocedural analysis. We acknowledge that we may thus still be missing certain cycles due to the incurred incompleteness of the points-to analysis.

For RCU, `main`, `foo_get_a`, and `foo_update_a` form a group, because they jointly access the pointer `gbl_foo` as well as the global objects `foo1` and `foo2` through this pointer. Furthermore, `main` and `foo_update_a` share the local object `new_val`, and `main` and `foo_get_a` share `a_value`, both of which are communicated via a pointer.

Finding Patterns. With thread groups established, we enumerate cycles as described in Alglave et al. [2013a]. We briefly recall this step here for completeness:

—we first construct one control-flow graph (CFG) per thread;
—then, we add communication edges between shared memory accesses to the same object, if at least one of these objects is a write (this is the cmp relation in the definition of critical cycles given at the beginning of this section);
—we enumerate all cycles amongst the CFGs and communication edges using Tarjan's 1973 algorithm [Tarjan 1973], resulting in a set that also contains all critical cycles (but possibly more); and
—as final step, we filter the set of cycles for those that satisfy the conditions of static critical cycles or SC PER LOCATION as described earlier.

Let us explain how we may find the mp pattern (see Figure 8) in RCU. The writing thread T_0 is given by the function `foo_update_a`, the reading thread T_1 by the function `foo_get_a`. Now, for the code of the writing thread T_0: in `foo_update_a`, we write `foo2` at line 11, then we have an lwsync at line 17 and a write to `gbl_foo` at line 18.

For the code of the reading thread T_1: the function `foo_get_a` first copies the value of `gbl_foo` at line 29 into the local variable `_________p1`. Now, note that the write to `gbl_foo` at line 18 made `gbl_foo` point to `foo2`, due to the assignment to `new_fp` at line 12.

Thus, dereferencing `_________p1` at line 31 causes a read of the object `foo2` at that line. Observe that the dereferencing introduces an address dependency between the read of `gbl_foo` and the read of `foo2` on T_1.

Categorisation of Cycles. As we said earlier, for each cycle that we find, we apply a categorisation according to the axioms of our model (see Figure 5). For the purpose of this

Table XIII. Patterns in PostgreSQL

pattern	# cycles
r (see Fig. 16)	93
w+rr+2w	68
w+rr+wr+ww	62
z6.4	54
sb (see Fig. 14)	37
2+2w (see Fig. 13(a))	25
w+rwc (see Fig. 19)	23
mp (see Fig. 8)	16
w+rw	14
s (see Fig. 39)	14
z6.5	6
w+rw+wr	6
w+rw+2w	4
z6.0	2
wrc (see Fig. 11)	2
lb (see Fig. 7)	2
irrwiw	2
coWR (see Fig. 6)	19
coWW (see Fig. 6)	6
coRW1 (see Fig. 6)	4
coRW2 (see Fig. 6)	4

categorisation, we instantiate our model for SC (see Figure 21). We use the sequence of relations in a given cycle: for example, for mp, this sequence is lwsync; rfe; dp; fre. We first test if the cycle is an SC PER LOCATION cycle: we check if all relations in our input sequence are either po-loc or com. If, as for mp, this is not the case, we proceed with the test for NO THIN AIR. Here, we check if all relations in our sequence match hb (i.e., po $\cup$ fences $\cup$ rfe). As mp includes an fre, the cycle cannot be categorised as NO THIN AIR, and we proceed to OBSERVATION. Starting from fre, we find lwsync $\in$ prop (as prop $=$ po $\cup$ fences $\cup$ rf $\cup$ fr on SC) and rfe; dp $\in$ hb*. Thus, we categorise the cycle as as a observation cycle. In the general case, we check for PROPAGATION last.

Litmus Tests. We exercised mole on the set of litmus tests that we used for exercising cbmc (see Section 8.4). For each test, we find its general pattern (using the naming scheme that we presented in Section 4): for example, for mp, we find the cycle po; rfe; po; fre. Note that our search looks for memory barriers but does not try to look for dependencies; this means that for the variant mp+lwfence+addr of the mp pattern, we find the cycle lwfence; rfe; po; fre, where the barrier appears but not the dependency.

Examples That We Studied Manually Before. Examples that we have studied in the past (see Alglave et al. [2013a, 2013b]) include Apache, PostgreSQL, and RCU, as mentioned in Section 8.4. We analysed these examples with mole to confirm the patterns that we had found before.[14]

In Apache, we find five patterns distributed over 75 cycles: 4 $\times$ mp (see Figure 8); 1 $\times$ s (see Figure 39); 28 $\times$ coRW2, 25 $\times$ coWR, and 17 $\times$ coRW1 (see Figure 6).

In PostgreSQL, we find 22 different patterns distributed over 463 cycles. We give the details in Table XIII.

[14]In the remainder of this paper, we mention patterns that we have not displayed in the article, and that do not follow the convention outlined in Table III: z6.[0-5], 3.2w, or irrwiw. For the sake of brevity, we do not show them in the article and refer the reader to the companion Web site: `http://diy.inria.fr/doc/gen.html#naming`.

Table XIV. Patterns in RCU

pattern	# cycles	memory locations	line numbers
2+2w (see Fig. 13(a))	6	foo2, gbl_foo	1, 3, 16
3.2w	4	foo1, foo2, gbl_foo	3, 16, 39, 40
w+rr+ww+ww	3	foo1, foo2, gbl_foo	3, 14, 15, 16, 39
z6.5	2	foo1, foo2, gbl_foo	3, 11, 14, 39, 40
r (see Fig. 16)	2	foo1, foo2	3, 11, 15
mp (see Fig. 8)	2	foo1, gbl_foo	14, 15, 38, 39
w+rr+ww+wr	2	foo1, foo2, gbl_foo	3, 11, 14, 29, 31, 39
z6.3	1	foo1, foo2, gbl_foo	3, 16, 29, 31
w+rr+2w	1	foo1, gbl_foo	29, 31, 38, 39
coWW (see Fig. 6)	1	foo2	15 16

In RCU we find nine patterns in 23 critical cycles, as well as one SC PER LOCATION cycle. We give the details in Table XIV. For each pattern, we give one example cycle: we refer to the excerpt in Figure 40 to give the memory locations and line numbers[15] it involves. Note that we list an additional example of mp in the table, different from the one explained earlier.

9.2. Results for Debian 7.1

We report on the results of running mole on 137,163 object files generated while compiling source packages using goto-cc, in 1,251 source packages of the Debian Linux distribution, release 7.1.

We provide all of the files compiled with goto-cc our experimental data (i.e., the patterns per packages) at `http://diy.inria.fr/mole`.

Our experiment runs on a system equipped with eight cores and 64GB of main memory. In our setup, we set the time and memory bounds for each object file subject to cycle search to 15 minutes and 16GB of RAM. We spent more than 199 CPU days in cycle search, yet 19, 930 runs did not finish within the preceding time and memory limits. More than 50% of time is spent within cycle enumeration for a given graph of CFGs and communication edges, whereas only 12% is spent in the points-to analysis. The remaining time is consumed in generating the graph. The resulting 79GB of raw data were further processed to determine the results presented next.

Note that, at present, we have no information on how many of the detected cycles may be spurious for one of the reasons discussed earlier. We aim to refine these results in future work.

9.2.1. General Results. Next, we give a general overview of our experiments. We detected a total of 86,206,201 critical cycles, plus 11,295,809 SC PER LOCATION cycles. Amongst these, we find 551 different patterns. Figure 41 gives the 30 most frequent patterns.

The source package with most cycles is mlterm (a multilingual terminal emulator, `http://packages.debian.org/wheezy/mlterm`, 4,261,646 cycles) with the most frequently occurring patterns iriw (296,219), w+rr+w+rr+w+rw (279,528), and w+rr+w+rw+w+rw (218,061). The source package with the widest variety of cycles is ayttm (an instant messaging client, `http://packages.debian.org/wheezy/ayttm`, 238 different patterns); its most frequent patterns are z6.4 (162,469), z6.5 (146,005), and r (90,613).

We now give an account of what kind of patterns occur for a given functionality. By "functionality," we mean what the package is meant for, such as Web servers (httpd),

[15]Lines 1 and 3 result from initialisation of objects with static storage duration, as prescribed by the C11 standard [ISO 2011, Sec. 6.7.9].

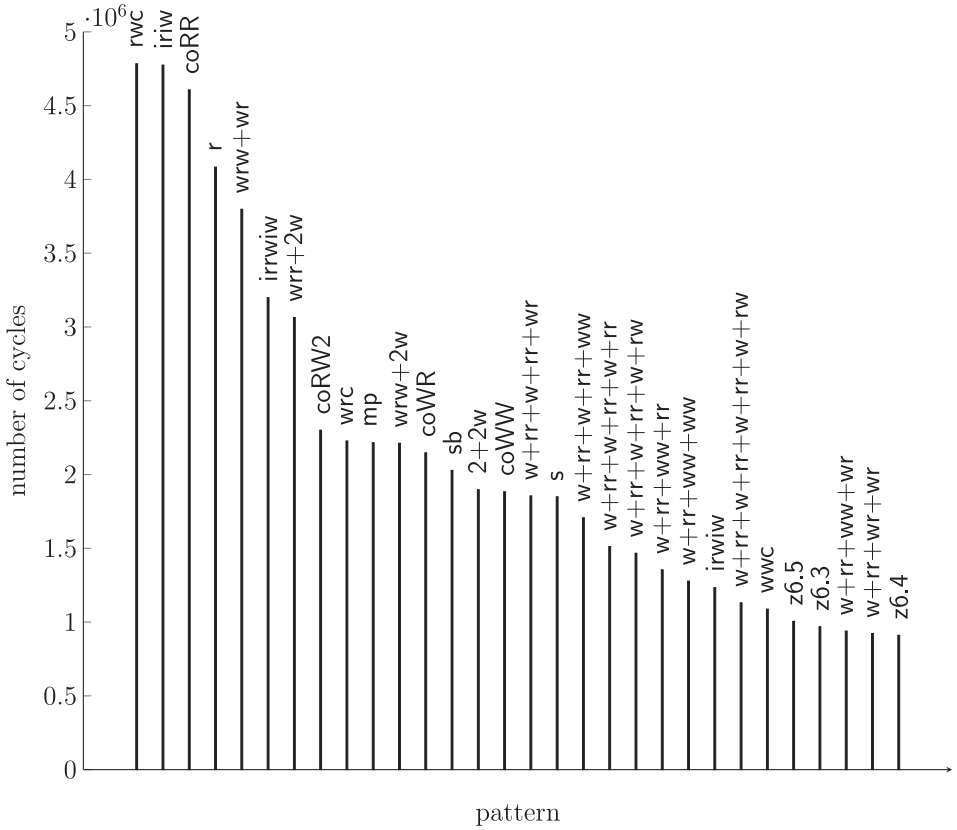


Fig. 41. Thirty most frequent patterns.

Table XV. Patterns per Functionality

function	patterns	typical packages
httpd (11)	wrr+2w (30,506), mp (27,618), rwc (13,324)	libapache2-mod-perl2 (120,869), apache2 (70,283), webfs (27,260)
mail (24)	w+rr+w+rw+ww (75,768), w+rr+w+rr+ww (50,842), w+rr+w+rr+w+rw (45,496)	opendkim (702 534), citadel (337,492), alpine (105,524)
games (57)	2+2w (198,734), r (138,961), w+rr+w+rr+wr (134,066)	spring (1,298,838), gcompris (559,905), liquidwar (257,093)
libs (266)	iriw (468,053), wrr+2w (387,521), irrwiw (375,836)	ecore (1,774,858), libselinux (469,645), psqlodbc (433,282)

mail clients (mail), video games (games), or system libraries (libs). For each functionality, Table XV gives the number of packages (e.g., 11 for httpd), the three most frequent patterns within that functionality with their number of occurrences (e.g., 30,506 for wrr+2w in httpd), and typical packages with the number of cycles contained in that package (e.g., 70,283 for apache2).

9.2.2. Summary per Axiom. Table XVI gives a summary of the patterns we found, organised per axioms of our model (see Section 4). We chose a classification with respect to SC—that is, we fixed prop to be defined as shown in Figure 21. For each axiom, we also give some typical examples of packages that feature patterns relative to this axiom.

Table XVI. Patterns per Axiom

axiom	# patterns	typical packages
SC PER LOCATION	11,295,809	vips (412,558), gauche (391,180), python2.7 (276,991)
NO THIN AIR	445,723	vim (36,461), python2.6 (25,583), python2.7 (16,213)
OBSERVATION	5,786,239	mlterm (285,408), python2.6 (183,761), vim (159,319)
PROPAGATION	79,974,239	isc-dhcp (891,673), cdo (889,532), vim (878,289)

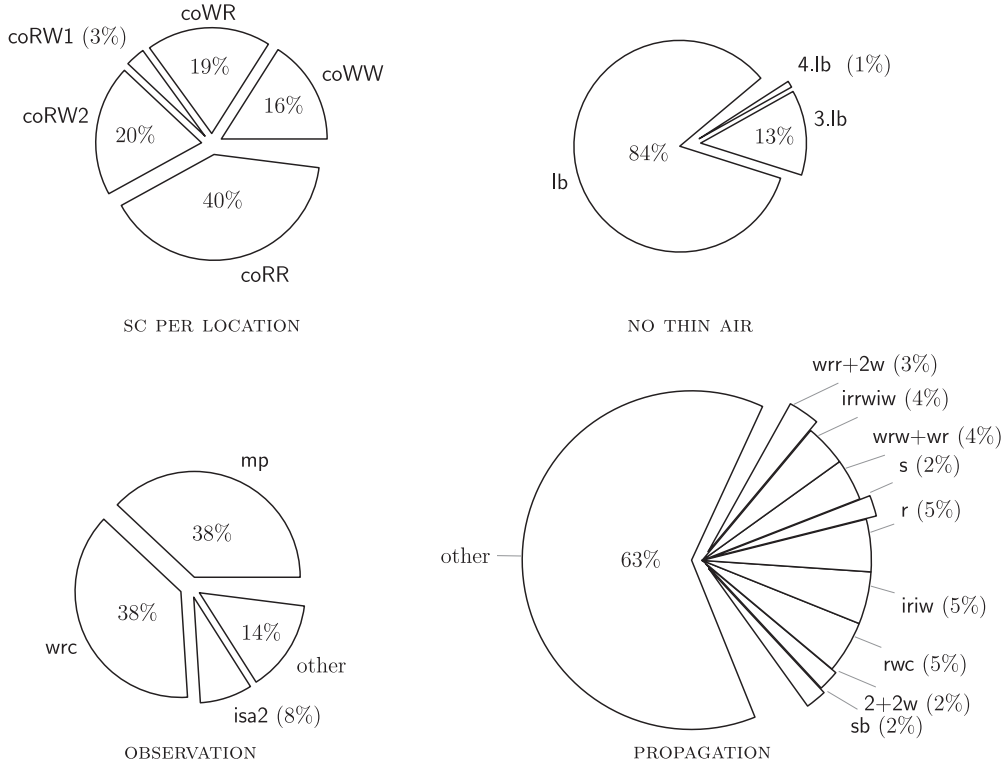


Fig. 42. Proportions of patterns per axiom.

We now give a summary of the patterns we found, organised by axioms. Several distinct patterns can correspond to the same axiom—for example, mp, wrc, and isa2 all correspond to the OBSERVATION axiom (see Section 4). For the sake of brevity, we do not list all 551 patterns. Figure 42 gives one pie chart of patterns per axiom.

Observations. We found 4,775,091 occurrences of iriw (see Figure 20), which represents 4.897% of all cycles that we have found. As such, it is the second most frequent pattern detected, which appears to invalidate the folklore claim that iriw is rarely used. It should be noted, however, that these static cycles need not correspond to genuine ones that are actually observed in executions, as discussed further later.

We found 4,083,639 occurrences of r (see Figure 16), which represents 4.188% of the cycles that we have found. Observe that r appears in PostgreSQL (see Table XIII) and RCU (see Table XIV), as well as in the 30 most frequent patterns (see Figure 41). This seems to suggest that a good model needs to handle this pattern properly.

We also found 4,606,915 occurrences of coRR, which corresponds to the acknowledged ARM bug that we presented in Section 8. This represents 4.725% of all cycles that we

have found. Additionally, we found 2,300,724 occurrences of coRW2, which corresponds to the violation of SC PER LOCATION, observed on ARM machines that we show in Figure 34. This represents 2.360% of all cycles that we have found. These two percentages perhaps nuance the severity of the ARM anomalies that we expose in Section 8.

We believe that our experiments with mole provide results that could be used by programmers or static analysis tools to identify where weak memory may come into play and ensure that it does not introduce unexpected behaviours. Moreover, we think that the data that mole gathers can be useful to both hardware designers and software programmers.

While we do provide quantitative data, we would like to stress that, at present, we have little information on how many of the detected cycles are genuine. Many of the considered cycles may be spurious, either because of additional synchronisation mechanisms such as locks, or simply because the considered program fragments are not executed concurrently in any concrete execution. Thus, as said previously, at present, the results are best understood as warnings similar to those emitted by compilers. In future work, we will work towards the detection of spurious cycles, as well as aim at studying particular software design patterns that may give rise to the most frequently observed patterns of our models.

We nevertheless performed manual spot tests of arbitrarily selected cycles in the packages 4store, acct, and acedb. For instance, the SC PER LOCATION patterns in the package acct appear genuine, because the involved functions could well be called concurrently as they belong to a memory allocation library. An analysis looking at the entire application at once would be required to determine whether this is the case. For other cases, however, it may not at all be possible to rule out such concurrent operations: libraries, for instance, may be used by arbitrary code. In those cases, only locks (or other equivalent mutual exclusion primitives) placed on the client side would guarantee data-race free (and thus weak-memory insensitive) operation. The same rationale applies for other examples that we looked at: although our static analysis considers this case to achieve the required safe overapproximation, the code involved in some of the iriw cycles in the package acedb or 4store is not obviously executed concurrently at present. Consequently, these examples of iriw might be spurious, but we note that no locks are in place to guarantee this.

For the RCU and PostgreSQL examples presented in this article, we use harnesses that perform concurrent execution. For RCU, this mimics the intended usage scenario of RCU (concurrent readers and writers), and in the case of PostgreSQL, this was modelled after a regression test[16] built by PostgreSQL's developers. Consequently, we are able to tell apart genuine and spurious cycles in those cases.

For PostgreSQL, the SC PER LOCATION patterns (coWW, coWR, coRW1, and coRW2, listed in Table XIII) and the critical cycles described in detail in Alglave et al. [2013a] are genuine: one instance of lb (amongst the 2 listed in Table XIII) and one instance of mp (amongst the 16 listed).

For RCU, the coWW cycle listed in Table XIV and the instance of mp described earlier (see top of p. 65) are genuine, but note that the latter contains synchronisation using lwsync, which means that the cycle is forbidden on Power.

All other cycles appear to be genuine as well but are trivially forbidden by the ordering of events implied by spawning threads. For example, we report instances of mp in RCU over lines 38 and 39 in function `main` as first thread, and lines 14 and 15 in function `foo_update_a` as second, and seemingly concurrent, thread. As that second thread is only spawned after execution of lines 38 and 39, no true concurrency is possible.

[16]See the attachment at `http://www.postgresql.org/message-id/24241.1312739269@sss.pgh.pa.us`.

10. CONCLUSION

In closing, we recapitulate the criteria that we listed in the introduction and explain how we address each of them.

Stylistic Proximity of Models. The framework that we presented embraces a wide variety of hardware models, including SC, x86 (TSO), Power, and ARM. We also explained how to instantiate our framework to produce a significant fragment of the C++ memory model, and we leave the definition of the complete model (in particular including consume atomics) for future work.

Concision. Concision is demonstrated in Figure 38, which contains the *unabridged* specification of our Power model.

Efficient Simulation and Verification. The performance of our tools, our new simulator herd (see Table IX), and the bounded model checker cbmc adapted to our new Power model (see Table XI) confirm (following Mador-Haim et al. [2012] and Alglave et al. [2013b]) that the modelling style is crucial.

Soundness w.r.t. Hardware. To the best of our knowledge, our Power and ARM models are to this date not invalidated by hardware, except for the 31 surprising ARM behaviours detailed in Section 8. Moreover, we keep running experiments regularly, which we record at `http://diy.inria.fr/cats`.

Adaptability of the Model. This was demonstrated by the ease with which we were able to modify our model to reflect the subtle mp+dmb+fri-rfi-ctrlisb behaviour (see Section 8).

Architectural Intent. To the best of our knowledge, our Power model does not contradict the architectural intent, in that we build on the model of Sarkar et al. [2011], which should reflect said intent, and that we have regular contacts with hardware designers. For ARM, we model the mp+dmb+fri-rfi-ctrlisb behaviour, which is claimed to be intended by ARM designers.

Account for What Programmers Do. With our new tool mole, we explored the C code base of the Debian Linux distribution version 7.1 (about 200 millions lines of code) to collect statistics of concurrency patterns occurring in real-world code. Just like our experiments on hardware, we keep running our experiments on Debian regularly; we record them at `http://diy.inria.fr/mole`.

***

As future work, on the modelling side, we will integrate the semantics of the lwarx and stwcx. Power instructions (and their ARM equivalents ldrex and strex), which are used to implement locking or compare-and-swap primitives. We expect their integration to be relatively straightforward: the model of Sarkar et al. [2012] uses the concept of a write reaching coherence point to describe them, a notion that we have in our model as well.

On the rationalist side, it remains to be seen if our model is well suited for proofs of programs: we regard our experiments w.r.t. verification of programs as preliminary.

ACKNOWLEDGMENTS

We thank Nikos Gorogiannis for suggesting that the extension of the input files for herd should be .cat and Daniel Kroening for infrastructure for running mole. We thank Carsten Fuhs (even more so since we forgot to thank him in Alglave et al. [2013b]) and Matthew Hague for their patient and careful comments on a draft. We thank Mark Batty, Shaked Flur, Vinod Grover, Viktor Vafeiadis, Tyler Sorensen, and Gadi Tellez for comments on a draft. We thank our reviewers for their careful reading, comments, and suggestions. We thank Arthur Guillon for his help with the simulator of Boudol et al. [2012]. We thank Susmit Sarkar, Peter

Sewell, and Derek Williams for discussions on the Power model(s). Finally, this article would not have been the same without the last year of discussions on related topics with Richard Bornat, Alexey Gotsman, Peter O'Hearn, and Matthew Parkinson.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. 2012. Counter-example guided fence insertion under TSO. In *Proceedings of TACAS*. Springer-Verlag, Berlin, Heidelberg, 204–219.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. 2013. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *Proceedings of TACAS*. Springer-Verlag, Berlin, Heidelberg, 530–536.

Allon Adir, Hagit Attiya, and Gil Shurek. 2003. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Transactions on Parallel and Distributed Systems* 14, 5, 502–515.

Sarita V. Adve and Hans-Juergen Boehm. 2010. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM* 53, 8, 90–101.

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *IEEE Computer* 29, 12, 66–76.

Jade Alglave. 2010. *A Shared Memory Poetics*. Ph.D. Dissertation. Université Paris 7.

Jade Alglave. 2012. A formal hierarchy of weak memory models. *Formal Methods in System Design* 41, 2, 178–210.

Jade Alglave, Anthony C. J. 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 *Proceedings of AMP*. ACM Press, New York, NY, 13–24.

Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig. 2011a. Soundness of data flow analyses for weak memory models. In *Proceedings of APLAS*. Springer-Verlag, Berlin, Heidelberg, 272–288.

Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013a. Software verification for weak memory via program transformation. In *Proceedings of ESOP*. Springer-Verlag, Berlin, Heidelberg, 512–532.

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013b. Partial orders for efficient bounded model checking of concurrent software. In *Proceedings of CAV*. Springer-Verlag, Berlin, Heidelberg, 141–157.

Jade Alglave and Luc Maranget. 2011. Stability in weak memory models. In *Proceedings of CAV*. Springer-Verlag, Berlin, Heidelberg, 50–66.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in weak memory models. In *Proceedings of CAV*. Springer-Verlag, Berlin, Heidelberg, 258–272.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011b. Litmus: Running tests against hardware. In *Proceedings of TACAS*. Springer-Verlag, Berlin, Heidelberg, 41–44.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in weak memory models (extended version). *Formal Methods in System Design* 40, 2, 170–205.

ARM Ltd. 2010. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*. ARM Ltd.

ARM Ltd. 2011. *Cortex-A9 MPCore, Programmer Advice Notice, Read-after-Read Hazards*. ARM Ltd.

Arvind and Jan-Willem Maessen. 2006. Memory Model = Instruction Reordering + Store Atomicity. In *Proceedings of ISCA*. IEEE Computer Society, Washington, DC, 29–40.

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of POPL*. ACM Press, New York, NY, 7–18.

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's decidable about weak memory models? In *Proceedings of ESOP*. Springer-Verlag, Berlin, Heidelberg, 26–46.

Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. 2011. Getting rid of store-buffers in TSO analysis. In *Proceedings of CAV*. Springer-Verlag, Berlin, Heidelberg, 99–115.

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *Proceedings of POPL*. ACM Press, New York, NY, 235–248.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of POPL*. ACM Press, New York, NY, 55–66.

Yves Bertot and Pierre Casteran. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag.

Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of PLDI*. ACM Press, New York, NY, 68–78.

Hans-Juergen Boehm and Sarita V. Adve. 2012. You don't know jack about shared variables or memory models. *Communications of the ACM* 55, 2, 48–54.

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and enforcing robustness against TSO. In *Proceedings of ESOP*. Springer-Verlag, Berlin, Heidelberg, 533–553.

Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding robustness against total store ordering. In *Proceedings of ICALP (2)*. Springer-Verlag, Berlin, Heidelberg, 428–440.

Gérard Boudol and Gustavo Petri. 2009. Relaxed memory models: An operational approach. In *Proceedings of POPL*. ACM Press, New York, NY, 392–403.

Gérard Boudol, Gustavo Petri, and Bernard P. Serpette. 2012. Relaxed operational semantics of concurrent programming languages. In *Proceedings of EXPRESS/SOS*. 19–33.

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of PLDI*. ACM, New York, NY, 12–21.

Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. 2013. *Understanding eventual consistency*. Technical Report TR-2013-39. Microsoft Research.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: Specification, verification, optimality. In *Proceedings of POPL*. ACM Press, New York, NY, 271–284.

Sebastian Burckhardt and Madan Musuvathi. 2008. Memory Model Safety of Programs. In *Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly (EC)*^{2}.

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java memory model: Operationally, denotationally, axiomatically. In *Proceedings of ESOP*. Springer-Verlag, Berlin, Heidelberg, 331–346.

Nathan Chong and Samin Ishtiaq. 2008. Reasoning about the ARM weakly consistent memory model. In *Proceedings of MSPC*. ACM Press, New York, NY, 16–19.

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *Proceedings of TACAS*. Springer-Verlag, Berlin, Heidelberg, 168–176.

William Collier. 1992. *Reasoning About Parallel Architectures*. Prentice Hall.

Compaq Computer Corp. 2002. *Alpha Architecture Reference Manual*. Compaq Computer Corp.

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of ISCA*. ACM Press, New York, NY, 15–26.

Jacob Goodman. 1989. *Cache consistency and sequential consistency*. Technical Report. IEEE Scalable Coherent Interface Group.

Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj. 2004. QB or not QB: An efficient execution verification tool for memory orderings. In *Proceedings of CAV*. Springer-Verlag, Berlin, Heidelberg, 401–413.

Michael J. C. Gordon. 2002. Relating event and trace semantics of hardware description languages. *Computer Journal* 45, 1, 27–36.

Richard Grisenthwaite. 2009. *ARM Barrier Litmus Tests and Cookbook*. ARM Ltd.

Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, Juin-Yeu Joseph Lu, and Sridhar Narayanan. 2004. TSOtool: A program for verifying memory systems using the memory consistency model. In *Proceedings of ISCA*. IEEE Computer Society, Washington, DC, 114–123.

C. A. R. Hoare and Peter E. Lauer. 1974. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica* 3, 135–153.

David Howells and Paul E. MacKenney. 2013. Linux Kernel Memory Barriers, 2013 version. Retrieved May 29, 2014, from `https://www.kernel.org/doc/Documentation/memory-barriers.txt`.

IBM Corp. 2009. *Power ISA Version 2.06*. IBM Corp.

Intel Corp. 2002. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*. Intel Corp.

Intel Corp. 2009. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corp.

ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization.

Daniel Jackson. 2002. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology* 11, 2, 256–290.

Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of CAV*. Springer-Verlag, Berlin, Heidelberg, 226–239.

Michael Kuperstein, Martin T. Vechev, and Eran Yahav. 2010. Automatic inference of memory fences. In *Proceedings of FMCAD*. IEEE Computer Society, Washington, DC, 111–119.

Michael Kuperstein, Martin T. Vechev, and Eran Yahav. 2011. Partial-coherence abstractions for relaxed memory models. In *Proceedings of PLDI*. ACM Press, New York, NY, 187–198.

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computer Systems* 28, 9, 690–691.

Richard J. Lipton and Jonathan S. Sandberg. 1988. *PRAM: A scalable shared memory*. Technical Report CS-TR-180-88. Princeton University.

Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. 2012. Dynamic synthesis for relaxed memory models. In *Proceedings of PLDI*. ACM Press, New York, NY, 429–440.

Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. 2010. Generating litmus tests for contrasting memory consistency models. In *Proceedings of CAV*. Springer-Verlag, Berlin, Heidelberg, 273–287.

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*. Springer-Verlag, Berlin, Heidelberg, 495–512.

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of POPL*. ACM Press, New York, NY, 378–391.

Paul E. McKenney and Jonathan Walpole. 2007. What is RCU, fundamentally? Retrieved May 29, 2014, from `http://lwn.net/Articles/262464/`.

Gil Neiger. 2000. A taxonomy of multiprocessor memory-ordering models. In *Tutorial and Workshop on Formal Specification and Verification Methods for Shared Memory Systems*.

Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. 2011. Lem: A lightweight tool for heavyweight semantics. In *Proceedings of ITP*. Springer-Verlag, Berlin, Heidelberg, 363–369.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs*. Springer-Verlag, Berlin, Heidelberg, 391–407.

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*. ACM Press, New York, NY, 311–322.

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding power multiprocessors. In *Proceedings of PLDI*. ACM Press, New York, NY, 175–186.

Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *Proceedings of POPL*. ACM Press, New York, NY, 379–391.

Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming and Language Systems* 10, 2, 282–312.

SPARC International Inc. 1992. *The SPARC Architecture Manual Version 8*. SPARC International Inc.

SPARC International Inc. 1994. *The SPARC Architecture Manual Version 9*. SPARC International Inc.

Robert C. Steinke and Gary J. Nutt. 2004. A unified theory of shared memory consistency. *Journal of the ACM* 51, 5, 800–849.

Robert Tarjan. 1973. Enumeration of the elementary circuits of a directed graph. *SIAM Journal of Computing* 2, 3, 211–216.

Joel M. Tendler, J. Steve Dodson, J. S. Fields Jr., Hung Le, and Balaram Sinharoy. 2002. POWER4 system microarchitecture. *IBM Journal of Research and Development* 46, 1, 5–26.

Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of PLDI*. ACM Press, New York, NY, 341–350.

Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. 2004. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *Proceedings of IPDPS*. IEEE Computer Society, Washington, DC, 31b.

Francesco Zappa Nardelli, Peter Sewell, Jaroslav Sevcik, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. 2009. Relaxed memory models must be rigorous. In *Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly (EC)*^{2}.

Received August 2013; revised December 2013; accepted February 2014