# Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model

Christopher Pulte
University of Cambridge
United Kingdom
Christopher.Pulte@cl.cam.ac.uk

Jean Pichon-Pharabod
University of Cambridge
United Kingdom
Jean.Pichon@cl.cam.ac.uk

Jeehoon Kang*
KAIST
Korea
jeehoon.kang@kaist.ac.kr

Sung-Hwan Lee
Seoul National University
Korea
sunghwan.lee@sf.snu.ac.kr

Chung-Kil Hur†
Seoul National University
Korea
gil.hur@sf.snu.ac.kr

## Abstract

For ARMv8 and RISC-V, there are concurrency models in two styles, extensionally equivalent: axiomatic models, expressing the concurrency semantics in terms of global properties of complete executions; and operational models, that compute incrementally. The latter are in an abstract microarchitectural style: they execute each instruction in multiple steps, out-of-order and with explicit branch speculation. This similarity to hardware implementations has been important in developing the models and in establishing confidence, but involves complexity that, for programming and model-checking, one would prefer to avoid.

We present new more abstract operational models for ARMv8 and RISC-V, and an exploration tool based on them. The models compute the allowed concurrency behaviours incrementally based on thread-local conditions and are significantly simpler than the existing operational models: executing instructions in a single step and (with the exception of early writes) in program order, and without branch speculation. We prove the models equivalent to the existing ARMv8 and RISC-V axiomatic models in Coq. The exploration tool is the first such tool for ARMv8 and RISC-V fast enough for exhaustively checking the concurrency behaviour of a number of interesting examples. We demonstrate using the tool for checking several standard concurrent datastructure and lock implementations, and for interactively stepping through model-allowed executions for debugging.

***Keywords*** Relaxed Memory Models, Operational Semantics, ARM, RISC-V

## 1 Introduction

Writing relaxed-memory concurrent software is notoriously hard. To make matters worse, testing alone cannot give high confidence: the behaviour is highly non-deterministic and certain relaxed behaviours are only observed very rarely, e.g. only once in thousands or millions of executions, sometimes only on particular hardware, sometimes not currently observable at all but allowed by the specification (and so perhaps exhibited with future compiler generations or hardware).

Therefore, precise semantics of the concurrency behaviour and tool support for checking software correctness are highly desirable. More specifically, to safely write such software — typically concurrency library implementations — it is useful to have *(i)* a clear and precise semantics, ideally helping programmers avoid most bugs in the first place, *(ii)* exhaustive exploration tools, to find concurrency bugs or show their absence in bounded executions, and *(iii)* interactive exploration tools, helping to pin down the source of unexpected behaviours of the program.

While such libraries are mostly written in higher-level languages such as C/C++, semantics and tool support at the assembly level offer several benefits. First, after extensive past research, hardware concurrency models are quite well-understood. For higher-level languages such as C/C++ [16] only non-atomic accesses and the sequential consistency and release/acquire fragments are well-understood. High-performance algorithms, however, also involve relaxed 'atomics' and 'consume' atomics, for which there still is not an accepted high-level language semantics [15, 27, 29, 38]. Hence, in practice programmers often write code that does not follow the C/C++ concurrency model, relying instead on the compiled code to provide stronger guarantees, e.g. Linux has its own memory model [9, 25]. For example, in C/C++, often weakening 'consume' or 'acquire' loads to 'relaxed' loads, though unsound in the source language, generates more efficient code that behaves correctly under the hardware memory model. Second (and related), in many cases concurrency libraries include hand-written assembly modules or inline assembly, optimised for performance on particular hardware (requiring a clear understanding of the assembly semantics). Third, the compiler need not be trusted because even compiler-introduced bugs can be detected. Fourth, such tools apply for any source language or memory model: e.g. C, C++, Rust, or Linux.

[*] Work done while at Seoul National University.

[†] Hur is the corresponding author.

Such assembly tools can, for instance, effectively be used in the following scenario of a code base with custom concurrency libraries: the library functions are separately compiled to assembly, debugged and exhaustively checked (up to some bounds) by the exploration tools, and then linked at the assembly level with client software of the remaining code base (in turn perhaps checked/verified using source level tools).

In this paper we present a semantics and tools aiming to address these goals for the widely used ARMv8 processor architecture [12], that has recently moved to a multicopy-atomic semantics [20, 39], and for RISC-V, whose community has recently ratified a very similar memory model [42]. ARMv8 and RISC-V have relaxed memory models that allow the effects of various processor optimisations to become observable to the programmer: instructions execute out-of-order and speculatively (past unresolved conditional or computed branches), speculative writes can be forwarded to program-order-later reads, processors have store queues and caches to speed up memory accesses, etc. At the same time, the architectures give guarantees of coherence, ordering resulting from dependencies and memory barriers, and atomicity of load/store exclusive instructions. The resulting memory models are subtle, and a clear semantics and tool support are particularly valuable.

For ARMv8-A and RISC-V there are two existing styles of semantics and associated tools: axiomatic and operational, with the latter in an abstract microarchitectural style.

The axiomatic model for ARMv8-A [12, 20, 39] is incorporated by the architecture text; it is formalised by Deacon in herd [11]. The RISC-V model, recently defined by the RISC-V Memory Model Task Group (chaired by Lustig), is similar, and is formalised both in Alloy [26] and in herd. These axiomatic models describe the semantics by directly stating properties of the legal executions, as axioms about candidate executions. This makes for abstract and concise definitions. But they only specify global properties of completed executions and do not compute the legal outcomes incrementally. We want to support stepping through executions for debugging. Moreover, herd only supports a few instructions, without a substantial instruction-set architecture (ISA) model, and the Alloy model is a pure memory model; it cannot run machine instructions.

The existing operational models for ARMv8 and RISC-V are variants of the Flat operational model, by Pulte, Flur, et al. [39]. Flat computes the possible executions of a concurrent program incrementally, based on the legal traces from the initial state, where the model transitions enabled in a given state are subject to thread-local conditions. It features substantial ISA models for ARM and RISC-V. Moreover it is integrated into the rmem tool [22, 23, 39–41] that supports exhaustive and interactive exploration, a web user interface, debugging facilities, etc. The biggest example exhaustively checked with Flat so far is a spinlock example from the Linux kernel. The model has an abstract micro-architectural flavour. This strengthens confidence in the model, and was important in developing the model in discussion with hardware architects. However, it makes the model complex. It enables but also requires a programmer to think in terms of hardware mechanisms: the model executes instructions in multiple steps (per instruction) and out-of-order; it has explicit branch speculation and sometimes needs to restart instructions to repair mis-speculation.

In this paper we develop a new operational model, Promising-ARM/RISC-V, and an interactive and exhaustive exploration tool based on it, inspired by the Promising semantics for C11 of Kang et al. [29]. Our model has the abstractness of the axiomatic models but computes locally and incrementally, making it a simpler operational model. In contrast to Flat, this model executes an instruction in a single step and — except early writes — in order, and does not speculate branches. The exploration tool integrates models for large parts of the user-mode ARMv8 [22] and RISC-V [13] ISAs written in Sail [13, 24] (the same as those used by Flat). This provides a significant advantage over the axiomatic models that do not include a substantial ISA model. By integrating into rmem, our model also benefits from the infrastructure it offers, including a web UI, debugging facilities, e.g. the ability to set breakpoints, show DWARF debugging information for compiled code, etc. Even without applying any model checking techniques, our model is the first such tool for ARMv8 or RISC-V with sufficient performance for exhaustively checking several loop-bounded standard concurrent datastructure examples. We also formalise the model for a small idealised ISA and prove it equivalent to the ARMv8/RISC-V axiomatic models for finite executions, in Coq. To summarise, the contributions are:

- Promising-ARM/RISC-V, a simpler and more abstract operational model, executing instructions in program order, not out-of-order and not speculatively (§4, §5).
- A Coq proof of equivalence with the ARMv8 and RISC-V axiomatic models for a small idealised ISA (§6).
- An interactive and exhaustive exploration tool for ARMv8 and RISC-V user-mode assembly programs (§7).
- A demonstration of the tool for several concurrent datastructures, written in C++ and Rust, compiled with GCC or rustc, including the Chase-Lev deque, Michael-Scott queue, and Treiber stack (§8). We demonstrate using the tool also in checking agressively relaxed programs that are unsound in the source memory model but sound in ARMv8/RISC-V.

For the supplementary material, see [2].

***Caveats*** We do not yet model mixed-size accesses (Flat does), since their architecturally intended semantics is still being clarified for ARM and RISC-V. We expect we can cover them by handling certain memory accesses byte-wise and with a more fine-grained register dependency handling, analogously to Flat. We do not yet model read-modify-write instructions that we expect to work analogously to load/store exclusive instructions that we do cover. Like Flat, we only handle user-mode non-vector non-floating-point instructions, no systems features or supervisor mode. While in ARMv8 and RISC-V syntactic dependencies enforce ordering, ARM store exclusives are an exception. As a result, like Flat [39], the ARM model can deadlock (but is nonetheless equivalent to the axiomatic model). The RISC-V model has no such deadlocks (again, like Flat), see §4.3 for details.

## 2 Overview

Our model builds on the work for the C/C++ Promising semantics [29], but has a simpler memory semantics — enabled by the multicopy atomicity and the simpler notion of dependencies in ARMv8/RISC-V — and a new uniform treatment of the dependencies and ordering using timestamps. We now show the main ideas of our model, highlighting the benefits over Flat, and show a key property of the model that allows reducing non-determinism by enumerating the possible final memory states without interleaving reads.

***Out-of-order reads*** We first show how our model explains the effects of out-of-order reads though executing them in order. Consider the following example (for presentation using a simple calculus); assume every location is initialized to 0. Here Thread 1 (left) writes 37 to $x$ and 42 to $y$. The stores are kept in order by a strong barrier dmb.sy. Thread 2 (right) reads $y$. If the value read is 42 it reads $x$, otherwise it executes the instructions from $g$. Despite $e$ introducing a *control dependency* from $d$ to $f$, ARMv8/RISC-V allow executing $f$ before $d$, due to branch speculation, and hence the outcome where $d$ reads 42 and $f$ the initial value $x = 0$.

```
(a) store [x] 37;  ║ (d) r0 := load [y];  // 42
(b) dmb.sy;        ║ (e) if (r0 = 42)
(c) store [y] 42   ║ (f)    r2 := load [x] // 0
                   ║ (g) else ...
```

Following the mechanisms of micro-architecture, Flat allows this execution by branch speculation. In the initial state, Thread 2 can speculatively fetch and execute either branch of the conditional. Fetching the "then" part before the branch condition is resolved (before $a$ and $c$ are propagated) allows $f$ to read the initial write $x = 0$. Once $a$ and $c$ are propagated, $d$ reads from $c$ and resolves the conditional branch. Since the speculation was correct, $f$ can finish, resulting in the example outcome. Instead of fetching $f$, Thread 2 could have also fetched the "else" branch, in which case, after reading $y = 42$, Thread 2 would have detected the mis-speculation and discarded any already-executed instructions of that branch.

In contrast, our model *executes loads in order.* It records the full history of writes propagated so far and allows loads to read old values. The above execution is allowed as follows. Executing $a$, $b$, and $c$ results in memory $[0 : \langle x := y := 0\rangle; 1 : \langle x := 37\rangle; 2 : \langle y := 42\rangle]$. Here 0 to 2 are *timestamps*, list indices in memory. Then we execute Thread 2 sequentially: $d$ can read any write to $y$ in memory, either $\langle y := 0\rangle$ or $\langle y := 42\rangle$. After reading the latter, $e$ takes the "if" branch. Now $f$ can also read either write to $x$, $\langle x := 0\rangle$ or $\langle x := 37\rangle$. This treatment of loads, executing them in program order, leads to a simpler model: simplifying the dependency handling and removing the need to speculate branches and repair incorrect speculation. Moreover, it reduces non-determinism: resulting from the out-of-order execution of reads, and the speculative exploration of possible branch targets — if $e$ was a computed branch, Flat would have to allow fetching *any code location* as a possible successor instruction of $e$.

***Ordering memory accesses with views*** Changing $f$ in the previous example to $\mathtt{r}_2 := \mathtt{load}\ [\mathtt{x} + \mathtt{r}_0 - \mathtt{r}_0]$ makes $f$ *address-dependent* on $d$ and prevents executing $f$ before $d$. In the Flat model the dependency prevents executing $f$ early since the register values involved in computing the location of $f$ are not available until $d$ is done. In our model this is handled with *views.* A view records a timestamp of a memory write to capture some ordering requirement. After executing $a$ to $c$, memory is $[0 : \langle x := y := 0\rangle; 1 : \langle x := 37\rangle; 2 : \langle y := 42\rangle]$, as before. When $d$ reads $\langle y := 42\rangle$ it annotates register $r_0$ with $c$'s timestamp 2. Since $f$ depends on $r_0$, this constrains $f$ and prevents reading $x = 0$, which is "out-of-date" at time 2, due to the previous write $\langle x := 37\rangle$ at timestamp 1.

***Instantaneous instruction execution*** In the next example, PPOCA [41], the instructions $i$ and $j$ after $e$ write and read $z$, and $f$ address-depends on $j$. Assume $d$ reads 42 and $j$ reads from $i$. In ARMv8 and RISC-V, control dependencies prevent early execution of writes, and so $i$ cannot propagate to memory before $d$. Even though $f$ depends on $j$, which reads from $i$, $f$ can execute before $d$, and hence the outcome where $f$ reads $x = 0$ is allowed, because $i$ can *forward* to $j$ (without $i$ propagating). In Flat this is as follows: as before, Thread 2 speculatively executes the "then" branch; $i$ cannot propagate yet, because $e$ is unresolved; however, $i$ can execute some *intra-instruction* steps and determine its location $z$ and value 51 (in the real ISA this involves several register reads and arithmetic). In this half-executed state $i$ can forward its value 51 to $j$, resolving $f$'s location, and allowing it to read $x = 0$.

```
(a) store [x] 37;  ║ (d) r0 := load [y];  // 42
(b) dmb.sy;        ║ (e) if (r0 = 42)
(c) store [y] 42   ║ (i)    store [z] 51;
                   ║ (j)    r1 := load [z];  // 51
                   ║ (f)    r2 := load [x + (r1 - r1)] // 0
                   ║ (g) else ...
```

In our model *all instructions execute instantaneously*, and write forwarding is instead explained by views: a read normally receives a view including the view of the write it read from; if it reads from a write by its own thread, however, it acquires a smaller view. Here, after executing $a$ to $c$ the memory is as shown before; when $d$ reads $y := 42$ it annotates $r_0$ with timestamp 2; due to the control dependency on $r_0$, $i$ must write to memory at a timestamp greater than 2 (as explained later). But when $j$ reads from $i$ it can acquire a smaller view, 0, since it is reading from a write by its own thread. Hence $r_1$ also has view 0 and $f$ can read $x = 0$ from the write history. Executing each instruction in a single step significantly simplifies the model over Flat: conceptually, and by reducing the transitions and rules of the model.

***Out-of-order writes*** In the final example, Thread 1 reads $x$ and writes $y$; Thread 2 reads $y$ and writes what it read to $x$. The outcome where $a$ and $c$ read non-zero values is allowed, as $b$ can execute early. And that is how Flat allows it.

```
(a) r0 := load [x]; // 37 ║ (c) r0 := load [y]; // 37
(b) store [y] 37          ║ (d) store [x] r0
```

In our model, this is explained using *promises*. In a given state $S$, if a thread could take multiple steps from $S$ sequentially (with no steps by other threads) and produce a write $w$, then the thread is allowed to promise $w$ in $S$. A promise binds the promising thread to later *fulfil* the promise. Here, Thread 1 can promise $y := 37$ in the initial state, since executing sequentially it could read $x = 0$ (with $a$) and write $y := 37$ (with $b$). The promise yields memory $[0 : \langle x := y := 0\rangle; 1 : \langle y := 37\rangle]$. Now $c$ can read $y = 37$ and $d$ write $x := 37$. Finally, $a$ can read $x := 37$, and Thread 1 can execute $b$ and *fulfil* the promise $y := 37$. In contrast, due to the data dependency, in the initial state Thread 2 cannot promise $x := 37$: executing sequentially, $c$ must read $y = 0$, so $d$ would write $x := 0$.

As detailed later, dependencies also constrain promises in another way. When a thread promises a write at timestamp $t$, it is later required to fulfil the promise with a view smaller than $t$, effectively preventing writes from being promised "too early", using views.

***Writes first*** The example also allows us to illustrate the key idea for reducing the combinatorial problem of exhaustive enumeration. Naively, exhaustive execution would mean checking all interleavings of the reads and writes/promises. However, in our model for every legal trace there exists an equivalent one in which all promises are done first: for example, for the trace "$a$ (reading 0), $c$ (reading 0), $b$, $d$" the same outcome can be reached as follows: promise $y := 37$ (Thread 1), promise $x := 0$ (Thread 2), read $x = 0$ with $a$, read $y = 0$ with $c$. Due to this property, the model can enumerate all final memory states by exploring only the interleavings of write transitions, without interleaving reads.

We now define the sequential calculus and informally explain the model, before giving the precise definition.

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

**Figure 1.** The language

## 3 Language

To focus on the concurrency aspects we consider the small imperative language of Fig. 1; the executable tool of §7 handles user-mode parts of the real ARMv8 and RISC-V ISAs. Statements include loads, stores, barriers, register assignment, sequential composition, conditionals, and loops. Statements operate on registers, of which we assume we have an infinite supply. A load or store is annotated with (1) a boolean indicating whether it is an exclusive access, and (2) a read or write kind, respectively indicating whether it is a plain access or has special acquire or release ordering. A store writes a bit to a register indicating success or failure. Only store exclusives can fail, non-exclusive stores always succeed, but for uniformity of the syntax and the rules, non-exclusive stores also write the success bit, to an otherwise unused register, that we omit in the syntax. Following the ARM ISA, success is indicated by 0 (here called $\mathsf{v}_{\mathrm{succ}}$), and failure by 1 ($\mathsf{v}_{\mathrm{fail}}$). Whenever a load or store command is not annotated with a memory kind, we mean plain loads and stores; whenever not annotated to be exclusive, we assume it is non-exclusive. So $r := \texttt{load}\ [e]$ is a plain, non-exclusive load, and $\texttt{store}\ [e_1]\ e_2$ is a plain, non-exclusive store.[1]We treat '$(\texttt{if}\ (e)\ s_1\ s_2); s_3$' as equivalent to '$\texttt{if}\ (e)\ (s_1; s_3)\ (s_2; s_3)$' (parenthesising for clarity): control flow is not delimited in assembly programs, and so in our language the instructions in $s_3$ are control-dependent on expression $e$. (This matters for the ordering from control dependencies.) As usual, the executable model bounds loops.

## 4 Promising-ARM/RISC-V, informally

For presentation purposes, we use ARMv8 terminology for barriers: `dmb.sy` for full barriers, etc. We describe the semantics for programs with only plain loads and stores and full

[1] RISC-V has a load-reserve acquire-release and store-conditional acquire-release: instructions with (strong) acquire and release ordering combined. For simplicity of the presentation we omit this, but the executable model handles it. We plan on adding this to the Coq formalisation as well.

barriers. The remaining semantics is a natural extension to the model, detailed in §A of the supplementary material. We first explain the out-of-order execution of loads and how views constrain them in the *view semantics*. We then illustrate how the *promising semantics* extends it to account for the out-of-order execution of stores. Finally, we describe how certification avoids executions with unfulfilled promises.

### 4.1 View semantics

The view semantics underlying our model explains the effects of the out-of-order execution of reads — while executing programs in order — by recording the full write propagation history and allowing reading from older writes, not just the last same-address write [31]. The model state $\langle \vec{T}, M\rangle$ comprises the thread pool $\vec{T}$ and the memory $M$, where $\vec{T}$ maps each thread identifier *tid* to a statement (of type St), and a thread state, consisting of a register state and more components that we introduce as we proceed. We call a *thread* the pair of a statement and a thread state. We do not model dynamic thread creation; hence, the model transitions do not change the domain of the thread pool. (For reference, §5 has the formal definition of types and rules.)

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

Consider the following Message Passing (MP) example test, with instruction names $a - e$, and comments added for presentation; to easily distinguish, values are written in blue, thread identifiers brown, and (later) timestamps green. In this test, Thread 1 writes 37 to memory location $x$, and, after a strong dmb.sy barrier, writes 42 to $y$; Thread 2 reads $y$ and then $x$. To focus on the out-of-order execution of loads, we inserted the barrier $b$ between $a$ and $c$ to prevent their reordering. The execution of interest here is that where Thread 2 reads $y = 42$, and then the initial value $x = 0$. This is allowed in ARMv8/RISC-V because the (independent) loads on Thread 2 are allowed to execute out of order.

```
(a) store [x] 37; ‖ (d) r1 := load [y]; // 42
(b) dmb.sy;       ‖ (e) r2 := load [x] // 0
(c) store [y] 42  ‖
```

$r_1 = 42 \land r_2 = 0$ allowed

In our model, executing $a$, $b$, $c$ leads to the following transitions ($b$ does not change memory):

$$\langle \vec{T}, []\rangle \overset{(a)}{\Rightarrow} \langle \vec{T}', [\langle x := 37\rangle_1]\rangle \overset{(b)}{\Rightarrow} \langle \vec{T}'', [\langle x := 37\rangle_1]\rangle \overset{(c)}{\Rightarrow} \langle \vec{T}''', [\langle x := 37\rangle_1; \langle y := 42\rangle_1]\rangle$$

Now $d$ can read $y = 42$. Then, since loads can read not only from the last same-address write but also older writes in memory or the initial state, $e$ can read the initial $x = 0$.

***Views*** Memory barriers restore stronger ordering. Placing a dmb.sy barrier between the loads of Thread 2 orders them and prevents the behaviour where the load of $x$ reads 0 after $d$ reads 42. Our model handles this ordering using *views*:

**R1** A timestamp $t \in \mathbb{T} = \mathbb{N}$ is a natural number index of a write in the message history or 0, where list indices for memory start from 1 and timestamp 0 indicates the initial writes. A view $v \in \mathbb{V} = \mathbb{T}$ is simply a timestamp, indicating that the write at position $v$ and its predecessors in the message history have been "seen".

Before executing a load or store $i$, its *pre-view* is computed. The pre-view captures the dependencies and ordering requirements constraining the execution of $i$. For loads this constrains the values it can read from, for a store (in the later promising semantics) how "early" its write can be promised.

**R2** A view constrains loads: a thread can read from the most recent and older writes, but no older than the view allows — it must not read from writes overwritten by newer "seen" same-address writes.

After executing $i$, its *post-view* is computed. It captures the constraints $i$ imposes on instructions ordered with $i$.

**R3** The post-view of a load is the maximum of its pre-view and the *read-view*. In the examples we consider first, the read-view is simply the timestamp of the write the load reads from; we later refine this to handle *forwarding*. The post-view of a store is the timestamp of its write message (which is always strictly greater than its pre-view).

We gradually introduce how the pre-view of loads and stores is computed.

***Memory barriers*** Returning to the example, we model the effects of memory barriers using views:

**R4** Each thread state maintains views $\mathsf{v}_{\mathsf{rOld}}, \mathsf{v}_{\mathsf{wOld}}, \mathsf{v}_{\mathsf{rNew}}, \mathsf{v}_{\mathsf{wNew}} : \mathbb{V}$. Initially, all views are 0.

**R5** $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$, respectively, are the maximal post-view of all loads and stores executed so far by the thread.

**R6** $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$, respectively, contribute to the pre-view of all future loads and stores.

**R7** dmb.sy updates both $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to the maximum of $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$: all future loads and stores program-order-after the barrier are constrained by the post-views of those program-order-before the barrier.

Intuitively, dmb.sy orders loads and stores before it with those after it; it is the strongest form of barrier. Other barriers update these two views in a similar but weaker way.

```
(a) store [x] 37; ‖ (d) r1 := load [y]; // 42
(b) dmb.sy;       ‖ (e) dmb.sy;
(c) store [y] 42  ‖ (f) r2 := load [x] // 0
```

$r_1 = 42 \land r_2 = 0$ forbidden

In the example, after executing $a, b, c$, the memory is $[1\colon \langle x := 37\rangle_1; 2\colon \langle y := 42\rangle_1]$, with timestamps 1 and 2 shown explicitly for presentation. Now, if Thread 2 reads 42, $d$'s post-view is 2. This is recorded in $\mathsf{v_{rOld}}$. Executing $e$ includes $\mathsf{v_{rOld}}$ into $\mathsf{v_{rNew}}$ and $\mathsf{v_{wNew}}$: (just showing Thread 2)

$$\begin{aligned}\langle \mathsf{v_{rOld}} = 0, \mathsf{v_{rNew}} = 0, \dots\rangle &\overset{(d)}{\Rightarrow} \langle \mathsf{v_{rOld}} = 2, \mathsf{v_{rNew}} = 0, \dots\rangle\\ &\overset{(e)}{\Rightarrow} \langle \mathsf{v_{rOld}} = 2, \mathsf{v_{rNew}} = 2, \dots\rangle\end{aligned}$$

When executing $f$ in the resulting state, $f$ is constrained by a pre-view that includes $\mathsf{v_{rNew}} = 2$. Since $\langle x := 37\rangle_1$ is seen with view 2, $f$ must not read from a write older than that.

While `dmb.sy` provides strong ordering, it comes at a performance cost. Concurrent ARM programs also rely on ordering resulting from dataflow dependencies.

***Address dependencies*** The next example replaces the `dmb.sy` between the loads of Thread 2 with a *syntactic* address dependency from the first load ($d$) to the second ($e$). Register $r_1$ holding the return value of $d$ is used to compute the address "x + (r1 − r1)" of load $e$, which is enough to order $d$ before $e$, even though the value does not depend on what $d$ read. Similarly to the `dmb.sy`, the ordering from the syntactic dependency means that if $d$ reads 42, then $e$ must read 37.

```
(a) store [x] 37; ‖ (d) r1 := load [y]; // 42
(b) dmb.sy;       ‖ (e) r2 := load [x + (r1 − r1)] // 0
(c) store [y] 42  ‖
```

$r_1 = 42 \wedge r_2 = 0$ forbidden

Our model accounts for this using *register views*:

**R8** The register state $\mathsf{regs} : \mathsf{Reg} \to (\mathsf{Val} \times \mathbb{V})$ of a thread maps each register of the thread not only to a value, but also to an associated view (of type $\mathbb{V}$). We write $v@v$ for a value-view pair.

**R9** When an instruction writes a register it also updates this view, to specify which writes have to have been seen in order to produce the value. For any arithmetic instruction, the view of the output register is the maximum of the views of its input registers; for a load it is the post-view (the maximum of its pre-view and read view).

**R10** Finally, the pre-view of a load or store is the maximal view of its input registers (for loads the registers in the "address expression", for stores also that of the data) and the $\mathsf{v_{rNew}}$ or $\mathsf{v_{wNew}}$ view, respectively.

This will later be refined to handle more dependencies and *weaker barriers*, *release/acquire*, and *exclusives*.

Assuming the previous order $a, b, c$, when $d$ reads $y = 42$, Thread 2 executes as follows:

$$\begin{aligned} &\langle \mathsf{regs} = \{r_1 \mapsto 0@0, \dots\}, \mathsf{v_{rOld}} = 0, \mathsf{v_{rNew}} = 0\rangle\\ \overset{(d)}{\Rightarrow} &\langle \mathsf{regs} = \{r_1 \mapsto 42@2, \dots\}, \mathsf{v_{rOld}} = 2, \mathsf{v_{rNew}} = 0\rangle\end{aligned}$$

Now, while $\mathsf{v_{rNew}} = 0$, the pre-view of $e$ is 2, because $r_1$ is one of its input registers. Therefore $e$ is constrained by view 2, and thus cannot read the initial value $x = 0$.

***Coherence*** Accessing the same location multiple times also induces constraints. Consider the following example, which adds a later, independent, load $f$ to $x$ to Thread 2. While $f$ is not ordered with $d$, the execution where $d$ reads $y = 42$, $e$ reads $x = 37$ and $f$ reads $x = 0$ is forbidden, since it violates the principle of *coherence*: $a$ is ordered after the implicit initial $x = 0$ in memory. So if $e$ has read $x = 37$, the program-order-later $f$ must not read the coherence-superseded $x = 0$.

```
(a) store [x] 37; ‖ (d) r1 := load [y]; // 42
(b) dmb.sy;       ‖ (e) r2 := load [x + (r1 − r1)]; // 37
(c) store [y] 42  ‖ (f) r3 := load [x] // 0
```

$r_1 = 42 \wedge r_2 = 37 \wedge r_3 = 0$ forbidden

To account for the architectural coherence requirements:

**R11** Each thread state maintains the coherence view $\mathsf{coh} : \mathsf{Loc} \to \mathbb{V}$. It maps a location $x$ to the maximal post-view of all loads and stores on $x$ executed so far by that thread.

**R12** A load or store on $x$ is constrained not only by its pre-view, but also the coherence view $\mathsf{coh}(x)$.

Since $d$ reads $y = 42$ at timestamp 2, the register view of $r_1$ is 2, and so is the post-view of $e$. Thus, after $e$, the thread state is $\langle \mathsf{coh} = \{x \mapsto 2, \dots\}, \dots\rangle$. Then, although the pre-view of $f$ is 0, $f$ is also constrained by $\mathsf{coh}(x) = 2$, and thus cannot read the initial $x = 0$.

***Store forwarding*** However, while a load has to read from a write respecting coherence order, it does not have to effectively happen in order. Consider the next example, where Thread 1 is unchanged, but where Thread 2 now contains an earlier read $d$ from $y$, followed by a write $e$ of 51 to $y$, before the basic block of a read $f$ from $y$ followed by a read $g$ from $x$ with an address dependency on $f$. Assume $d$ reads $y = 42$, and $f$ reads $y = 51$. While $g$ is ordered after $f$ by the address dependency, $g$ is still allowed to read the initial $x = 0$: a load can "finish" before a same-thread store it reads from by *forwarding* when address and data of the store are determined, and so $f$ can execute and resolve $g$'s dependency before $e$ and even $d$.

```
(a) store [x] 37; ‖ (d) r0 := load [y]; // 42
(b) dmb.sy;       ‖ (e) store [y] 51;
(c) store [y] 42  ‖ (f) r1 := load [y]; // 51
                  ‖ (g) r2 := load [x + (r1 − r1)] // 0
```

$r_0 = 42 \wedge r_1 = 51 \wedge r_2 = 0$ allowed

For $d$ to read $y = 42$, our model must execute in the order $a, b, c, d$ and then $e$, leading to memory $[1\colon \langle x := 37\rangle_1; 2\colon \langle y := 42\rangle_1; 3\colon \langle y := 51\rangle_2]$. If $f$ now were to read $y = 51$ at timestamp 3 its post-view would become 3, and so would the view of register $r_1$; this would not allow $g$ to read the initial $x = 0$, since it would be constrained by pre-view 3 due to $r_1$. To allow this behaviour, each thread state records information about the thread's own writes, and the definition of the read-view specially handles the case in which a load reads from a write by its own thread, to allow it to obtain a smaller post-view than the write's timestamp:

**R13** Each thread state has a *forward bank* $\mathsf{fwdb} : \mathsf{Loc} \to \langle \mathsf{time} : \mathbb{T}, \mathsf{view} : \mathbb{V}, \mathsf{xcl} : \mathbb{B}\rangle$ holding for each location $x$ a record about the last write to $x$ propagated by the thread:

**R14** Whenever a thread executes a store to $x$ it updates $\mathsf{fwdb}(x)$ to record the timestamp of the write (time), the maximal view of the store's input registers (view), and whether it was a write exclusive (xcl). I.e. view captures its address and data dependencies.

**R15** Initially, $\mathsf{fwdb}(x)$ is $\langle \mathsf{time} = 0, \mathsf{view} = 0, \mathsf{xcl} = \mathtt{false}\rangle$ for any location $x$.

**R16** The read-view of a load to some location $x$ is refined as follows: if the read message's timestamp equals $\mathsf{fwdb}(x).\mathsf{time}$ (*i.e.* the load reads the last write at $x$ by its thread), its read-view is the associated forward view $\mathsf{fwdb}(x).\mathsf{view}$. Otherwise it is the read message's timestamp, as before.

Since the post-view of a load includes the read-view, the latter means that when reading by forwarding the post-view contains the address and data dependencies of the write instead of the write's timestamp. The xcl is only for exclusive instructions (§A.2).

In the example above, $d$ reads $y = 42$ at timestamp $2$, updating $\mathsf{coh}(y)$ to $2$; $e$ writes $y = 51$ at timestamp $3$, updating $\mathsf{coh}(y)$ to $3$ and $\mathsf{fwdb}(y)$ to $\langle \mathsf{time} = 3, \mathsf{view} = 0, \mathsf{xcl} = \mathtt{false}\rangle$, since $e$ has no input register; $f$ reads $y = 51$ at timestamp $3$, with pre-view $0$, read-view $0$ and post-view $0$, since the forward view of the write $y = 51$ is $\mathsf{fwdb}(y).\mathsf{view} = 0$, thereby setting the view of $r_1$ to $0$; finally, $g$ can read the initial $x = 0$ with pre-view $0$, since its sole input register, $r_1$, has view $0$.

It is important to note that, as seen in this example, in general the coherence view $\mathsf{coh}(x)$ on a location $x$ is never merged into any other views such as pre-views, post-views and register-views, so that its effect is limited to loads and stores on location $x$ only.

## 4.2 Promising semantics

In ARMv8/RISC-V, stores can be executed out of order, too. Promising-ARM/RISC-V models such behaviours by adding the notion of *promises* on top of the view semantics presented so far. As a motivating example, consider the next program. Here Thread 1 reads from $x$, and writes the value it reads to $y$; Thread 2 reads from $y$, and writes 42 to $x$. So far, we have not introduced any mechanism that would allow both $a$ and $c$ to read values different from 0: at least one of $a$ and $c$ would have to have executed first in the initial memory, and therefore would have to read 0.

```
(a) r1 := load [x]; // 42 ║ (c) r2 := load [y]; // 42
(b) store [y] r1          ║ (d) store [x] 42
```

$r_1 = r_2 = 42$ allowed

However, since $d$ is independent of $c$, in ARMv8/RISC-V it is allowed to execute early, and so both $a$ and $c$ can read 42, which corresponds to an execution order $d, a, b, c$.

***Promises*** To model out-of-order execution of writes, we add the notions of promise and fulfilment.

**R17** Each thread state maintains a set of timestamps $\mathsf{prom} : \mathsf{set}\ \mathbb{T}$, called its *promise set*, which records the timestamps of the outstanding promised writes of the thread.

**R18** A thread with ID *tid* is allowed to *promise* a write $x = v$, which appends $\langle x := v\rangle_{tid}$ to memory and adds the timestamp $t$ of the write message $\langle x := v\rangle_{tid}$ to prom, but does not otherwise change the thread state. As far as other threads are concerned, this write is no different from other writes in memory (prom is thread-local information).

**R19** The thread is required to *fulfil* this promise $x = v$ at timestamp $t$ at a later stage by executing a store instruction, removing the promise from prom. Specifically, the store must generate a write $x = v$ whose pre-view and coherence view $\mathsf{coh}(x)$ are strictly smaller than the promise timestamp $t$.

**R20** We split the execution of a write into a promise and its fulfilment. A normal write that is not executed early is accounted for by promising it just before a store fulfils it.

(We explain later exactly how executions with unfulfilled promises are prevented and how the executable model computes the possible promises in a given state.)

Note that the timestamp of a write is always bigger than its pre-view because it is appended at the end of memory with a fresh timestamp and immediately fulfilled.

The pre-view of a store essentially constrains promises by constraining the fulfilment: a promise cannot be made "too early", because it cannot be fulfilled if its timestamp is not strictly larger than its pre-view. With only these rules added to the underlying view semantics, in what follows, we show how promises capture the out-of-order execution of stores.

***Out-of-order execution of writes*** The behaviour in the previous example is explained as follows. Thread 2 first promises write $x = 42$ at timestamp 1, resulting in promise set $\mathsf{prom} = \{1\}$ and memory $[1{:}\ \langle x := 42\rangle_2]$. Now, on Thread 1 $a$ can read $x = 42$ and write $y = 42$ (by a normal write), resulting in memory $[1{:}\ \langle x := 42\rangle_2, 2{:}\ \langle y := 42\rangle_1]$. Then, $c$ can read $y = 42$, and $d$ can fulfil the promise $x = 42$ at timestamp 1, yielding $\mathsf{prom} = \{\}$; $d$'s pre-view and $\mathsf{coh}(x)$ are 0, strictly smaller than the promise timestamp 1, as required.

***Memory barriers*** Placing a barrier on Thread 2 prevents the out-of-order execution of writes.

```
(a) r1 := load [x]; // 42 ║ (c) r2 := load [y]; // 42
(b) store [y] r1          ║ (d) dmb.sy;
                          ║ (e) store [x] 42
```

$r_1 = r_2 = 42$ forbidden

The model handles this using views. As before, consider the state after Thread 2 promised $x = 42$ and Thread 1 executed $a, b$, resulting in memory $[1{:}\ \langle x := 42\rangle_2, 2{:}\ \langle y := 42\rangle_1]$. Here, $c$ is not allowed to read $y = 42$, because Thread 2 would not be able to fulfil the promise at timestamp 1. Suppose $c$ does read $y = 42$ at timestamp 2. Then Thread 2 has $\mathsf{v}_{\mathsf{rOld}} = 2$ after $c$ and $\mathsf{v}_{\mathsf{wNew}} = 2$ by `dmb.sy` after $d$. Then the pre-view of $e$ is 2 due to $\mathsf{v}_{\mathsf{wNew}}$, which is not smaller than the promise timestamp 1. If, instead, $c$ reads the initial $y = 0$, $e$ can fulfil the promise.

***Coherence*** Also, replacing $d$ by $\mathtt{r}_3 := \mathtt{load}\ [\mathtt{x} + (\mathtt{r}_2 - \mathtt{r}_2)]$ constrains $e$ by coherence and forbids the same behaviour. To illustrate, suppose we execute up to $c$ as before. Since $c$ reads $y = 42$ and thus $r_2$ holds 42@2, $d$ is constrained by pre-view 2 and must read $x = 42$ at timestamp 1. Now although $r_2$ and $r_3$ are not used by $e$, $e$ still cannot fulfil its promise of $x = 42$ at timestamp 1 since $d$ updated $\mathsf{coh}(x)$ to its post-view 2 and $e$ is constrained by $\mathsf{coh}(x) = 2 \not< 1$.

***Address and data dependencies*** Replacing the barrier on Thread 2 by a dependency from the load to the store — whereby the address or data of the store depends on the result of the load — also prevents the behaviour. Similarly to before, consider the execution in which Thread 2 promises $x = 42$ at timestamp 1, $a$ reads $x = 42$, $b$ writes $y = 42$ at timestamp 2, and $c$ reads $y = 42$, thereby setting $r_2$'s register view to 2. Here $d$'s pre-view includes $r_2$'s register view 2, and so $d$ cannot fulfil the promise at timestamp 1. Changing $d$ to '$\mathtt{store}\ [\mathtt{x}]\ (42 + (\mathtt{r}_2 - \mathtt{r}_2))$' leads to the same behaviour.

```
(a) r1 := load [x]; // 42  ||  (c) r2 := load [y]; // 42
(b) store [y] r1;          ||  (d) store [x + (r2 − r2)] 42
```

$r_1 = r_2 = 42$ forbidden

***Control and address-po dependencies*** While in ARMv8 and RISC-V loads are allowed to execute speculatively past conditional branches, stores are not. Control dependencies order writes with respect to reads affecting the control flow. Similarly, stores wait for the address of all program-order-earlier memory accesses to be determined (*address-po* dependency). Placing a conditional branch depending on $c$'s return value before the write of $x$ also prevents promising it early: the behaviour in which both $a$ and $c$ read 42 is forbidden in the example below, due to the control dependency of $e$ on $c$.

```
(a) r1 := load [x]; // 42  ||  (c) r2 := load [y]; // 42
(b) store [y] r1           ||  (d) if ((r2 − r2) = 0)
                           ||  (e)     store [x] 42
```

$r_1 = r_2 = 42$ forbidden

To capture such dependencies we introduce the view $\mathsf{v}_{\mathsf{CAP}}$.

**R21** Each thread state has a view $\mathsf{v}_{\mathsf{CAP}} : \mathbb{V}$, initially set 0.

**R22** Whenever a thread executes a conditional branch, the maximal view of the branch's input registers is merged into $\mathsf{v}_{\mathsf{CAP}}$. Similarly, when a load or store is executed, the maximal view of the input registers used to compute the address is merged into $\mathsf{v}_{\mathsf{CAP}}$.

**R23** Finally, the pre-view of a store instruction is refined to include $\mathsf{v}_{\mathsf{CAP}}$ (*i.e.* the pre-view is the maximal view of the input registers and $\mathsf{v}_{\mathsf{wNew}}$ and $\mathsf{v}_{\mathsf{CAP}}$).

Assume again an execution in which $x = 42$ is promised at timestamp 1 by Thread 2, $a$ reads $x = 42$, $b$ writes $y = 42$ at timestamp 2, and $c$ reads $y = 42$, thereby setting $r_2$'s view to 2. Then $d$ merges $r_2$'s view (2) into $\mathsf{v}_{\mathsf{CAP}}$ since $r_2$ is used to compute the branch condition. In case $d$ is $\mathtt{store}\ [\mathtt{z} + (\mathtt{r}_2 - \mathtt{r}_2)]\ \mathtt{0}$, register $r_2$'s view is also merged into $\mathsf{v}_{\mathsf{CAP}}$ since $r_2$ is used to compute the address. Then $e$'s pre-view includes $\mathsf{v}_{\mathsf{CAP}} = 2$, and thus $e$ cannot fulfil the promise at timestamp 1. Replacing $d$ by an address-dependent load or store to an otherwise unused memory location $z$ (e.g. $\mathtt{store}\ [\mathtt{z} + (\mathtt{r}_2 - \mathtt{r}_2)]\ \mathtt{0}$) introduces the same ordering and also forbids the behaviour.

***Release/acquire, weaker barriers, load/store exclusives*** In addition to the full `dmb.sy` barriers ARMv8 and RISC-V also have a number of weaker barriers; both also have release/acquire instructions, so-called half-barriers. Moreover, ARMv8 and RISC-V have load/store exclusive instructions (called load reserve and store conditional in RISC-V) that provide inter-thread *atomicity guarantees*: when a load exclusive $r$ *pairs* with a program-order-later store exclusive $w$ from the same thread and to the same location, the architectures guarantee exclusive access to this location to the thread if the store exclusive succeeds; otherwise the store fails.

For space reasons we omit the details on these instructions in §5. The instructions are natural extensions to the model and the supplementary material (§A) contains the full model definition including these; the Coq formalisation and proof, and the executable model both handle these instructions.

### 4.3 Certification

Our description so far has focussed on the thread steps and has assumed *consistent traces*, traces in which all promises are fulfilled. Indeed, the semantics given by traces of these thread steps, restricted to consistent traces, precisely models the legal behaviours of ARMv8/RISC-V (*i.e.*, is equivalent to the axiomatic model): (combining Theorems 6.1 and 6.2).

However, we have not yet discussed how the model ensures threads only take consistent steps (steps of consistent executions). Rather than merely discarding traces with unfulfilled promises at the end of the execution, directly preventing inconsistent thread steps is desirable for two reasons: (1) The model works incrementally in terms of thread-local conditions, thereby also improving interactive exploration. (2) It removes unnecessary non-determinism resulting from promises that eventually are unfulfilled, for executability and exhaustive exploration.

We now show how such inconsistent thread steps can be prevented. To this end, we first define what it means for a thread to *execute sequentially*. It means that the thread executes alone (no other threads executing) and every new promise is immediately followed by its fulfilment (effectively doing all writes in program-order). The model then prevents inconsistent thread steps using a simple, thread-local definition of *certification*, allowing any given thread step only if it leads to a *certified thread configuration*.

R24 A *thread configuration* $\langle T, M\rangle$, consisting of a thread state $T$ and memory state $M$ is *certified* if there exists a sequential execution from $\langle T, M\rangle$ to another thread configuration $\langle T', M'\rangle$ such that $T'$ has no outstanding promises.

Restricting thread steps to steps certified as above, is *sound*, not preventing any consistent executions (See Theorem 6.2). For RISC-V, this definition is also *precise*, preventing any inconsistent executions (Theorem 6.3).

In ARMv8, however, it is not precise. Whereas generally, syntactic dependencies create memory ordering, in ARMv8, dependencies from a store exclusive's status register write are an exception. The consequence in our model is the following: if a thread $t$ makes a promise that relies on some store exclusive $s$ succeeding, it can do this even before $s$ is propagated. If $s$ later turns out to conflict with the write of another thread and fails, $t$ cannot fulfil its promises. The certification definition above does not prevent this, for the same reasons that Flat suffers from such deadlocks [39]. The supplementary material (§C.2) details these issues and discusses a prototype extension of the model with locks and a more sophisticated certification that takes locking into account to prevent deadlocks.

The above definition provides a simple executable check for whether a thread configuration is certified. However, the executable tool of §7 has to be able to compute for any given thread state *which* promises should be allowed: which promises lead to such certified configurations.

For the sake of the executable model, we give an equivalent algorithmic definition, called `find_and_certify`, that we proved correct in Coq (Theorem 6.4). To enumerate which promises a thread *tid* in configuration $\langle T, M\rangle$ is allowed to do, the algorithm works as follows:

1. Enumerate all possible traces of *tid* executing *sequentially*: this thread executing alone under current memory. (For programs with infinite loops the user can bound the depth.)
2. Discard the traces in which the final state of *tid* has unfulfilled promises.
3. For any remaining trace $t$: any write done during $t$ is a legal promise step if its store's *pre-view and coherence-view (at its location)* are less than or equal to the maximal timestamp of the current memory $M$ (the memory before the start of the certification).

Section B of the supplementary material gives an example for this algorithm.

## 5 The model, formally

Fig. 2 defines the types, Fig. 3 the rules of the model. For simplicity, values and addresses are mathematical integers. Note that some of the types are not used by the rules shown here, but used by the full definition of §A of the supplementary material that handles weaker barriers, release/acquire, and load/store exclusive instructions. For the common instructions, ARM and RISC-V only differ in acquire and exclusive instructions. The description cross-references the §4 rules.

***Auxiliaries*** The expressions interpretation function (second and third line) takes an expression and a register state $m$, and returns the expression's value and view. Constants have view 0; registers are looked up in $m$; the view for an arithmetic expression merges the arguments' views (**R9**). read$(M, l, t)$ gives the result of reading location $l$ at timestamp $t$ in memory $M$: for $t = 0$ the initial value $v_{\text{init}}$, here 0; otherwise either the value of the message in $M$ at timestamp $t$ if its location is $l$ or *none*. read-view$(a, rk, f, t)$ returns either the timestamp $t$ of the read message or the forward view of the message $f$ in the forward bank (**R13** – **R16**). Now we define thread-local steps, which do not change memory.

***Thread-local steps*** $T, M \xrightarrow{[t]}_{tid} T'$

**FULFIL**. This transition is annotated with the timestamp $t$ of the promise that is being fulfilled. (Other thread-local steps do not have the timestamp annotation.) The definition starts with the pre-condition (from top to bottom). First evaluate address and data expressions. Since we assume writes always promise first and then fulfil, this step requires the write to have been promised. Rules **R10**, **R6**, **R21** describe the components contributing to the pre-view. The pre-view and coherence view have to be less than $t$ (**R19**); the post-view is the timestamp $t$ (**R3**). The post-condition removes the promise (**R19**) and updates the coherence view to include $t$ (**R11**), certain views (**R5**, **R22**); the forward bank (**R14**). O **READ** also starts with the pre-condition (from top to bottom). First evaluate the address $l$; in order to read $v$ it must be $v$ = read$(M, l, t)$ as described above. The pre-view calculation is described in **R10**, **R6**. The pre-view (**R2**) and the coherence view (**R12**) constrain the read. The post-view is defined in **R3**, **R16**. The post-condition updates the register with value and post-view (**R9**); coherence with post-view as in rule **R11**; and views as in **R5**, **R22**.

**DMB** is the rule for strong barriers (`dmb.sy` in ARMv8 and $\texttt{fence}_{RW,RW}$ in RISC-V). It updates $\mathsf{v}_{\text{rNew}}$ and $\mathsf{v}_{\text{wNew}}$, to include $\mathsf{v}_{\text{rOld}}$ and $\mathsf{v}_{\text{wOld}}$ according to the intuition given in **R5**, **R6**. The definition matches rules **R5** – **R7**. **REGISTER** A register assignment updates the register with the expressions and view from the evaluation of its expression (**R9**). **BRANCH:** The pre-condition evaluates the condition, branches as determined by this value, and updates $\mathsf{v}_{\text{CAP}}$ (**R22**). **SKIP, SEQ, and WHILE:** mostly as expected.

***Thread steps*** $\langle T, M\rangle \xrightarrow{[t]}_{tid} \langle T', M'\rangle$

**EXECUTE** lifts a thread-local step that does not change memory to a thread step. **PROMISE** allows promising any write message, appending this write to memory and recording its timestamp in prom. As thread-local steps, thread steps can

$$l \in \mathrm{Loc} \overset{def}{=} \mathrm{Val} \qquad v \in \mathrm{Val} \overset{def}{=} \mathbb{Z} \qquad tid \in \mathrm{TId} \overset{def}{=} \mathbb{N} \qquad t \in \mathbb{T} \overset{def}{=} \mathbb{N} \qquad \nu \in \mathbb{V} \overset{def}{=} \mathbb{T}$$

$$w \in \mathrm{Msg} \overset{def}{=} \langle \mathrm{loc}:\mathrm{Loc}; \mathrm{val}:\mathrm{Val}; \mathrm{tid}:\mathrm{TId}\rangle \qquad \langle x := v\rangle_{tid} \overset{def}{=} \langle \mathrm{loc}=x; \mathrm{val}=v; \mathrm{tid}=tid\rangle \qquad M \in \mathrm{Memory} \overset{def}{=} \mathrm{list}\ \mathrm{Msg}$$

$$ts \in \mathrm{TState} \overset{def}{=} \left\langle \begin{array}{l} \mathrm{prom} : \mathrm{set}\ \mathbb{T}; \quad \mathrm{regs} : \mathrm{Reg} \to \mathrm{Val} \times \mathbb{V}; \\ \mathrm{coh} : \mathrm{Loc} \to \mathbb{V}; \quad \mathrm{v_{rOld}}, \mathrm{v_{wOld}}, \mathrm{v_{rNew}}, \mathrm{v_{wNew}}, \mathrm{v_{CAP}}, \mathrm{v_{Rel}} : \mathbb{V}; \\ \mathrm{fwdb} : \mathrm{Loc} \to \langle \mathrm{time} : \mathbb{T}; \mathrm{view} : \mathbb{V}; \mathrm{xcl} : \mathbb{B}\rangle; \\ \mathrm{xclb} : \mathrm{option}\ \langle \mathrm{time} : \mathbb{T}; \mathrm{view} : \mathbb{V}\rangle \end{array} \right\rangle \qquad \begin{array}{r} T \in \mathrm{Thread} \overset{def}{=} \mathrm{St} \times \mathrm{TState} \\ \vec{T} \in \mathrm{TPool} \overset{def}{=} \mathrm{TId} \to \mathrm{Thread} \\ \langle \vec{T}, M\rangle \in \mathrm{Machine} \overset{def}{=} \mathrm{TPool} \times \mathrm{Memory} \end{array}$$

**Figure 2.** Types in the semantics

$$c\ ?\ v_1 : v_2 \overset{def}{=} \text{if } c \text{ then } v_1 \text{ else } v_2 \qquad c\ ?\ \nu \overset{def}{=} c\ ?\ \nu : 0 \qquad \nu_1 \sqcup \nu_2 \overset{def}{=} \max(\nu_1, \nu_2) \qquad v@\nu \overset{def}{=} \langle v, \nu\rangle : \mathrm{Val} \times \mathbb{V}$$

$$[\![(-)_1]\!]_{(-)_2} : \mathit{Expr} \to (\mathrm{Reg} \to \mathrm{Val} \times \mathbb{V}) \to \mathrm{Val} \times \mathbb{V}$$

$$[\![v]\!]_m \overset{def}{=} v@0 \qquad [\![r]\!]_m \overset{def}{=} m(r) \qquad [\![e_1 \ \mathsf{op}\ e_2]\!]_m \overset{def}{=} (v_1 \, [\![\mathsf{op}]\!]\, v_2)@(\nu_1 \sqcup \nu_2) \text{ with } [\![e_1]\!]_m = v_1@\nu_1, [\![e_2]\!]_m = v_2@\nu_2$$

$$\mathrm{read}(M, l, t) : \mathrm{option\ Val} \overset{def}{=} \text{if } t = 0 \text{ then } \mathrm{v_{init}} \text{ else (if } M(t).\mathrm{loc} = l \text{ then } M(t).\mathrm{val} \text{ else } \mathit{none})$$

$$\text{read-view}(a, rk, f, t) \overset{def}{=} (f.\mathrm{time} = t)\ ?\ f.\mathrm{view} : t$$

$$\boxed{T, M \xrightarrow{[t]}_{tid} T'}$$

(READ)

$$\frac{\begin{array}{l} l@\nu_{\mathrm{addr}} = [\![e]\!]_{ts.\mathrm{regs}} \quad \mathrm{read}(M, l, t) = v \quad \nu_{\mathrm{pre}} = \nu_{\mathrm{addr}} \sqcup ts.\mathrm{v_{rNew}} \\ \forall t'.\ t < t' \le (\nu_{\mathrm{pre}} \sqcup ts.\mathrm{coh}(l)) \implies M(t').\mathrm{loc} \ne l \\ \nu_{\mathrm{post}} = \nu_{\mathrm{pre}} \sqcup \text{read-view}(a, rk, ts.\mathrm{fwdb}(l), t) \\ ts' = ts\begin{bmatrix} \mathrm{regs}(r) \mapsto v@\nu_{\mathrm{post}}, & \mathrm{coh}(l) \mapsto ts.\mathrm{coh}(l) \sqcup \nu_{\mathrm{post}}, \\ \mathrm{v_{rOld}} \mapsto ts.\mathrm{v_{rOld}} \sqcup \nu_{\mathrm{post}}, & \mathrm{v_{CAP}} \mapsto ts.\mathrm{v_{CAP}} \sqcup \nu_{\mathrm{addr}} \end{bmatrix} \end{array}}{\langle r := \mathtt{load}_{\mathtt{false}, rk}\ [e], ts\rangle, M \to_{tid} \langle \mathtt{skip}, ts'\rangle}$$

(FULFIL)

$$\frac{\begin{array}{l} [\![e_1]\!]_{ts.\mathrm{regs}} = l@\nu_{\mathrm{addr}} \quad [\![e_2]\!]_{ts.\mathrm{regs}} = v@\nu_{\mathrm{data}} \quad t \in ts.\mathrm{prom} \\ M(t) = \langle l := v\rangle_{tid} \qquad \nu_{\mathrm{pre}} = \nu_{\mathrm{addr}} \sqcup \nu_{\mathrm{data}} \sqcup ts.\mathrm{v_{wNew}} \sqcup ts.\mathrm{v_{CAP}} \\ \nu_{\mathrm{pre}} \sqcup ts.\mathrm{coh}(l) < t \quad \nu_{\mathrm{post}} = t \\ ts' = ts\begin{bmatrix} \mathrm{prom} \mapsto ts.\mathrm{prom} \setminus \{t\}, & \mathrm{coh}(l) \mapsto ts.\mathrm{coh}(l) \sqcup \nu_{\mathrm{post}}, \\ \mathrm{v_{wOld}} \mapsto ts.\mathrm{v_{wOld}} \sqcup \nu_{\mathrm{post}}, & \mathrm{v_{CAP}} \mapsto ts.\mathrm{v_{CAP}} \sqcup \nu_{\mathrm{addr}}, \\ \multicolumn{2}{l}{\mathrm{fwdb}(l) \mapsto \langle \mathrm{time} = t; \mathrm{view} = \nu_{\mathrm{addr}} \sqcup \nu_{\mathrm{data}}; \mathrm{xcl} = \mathtt{false}\rangle} \end{bmatrix} \end{array}}{\langle r_{\mathrm{succ}} := \mathtt{store}_{\mathtt{false}, wk}\ [e_1]\ e_2, ts\rangle, M \xrightarrow{t}_{tid} \langle \mathtt{skip}, ts'\rangle}$$

(DMB)

$$\frac{\begin{array}{l} \nu = ts.\mathrm{v_{rOld}} \sqcup ts.\mathrm{v_{wOld}} \\ ts' = ts\begin{bmatrix} \mathrm{v_{rNew}} \mapsto ts.\mathrm{v_{rNew}} \sqcup \nu, \\ \mathrm{v_{wNew}} \mapsto ts.\mathrm{v_{wNew}} \sqcup \nu \end{bmatrix} \end{array}}{\langle \mathtt{fence}_{K_1, K_2}, ts\rangle, M \to_{tid} \langle \mathtt{skip}, ts'\rangle}$$

(REGISTER)

$$\frac{ts' = ts\left[\mathrm{regs}(r) \mapsto [\![e]\!]_{ts.\mathrm{regs}}\right]}{\langle r := e, ts\rangle, M \to_{tid} \langle \mathtt{skip}, ts'\rangle}$$

(BRANCH)

$$\frac{\begin{array}{c} [\![e]\!]_{ts.\mathrm{regs}} = v@\nu \\ ts' = ts[\mathrm{v_{CAP}} \mapsto ts.\mathrm{v_{CAP}} \sqcup \nu] \end{array}}{\langle \mathtt{if}\ (e)\ s_1\ s_2, ts\rangle, M \to_{tid} \langle v \ne 0\ ?\ s_1 : s_2, ts'\rangle}$$

(SKIP)

$$\frac{}{\langle \mathtt{skip}; s, ts\rangle, M \to_{tid} \langle s, ts\rangle}$$

(SEQ)

$$\frac{\langle s_1, ts\rangle, M \to_{tid} \langle s_1', ts'\rangle}{\langle s_1; s_2, ts\rangle, M \to_{tid} \langle s_1'; s_2, ts'\rangle}$$

(WHILE)

$$\frac{s' = \mathtt{if}\ (e)\ (s; \mathtt{while}\ (e)\ s)\ \mathtt{skip}}{\langle \mathtt{while}\ (e)\ s, ts\rangle, M \to_{tid} \langle s', ts\rangle}$$

$$\boxed{\langle T, M\rangle \xrightarrow{[t]}_{tid} \langle T', M'\rangle}$$

(EXECUTE)

$$\frac{T, M \to_{tid} T'}{\langle T, M\rangle \to_{tid} \langle T', M\rangle}$$

(PROMISE)

$$\frac{w.\mathrm{tid} = tid \quad t = |M| + 1 \quad ts' = ts[\mathrm{prom} \mapsto ts.\mathrm{prom} \cup \{t\}]}{\langle\langle s, ts\rangle, M\rangle \xrightarrow{t}_{tid} \langle\langle s, ts'\rangle, M +\!\!+ [w]\rangle}$$

$$\boxed{\langle \vec{T}, M\rangle \to \langle \vec{T}', M'\rangle}$$

(MACHINE-STEP)

$$\frac{\langle \vec{T}[tid], M\rangle \to_{tid} \langle T', M'\rangle \qquad \langle T', M'\rangle \text{ certified}}{\langle \vec{T}, M\rangle \to \langle \vec{T}\,[tid \mapsto T']\,, M'\rangle}$$

$$\langle T, M\rangle \text{ certified} \overset{def}{=} \exists T', M'.\ \langle T, M\rangle \xrightarrow{\mathrm{seq}}{}^{*}_{tid} \langle T', M'\rangle \wedge T'.\mathrm{prom} = \{\}$$

$$\boxed{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{tid} \langle T', M'\rangle}$$

(SEQ-EXEC)

$$\frac{T, M \to_{tid} T'}{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{tid} \langle T', M\rangle}$$

(SEQ-WRITE)

$$\frac{\begin{array}{c} \langle T, M\rangle \xrightarrow{t}_{tid} \langle T', M'\rangle \\ T', M' \xrightarrow{t}_{tid} T'' \end{array}}{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{tid} \langle T'', M'\rangle}$$

**Figure 3.** Formal definition of the model, including Thread-local steps, thread steps, and machine steps

be annotated with a timestamp $t$; this is used for the steps for promising and for promise fulfilment. While thread steps allow unconstrained promises, machine steps only allow certified promises. Note that "normal writes" are modelled as promises immediately followed by fulfilment.

***Machine steps*** $\langle \vec{T}, M\rangle \to \langle \vec{T}', M'\rangle$
Lifts *certified* thread steps (**R24**).

## 6 Proof

In Coq, we formally prove equivalence to the ARMv8 and RISC-V axiomatic models, deadlock freedom for RISC-V, and correctness of `find_and_certify`. The former currently assumes known simplifications of the axiomatic models, to unify ARMv8 and RISC-V for the Coq proof. We call the unified model Axiomatic (see supplementary material §D).

**Theorem 6.1.** *For a program p, $\vec{R}$ is a final register state of a legal candidate execution of p in* Axiomatic *if and only if it is that of a valid execution of p in* Promising*-ARM/RISC-V.*

**Theorem 6.2.** *Moreover,* Promising*-ARM/RISC-V is equivalent to* Promising*-ARM/RISC-V without certification.*

**Theorem 6.3** (Deadlock freedom)**.** *For any machine state in* Promising*-RISC-V where every thread state of it is certified, there exists an execution to a machine state with no promises.*

**Theorem 6.4** (Correctness of `find_and_certify`)**.** *Assume the thread configuration $\langle T, M\rangle$ is certified, and promising p leads to $\langle T', M'\rangle$. Then $\langle T', M'\rangle$ is certified if and only if $p \in$ `find_and_certify`$\langle T, M\rangle$.*

## 7 Executable tool

Exhaustively enumerating all outcomes of a concurrent program is combinatorially challenging. The main optimisation the executable tool incorporates is based on the following key property of the model, proved in Coq.

**Theorem 7.1.** *For every* Promising *trace tr, there exists a trace tr′ with same final state such that tr′ can be split into a sequence of promise transitions followed by only non-promise transitions.*

The model uses this property as follows: (1) The model starts in "promise-mode", exploring all possible interleavings of only promise transitions. (2) Once reaching a state in which it is possible to continue executing with no further promises it can enter "non-promise-mode". Here no promise transitions are enabled. Since the memory is now fixed, the remaining execution of the threads is completely independent. Hence, the model can compute all outcomes of a given program by first enumerating all "final memories", interleaving only promise transitions, and then exploring which final thread states are possible under this memory without interleaving any reads, greatly reducing the non-determinism.

Another optimisation we implement allows the user to supply information about which memory locations are shared between threads and which are purely thread-local. The model then treats accesses to non-shared locations as register reads/writes, again reducing the interleavings. In order to prevent user errors, the model could check that non-shared locations are accessed by one thread only. (The model currently does not, but it is an easy addition.) Moreover, we plan to extend the model to automatically derive this information, using existing mechanisms for promise certification.

***Executable model*** The executable model closely follows the Coq definitions where possible, but differs from it in two ways. Firstly, the executable model does not use OCaml code automatically produced by Coq, since interfacing with the existing rmem and Sail infrastructure would be difficult. Secondly, while the Coq model formalises the small imperative language, the executable model integrates definitions for user-mode ARMv8 and RISC-V instructions, meaning it needs logic for computing the views of loads and stores of the ISA definitions [13, 22]. (Like Flat, our Sail model does not yet include ARM's weaker load acquire LDAPR introduced in ARMv8.3. We do cover its concurrency behaviour in the Coq model.) We ensured our model also experimentally agrees with the axiomatic models on suites of around 6,500 litmus tests for ARMv8 and 7,000 for RISC-V (150 RISC-V tests we cannot run because they have AMO instructions).

## 8 Tool use and evaluation

To evaluate the exploration tool, we test several standard datastructure and lock implementations, for ARMv8. We first demonstrate the use of the tool for one of the examples and then give a summary of all results and run times. The code of all examples is in the supplementary material.

***Example use case*** We consider the Michael & Scott queue, a queue allowing concurrent enqueueing and dequeueing [36] that we implement in C++. To the code for the actual data structure we add test code: three threads running in parallel, each doing enqueue and dequeue operations while recording which data they enqueue and dequeue. The source altogether has 215 lines of C++ code, which we compile using a standard GCC 6.3.0 cross compiler with -O3, obtaining 472 lines of assembly code. We then run a script that maps the assembly into the *litmus* format used by herd and rmem: our tool does not support dynamic thread creation yet; the litmus format allows writing the code directly as a parallel thread composition. Another limitation is that our tool does not yet support dynamic memory allocation, which we "fake" here with a very naive `malloc` as part of the source code (see code). We are planning to add dynamic thread creation and memory allocation, which just requires additional engineering.

Running our model exhaustively, integrated into rmem, outputs the list of possible final outcomes and allows us to check whether the code has behaved correctly. For a version in which Thread 1 enqueues once and Threads 2 and 3 each try to dequeue once the tool finishes in 9 seconds, reporting no incorrect state. We also try other versions (see Table 2), finding no incorrect states.

When implementing the queue, we initially chose conservative acquire/release ordering for all but two C++ atomic accesses. Investigating the assembly shows the assembly code under ARM's memory model provides stronger guarantees than required. We experiment with relaxing some of the atomics and run the tool again. After roughly two minutes it reports an incorrect state: one in which the enqeue operation succeeded, the queue is empty, but neither of the dequeueing threads has read the data. The tool also provides a witnessing trace, that allows interactively stepping through the execution for debugging. (Table 2 shows numbers for an optional optimisation, but for which the model currently does not produce fine-grained interactive traces; this just requires additional engineering.)

In this case, we see the following behaviour. The first transitions are all promising transitions by the enqueueing thread, Thread 1, partly for initialising the queue. The queue is implemented as a linked list. Each queue element is a `struct` with two fields, `data` and `next`, a pointer to the next element. The queue contains an initial dummy element, `Init`, and fields `head` and `tail` pointing to start and end of the queue. Thread 1 sets these up: it writes `Init` with `data = 0` and `next = null`, and points `head` and `tail` to `Init`. Thread 1 also does the first enqueue operation, of an element $e$. The steps are, in program order:

1. create the new element $e$, with `next = null` and `data` (in this case) set to 1,
2. and enqueue $e$: find the tail of the queue, here `Init`, and set its `next` field to point to $e$.

In this trace, the problematic behaviour occurs in the third transition: Thread 1 writes `Init`'s `next` field to point to $e$, before having written $e$'s `data` (executing step 2 before step 1). Thus, later, when Thread 2 dequeues $e$, it can read the initial value 0 for $e$'s `data`, even though 1 would be correct.

Here the bug is easily fixed, by making the above two steps execute in program-order. We fix the issue by making the write of the `next` field in step 2 a release write, preventing "publishing" the new element before its data has been written. The resulting code is unsound in C++ but still sound in ARM.

***Tested examples and results*** We give an overview of the tests we ran in Table 1. We test, from simple to complex: three different spinlock variants, implemented in assembly (SLA), C++ (SLC), and Rust (SLR), where Linux-Spinlock (SLA) is an example taken from a Linux kernel spinlock implementation [25, 33], which was also used by Pulte et al. [39] for demonstrating Flat[2] (their other test, `spin_unlock_wait`, requires mixed-size support); single-producer-single-consumer (PCS) and single-producer-multiple-consumers (PCM) circular queues; a ticket lock (TL); Treiber stack [17], separately implemented in C++ (STC) and Rust (STR); Chase-Lev dequeue (DQ) [28, 32]; and the aforementioned variants of the Michael & Scott queue (QU). In addition, for the last four examples we also try versions optimised for ARMv8. Table 2 shows selected results, more in the supplementary material (§E). All programs in C++ and Rust are compiled with GCC 6.3.0 and RUSTC 1.30 with optimization level 3. All numbers are from a standard desktop machine, Ubuntu 16.04 Intel Core i7-7700 at 3.60GHz, 8GB memory.

The names mean the following. For the spinlock tests: spinlock-$n$ means $n$ loop unrollings on all threads; PCM-$n$-$n$-$n$: Thread 1 producing $n$ times, Threads 2 and 3 consuming $n$ times; PCS-$n$-$n$ the same for Thread 1 producing, Thread 2 consuming; TL-$n$: threads spin $n$ times to acquire lock; STC/R-*abc*-*def*-*ghi*: Thread 1 pushing $a$ times, popping $b$ times, and again pushing $c$ times, and analogously for Thread 2 with *def* and Thread 3 with *ghi*; DQ/(opt)-*abc*-*d*-*e*: Thread 1 pushes $a$ times, pops $b$ times, and pushes $c$ times, Thread 2 steals $d$ times, and Thread 3 steals $e$ times; QU/(opt)-*abc*-*def*-*ghi*: Thread 1 enqueues $a$ times, dequeues $b$ times, and enqueues $c$ times again, analogously with *def* for Thread 2 and *ghi* for Thread 3.

| *Test* | *Lang* | *LOC* | *Ts* | *Test* | *Lang* | *LOC* | *Ts* |
|---|---|---|---|---|---|---|---|
| SLA | ARMv8 | 44 | 2 | TL | C++ | 120 | 3 |
| SLC | C++ | 51 | 3 | STC | C++ | 366 | 3 |
| SLR | Rust | 84 | 3 | STR | Rust | 393 | 3 |
| PCS | C++ | 69 | 2 | DQ | C++ | 247 | 3 |
| PCM | C++ | 130 | 3 | QU | C++ | 473 | 3 |

**Table 1.** *LOC* = assembly lines, *Ts* = number of threads

| *Test* | *Promising* | *Flat* |
|---|---|---|
| SLA-7 | 0.61 | 9108.53 |
| SLC-3 | 6.58 | 1472.74 |
| SLR-3 | 4.88 | 52.52 |
| PCS-3-3 | 1.36 | 249.26 |
| PCM-3-3-3 | 71.12 | ooT |
| TL/(opt)-3 | 18.08 / 20.13 | ooT / ooT |
| STC/(opt)-100-010-010 | 0.42 / 0.42 | 2144.52 / 5943.50 |
| STC/(opt)-100-100-010 | 8.70 / 8.70 | ooT / ooT |
| STC/(opt)-210-011-000 | 615.41 / 637.98 | ooT / ooT |
| STR-100-010-010 | 0.39 | 77.21 |
| STR-100-100-010 | 7.30 | 8940.03 |
| STR-210-011-000 | 522.19 | ooT |
| DQ/(opt)-100-1-0 | 0.30 / 0.30 | 2.93 / 2.97 |
| DQ/(opt)-110-1-0 | 0.44 / 0.44 | 1042.88 / 1114.39 |
| DQ/(opt)-211-2-1 | 28.55 / 111.54 | ooT / ooT |
| QU/(opt)-100-000-000 | 1.34 / 2.95 | 2983.11 / ooT |
| QU/(opt)-100-010-000 | 2.55 / 5.66 | ooT / ooT |
| QU/(opt)-110-100-010 | 2108.12 / ooT | ooT / ooT |

**Table 2.** Run times in seconds. ooT = more than four hours.

We tried running the examples on herd, but all but the spin and ticket locks (SLC,SLR,TL) require instructions unsupported by herd. For SLC and TL, we ran the tests on herd:

- SLC-1: 14.72 sec, (Promising: 3.21 sec)
- SLC-2: stack overflow in 123.51 sec, (Promising: 4.69 sec)
- TL-1: 31.04 sec, (Promising: 10.16 sec)
- TL-2: 2370.23 sec, (Promising: 13.72 sec)

The results show that the Promising model scales much better than Flat for the tested examples. However, we believe, we may further improve performance significantly over the current results by studying existing model checking techniques. For instance, for certain litmus tests Promising does not perform as well as Flat: tests with a large number of writes whose interleaving does not matter. In such tests,

[2] We change mixed-size loads in the example to same-size and confirmed that performance is unaffected.

Promising explores all interleavings of the writes, and each interleaving leads to a different memory state, even if the order of some writes does not affect the possible final outcomes. Here we believe partial-order reduction techniques [21] can help improve performance.

Where data is available, the results also show the Promising model performs much better than herd. The comparison here is less clear: on the one hand herd does not accurately capture the full ISA behaviour, meaning it has to do less work during the exhaustive search; on the other hand herd is not designed primarily to be a fast model checker, and it is not clear how the axiomatic approach compares to ours in search complexity. Herd computes the set of allowed behaviours in two steps: first, herd enumerates the set of all possible *candidate executions*, concrete executions given by a program-order unfolding, write values for the reads, and relations capturing the concurrency behaviour satisfying only minimal well-formedness conditions; then, herd discards executions violating the model's axioms. A naive generation of the set of candidate executions is expensive. However, it may be possible to efficiently generate candidate executions using an approach similar to the promise-first enumeration described here, or adapt the model checker of Kokologiannakis et al. [30] based on axiomatic models to ARMv8/RISC-V.

## 9 Related work

***Models*** The reference axiomatic memory model, written in herd, and the Flat model have already been discussed. We build on extensive past research on ARMv8 and Power concurrency [6, 7, 10, 11, 18, 19, 22–24, 34, 35, 39–41]. The main inspiration for Promising-ARM is the promising semantics of Kang et al. for C/C++ 11 [29]. As described in the introduction, Promising-ARM uses the same concepts: views, promises, and certification, but leverages them in a different way: (1) Views consist of a single timestamp, and timestamps at different locations are comparable, reflecting multicopy atomicity. Messages do not carry views, and barriers work purely thread-locally. (2) In order to capture the semantic dependencies, Promising C11 uses *future-memory quantification* for certification, which makes exhaustive execution hard. Here, dependencies are purely syntactic, captured by associating timestamps to registers. (3) The key property of write-first execution (Theorem 7.1) does not hold for Promising C11. The $I^2E$ models [14, 43, 44] aim to provide simple models in which instructions execute in order and atomically. However, $I^2E$ does not allow load-store reordering, allowed in ARMv8/RISC-V.

***Model checking*** As demonstrated, our tool can be used for exhaustive state exploration for small instances of concurrency libraries for ARMv8/RISC-V. Since the tool is *sound and complete*, listing precisely the architecturally allowed behaviours, checking such small instances can already be a useful tool for checking the library code: providing confidence in the correctness without generating false positives.

There are a number of existing concurrency model checking tools, none of which, however, apply to ARMv8 or RISC-V. Alglave et al. [8] develop a model checking tool based on axiomatic models for some weaker models including RMO and Power, but not ARMv8, and it does not integrate a substantial ISA model. Abdulla et al. [3–5] describe efficient model-checking algorithms for hardware models (TSO, PSO, and Power), proved sound for Power. They do not handle ARMv8; for their Nidhugg tool, ARM support is called "partial", under-approximating the behaviours [1], and they do not handle a (Power or ARM) ISA model, but a simple calculus. CDSChecker [37] tackles model checking for a variant of C/C++ 11 that allows load buffering. Kokologiannakis et al. [30] develop an efficient algorithm for a variant of C/C++ 11 that forbids load buffering.

## 10 Conclusion and future work

We have presented an operational model for ARMv8 and RISC-V concurrency, in a different style from the existing ones. The model applies the concepts of promises and views from the Promising C11 semantics to hardware concurrency, to enable a simpler and more abstract operational model. Promising-ARM/RISC-V offers the attractive mental model of instructions executing incrementally, instantaneously, and mostly in program order, which we hope can make the subtle concurrency behaviour of ARMv8 and RISC-V more accessible to both researchers and programmers.

The model enables an executable tool for exhaustive checking and interactive debugging. The experimental results show significant performance improvements over the Flat and herd models in exhaustive search. In the future we plan more work on this — exploring improvements enabled by the model's abstractness and by applying known model checking techniques. We believe Promising-ARM/RISC-V can serve as a basis for other research on hardware concurrency, for formal reasoning about ARMv8/RISC-V concurrent software and for testing and analysis tools.

## Acknowledgments

We thank Peter Sewell for valuable feedback. We thank Shaked Flur for interesting discussions, and Viktor Vafeiadis and Ori Lahav for helpful feedback. This work was partly supported by the EPSRC Programme Grant *REMS: Rigorous Engineering for Mainstream Systems* (EP/K008528/1), an ARM iCASE award, and by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-53.

## References

[1] 2016. nidhugg. https://github.com/nidhugg/nidhugg.
[2] 2019. promising-arm-riscv. https://github.com/promising-arm-riscv.
[3] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017. Stateless

model checking for TSO and PSO. *Acta Inf.* 54, 8 (2017), 789–818. https://doi.org/10.1007/s00236-016-0275-0

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2017. Context-bounded analysis for POWER. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II.* 56–74. https://doi.org/10.1007/978-3-662-54580-5_4

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless model checking for POWER. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II.* 134–156. https://doi.org/10.1007/978-3-319-41540-6_8

[6] Allon Adir, Hagit Attiya, and Gil Shurek. 2003. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.* 14, 5 (2003), 502–515. https://doi.org/10.1109/TPDS.2003.1199067

[7] 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 the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009.* 13–24. https://doi.org/10.1145/1481839.1481842

[8] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

[9] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening small children and disconcerting grown-ups: concurrency in the linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018.* 405–418. https://doi.org/10.1145/3173162.3177156

[10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings.* 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

[11] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014.* 40. https://doi.org/10.1145/2594291.2594347

[12] ARM. 2018. *ARM Architecture Reference Manual.*

[13] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *PACMPL* 3, POPL (2019), 71:1–71:31. https://dl.acm.org/citation.cfm?id=3290384

[14] Arvind and Jan-Willem Maessen. 2006. Memory model = instruction reordering + store atomicity. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA.* 29–40. https://doi.org/10.1109/ISCA.2006.26

[15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

[16] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 55–66. https://doi.org/10.1145/1926385.1926394

[17] Thomas J. Watson IBM Research Center and R. K. Treiber. 1986. *Systems programming: coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center. https://books.google.co.uk/books?id=YQg3HAAACAAJ

[18] Nathan Chong and Samin Ishtiaq. 2008. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008.* 16–19. https://doi.org/10.1145/1353522.1353528

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

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

[21] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005.* 110–121. https://doi.org/10.1145/1040305.1040315

[22] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 608–621. https://doi.org/10.1145/2837614.2837615

[23] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 429–442. http://dl.acm.org/citation.cfm?id=3009839

[24] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015.* 635–646. https://doi.org/10.1145/2830772.2830775

[25] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. 2018. Documentation/memory-barriers.txt. https://www.kernel.org/doc/Documentation/memory-barriers.txt.

[26] Daniel Jackson. 2003. Alloy: A Logical Modelling Language. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings.* 1. https://doi.org/10.1007/3-540-44880-2_1

[27] Alan Jeffrey and James Riely. 2016. On thin air reads towards an event structures model of relaxed memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016.* 759–767. https://doi.org/10.1145/2933575.2934536

[28] Jeehoon Kang. 2018. crossbeam-rfcs. https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md.

[29] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.*

175–189. http://dl.acm.org/citation.cfm?id=3009850

[30] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *PACMPL* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

[31] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 649–662. https://doi.org/10.1145/2837614.2837643

[32] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. 69–80. https://doi.org/10.1145/2442516.2442524

[33] Linux contributors. 2014. Documentation/locking/spinlocks.txt. https://www.kernel.org/doc/Documentation/locking/spinlocks.txt.

[34] 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 *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

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

[36] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*. 267–275. https://doi.org/10.1145/248052.248106

[37] Brian Norris and Brian Demsky. 2016. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51. https://doi.org/10.1145/2806886

[38] Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 622–633. https://doi.org/10.1145/2837614.2837616

[39] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[40] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 311–322. https://doi.org/10.1145/2254064.2254102

[41] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 175–186. https://doi.org/10.1145/1993498.1993520

[42] Andrew Waterman and Krste Asanović. 2018. *The RISC-V instruction set manual volume I: user-level ISA.*

[43] Sizhuo Zhang, Arvind, and Muralidaran Vijayaraghavan. 2016. Taming weak memory models. *CoRR* abs/1606.05416 (2016). arXiv:1606.05416 http://arxiv.org/abs/1606.05416

[44] Sizhuo Zhang, Muralidaran Vijayaraghavan, and Arvind. 2017. Weak memory models: balancing definitional simplicity and implementation flexibility. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*. 288–302. https://doi.org/10.1109/PACT.2017.29

# A Full model

We continue the informal description of the model from §4 to extend the model with release/acquire instructions, weaker barriers, and load/store exclusive instructions before giving the full model definition including these.

## A.1 Release/acquire accesses and weak barriers

In addition to the full `dmb.sy` barriers and dependencies, ordering can also be created in ARMv8/RISC-V using weaker barriers:

- Release and acquire are "half-barriers" that introduce ordering in one direction: a store release is ordered with respect to all program-order previous memory accesses, and a load acquire with all program-order later ones. Moreover, a strong load acquire (acq, not wacq) is ordered with respect to all program-order-earlier strong store releases (rel, not wrel). Only RISC-V features weak releases.
- `dmb.ld` orders any program-order earlier loads with any program-order later memory access.
- `dmb.st` creates ordering from any program-order earlier store to any program-order later store.
- `isb` orders any load $i$ before any program-order later store $i'$, if there is a conditional branch between $i$ and $i'$ whose condition depends on $i$, or if there is a memory access between $i$ and $i'$ whose address depends on $i$.
- The RISC-V equivalent of `dmb.sy`, `dmb.ld`, and `dmb.st` are $\texttt{fence}_{\texttt{RW,RW}}$, $\texttt{fence}_{\texttt{R,RW}}$, and $\texttt{fence}_{\texttt{W,W}}$, respectively. `isb` has no equivalent in RISC-V: the fence `fence.i` does not consider control and address-po dependencies. Since we do not model self-modifying code, `fence.i` would be a no-op in this model. RISC-V has some additional barriers, such as $\texttt{fence}_{\texttt{W,R}}$ and `fence.tso`, which work analogously to the ARMv8 barriers seen so far (see §A.3).

$$\begin{array}{l||l} (a)\ \texttt{store [x] 37}; & (c)\ \texttt{r}_1 := \texttt{load}_{\texttt{acq}}\ \texttt{[y]};\ //\ 42 \\ (b)\ \texttt{store}_{\texttt{rel}}\ \texttt{[y] 42} & (d)\ \texttt{r}_2 := \texttt{load [x]}\ //\ 0 \end{array}$$

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

***Release/acquire*** We return to the earlier MP example. Turning $b$ into a release write orders $b$ after $a$. Making $c$ an acquire load orders $d$ after $c$. Together, this forbids the behaviour in which $c$ reads 42 and $d$ 0. Assume Thread 1 promises $y = 42$ before $x = 37$, at timestamp 1. Executing $a$ places $x = 37$ at timestamp 2, and sets $\mathsf{v}_{\mathsf{wOld}} = 2$.

**ρ1** A store release includes in its pre-view the view of all previous memory accesses, captured by $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$.

Therefore, after $a$, the pre-view of $b$ is 2, and it cannot fulfil the promise at timestamp 1. So, in the example, when $c$ reads $y = 42$, memory must instead be $[1\colon \langle x := 37\rangle_1, 2\colon \langle y := 42\rangle_1]$ thereby setting $c$'s post-view to 2.

**ρ2** The load acquire, symmetrically to the store release, merges its post-view into $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$, affecting the pre-view of all future loads and store.

Therefore, $c$ sets $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to 2. Since in this state $d$ is constrained by timestamp $\mathsf{v}_{\mathsf{rNew}} = 2$ and the initial $x = 0$ is superseded by the write at timestamp $1 \leq 2$, the behaviour where $d$ reads $x = 0$ is forbidden.

In addition, ARMv8/RISC-V enforces ordering from strong store releases to program-order later strong load acquires. To model this ordering:

**ρ3** The thread state maintains a view $\mathsf{v}_{\mathsf{Rel}} : \mathbb{V}$ containing the maximal post-view of all strong releases executed so far.
**ρ4** The pre-view of any later strong load acquire includes $\mathsf{v}_{\mathsf{Rel}}$, enforcing the memory ordering.

The rules for barriers follow the same principle:

**ρ5** `dmb.st` updates $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{wOld}}$.
**ρ6** `dmb.ld` updates $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{rOld}}$.
**ρ7** `isb` updates $\mathsf{v}_{\mathsf{rNew}}$ to include $\mathsf{v}_{\mathsf{CAP}}$.

## A.2 Load/store exclusive instructions

The previously discussed instructions can only introduce intra-thread ordering. Exclusive instructions (called load reserve/store conditional in RISC-V) make it possible to provide inter-thread atomicity guarantees. If a load exclusive $a$ and a store exclusive $b$ are paired and the store exclusive $b$ is successful, then the write $w'$ of $b$ is guaranteed to be the immediate coherence successor of the write $w$ that $a$ reads from, apart from writes by the same thread (*i.e.* there are no writes from other threads to the same address between $w$ and $w'$ in memory).

$$\begin{array}{l||l} (a)\ \texttt{r}_1 := \texttt{load}_{\texttt{ex}}\ [\texttt{x}];\ //\ 37 & (c)\ \texttt{store}\ [\texttt{x}]\ 37; \\ (b)\ \texttt{r}_2 := \texttt{store}_{\texttt{ex}}\ [\texttt{x}]\ 42 & (d)\ \texttt{store}\ [\texttt{x}]\ 51; \\ & (e)\ \texttt{r}_3 := \texttt{load}\ [\texttt{x}]\ //\ 42 \end{array}$$

$r_1 = 37 \wedge r_2 = \mathsf{v}_{\mathsf{succ}} \wedge r_3 = 42$ forbidden

In this example, if $a$ reads $x = 37$ from $c$, and $b$ succeeds, then the write $x = 51$ by $d$ is not allowed to come between the writes of $c$ and $b$, and memory is not allowed to be $[1\colon \langle x := 37\rangle_2, 2\colon \langle x := 51\rangle_2, 3\colon \langle x := 42\rangle_1]$ (although different-address writes and non-exclusive writes to $x$ from Thread 1 are allowed in between the load exclusive and the store exclusive). A store exclusive is only allowed to be paired with the most recent program-order earlier load exclusive (whether at the same location or not), and only if there has been no interposing (successful or unsuccessful) store exclusive, independent of their locations.

To capture the pairing of load and store exclusives:

**ρ8** Each thread maintains an *exclusives bank* $\mathsf{xclb} : \mathsf{option}\ \langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}\rangle$, initially set to *none*, containing information about the last load exclusive when there has been no other store exclusive in that thread since then. Specifically,

**ρ9** $\mathsf{xclb}$ is set to $\langle \mathsf{time} = t; \mathsf{view} = v\rangle$ (we omit *some* for simplicity) whenever a load exclusive reads from timestamp $t$ with post-view $v$.

**ρ10** $\mathsf{xclb}$ is set to *none* whenever a store exclusive (successful or not) is executed.

Consider an execution of the previous example, where $c$ writes $x = 37$ at timestamp 1 and $a$ reads the write $x = 37$ and sets $\mathsf{xclb}$ to $\langle \mathsf{time} = 1, \mathsf{view} = 1\rangle$. Now if $d$ writes $x = 51$ at timestamp 2, $b$ cannot write to $x$ exclusively and must fail by the following rule:

**ρ11** A store exclusive to location $z$ at timestamp $t$ succeeds only if $\mathsf{xclb}$ is not *none* and, in case the message at $\mathsf{xclb.time}$ is also $z$ (so the load exclusive was to the same location), every message to $z$ in memory between $\mathsf{xclb.time}$ and $t$ is written by this thread.

**ρ12** When the store exclusive succeeds it writes to a register indicating its success. The associated view in the success case is its post-view in RISC-V, and 0 in ARMv8. This means in RISC-V if another write depends on the success of a store exclusive, this write can only be promised after that of the store exclusive. In contrast, in ARMv8 this ordering is not preserved. This is the source of the deadlocks discussed in §4.3, for which we present a solution in §C.

Specifically, in Thread 1, $\mathsf{xclb.time}$ is 1 and $d$ should write $x = 42$ at timestamp 3, but then the write $x = 51$ in the middle is written by Thread 2, which violates the above rule. Thus in order for $b$ to be successful, $b$ should be executed before $d$ resulting in memory $[1\colon \langle x := 37\rangle_2, 2\colon \langle x := 42\rangle_1, 3\colon \langle x := 51\rangle_2]$ after $d$. Then $e$ is constrained by $\mathsf{coh}(x) = 3$, which is due to $d$, and thus should read 51.

In addition to this atomicity guarantee, exclusives provide some ordering guarantees: the architectures guarantee that certain loads — load acquires in ARMv8, all loads in RISC-V — cannot read from a store exclusive by thread-internal forwarding. To capture this:

**ρ13** Recall, the forward bank $\mathsf{fwdb} : \mathsf{Loc} \rightarrow \mathsf{option}\ [\mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}; \mathsf{xcl} : \mathbb{B}]$ records in the $\mathsf{xcl}$ field whether the write in the forward bank is an exclusive write. The model then prevents a load acquire on ARM, and any load in RISC-V, from a location $z$ from obtaining the smaller forward view $\mathsf{fwdb}(z).\mathsf{view}$ if $\mathsf{fwdb}(z).\mathsf{xcl}$ is set.

RISC-V additionally guarantees ordering of the store exclusive with the paired load-exclusive even if the load and the store are to different addresses.[3] To this end:

**ρ14** Recall that the exclusives bank $\mathsf{xclb} : \mathsf{option}\ \langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}\rangle$ records the post-view of the load exclusive in the view field. In RISC-V, this view $\mathsf{xclb.view}$ is included in the paired store exclusive's pre-view.

### A.3 Formal model

Fig. 4 summarises the types used by the model that were introduced in §4. For simplicity, values and addresses are mathematical integers. Promising-ARM and Promising-RISC-V use the same definitions, with an architecture flag $a$ switching between ARM and RISC-V behaviour. This only affects the treatment of store exclusive instructions, in the store and load rules. However, not all instructions exist in both architectures: RISC-V has more barriers, and a weak store release. Fig. 5 gives the formal definition of the steps of the semantics, cross-referenced with the relevant rules in §4. First, we define auxiliary definitions.

***Expressions interpretation*** The function (second and third line) takes an expression and a register state $m$, and returns the expression's value and view. Constants have view 0; registers are looked up in $m$; the view for an arithmetic expression merges the views of the arguments (**r9**).

[3] In the case of ARMv8 the architecture specifies "constrained unpredictable" behaviour; this is still being clarified.

$$a \in \text{Arch} ::= \texttt{ARM} \mid \texttt{RISC-V} \qquad l \in \text{Loc} \overset{def}{=} \text{Val} \qquad v \in \text{Val} \overset{def}{=} \mathbb{Z} \qquad tid \in \text{TId} \overset{def}{=} \mathbb{N} \qquad t \in \mathbb{T} \overset{def}{=} \mathbb{N} \qquad \nu \in \mathbb{V} \overset{def}{=} \mathbb{T}$$

$$w \in \text{Msg} \overset{def}{=} \langle \text{loc}:\text{Loc}; \text{val}:\text{Val}; \text{tid}:\text{TId}\rangle \qquad \langle x := v\rangle_{tid} \overset{def}{=} \langle \text{loc}=x; \text{val}=v; \text{tid}=tid\rangle \qquad M \in \text{Memory} \overset{def}{=} \text{list Msg}$$

$$ts \in \text{TState} \overset{def}{=} \left\langle \begin{array}{l} \text{prom} : \text{set } \mathbb{T}; \quad \text{regs} : \text{Reg} \to \text{Val} \times \mathbb{V}; \\ \text{coh} : \text{Loc} \to \mathbb{V}; \quad \mathsf{v}_{\mathsf{rOld}}, \mathsf{v}_{\mathsf{wOld}}, \mathsf{v}_{\mathsf{rNew}}, \mathsf{v}_{\mathsf{wNew}}, \mathsf{v}_{\mathsf{CAP}}, \mathsf{v}_{\mathsf{Rel}} : \mathbb{V}; \\ \text{fwdb} : \text{Loc} \to \langle \text{time} : \mathbb{T}; \text{view} : \mathbb{V}; \text{xcl} : \mathbb{B}\rangle; \\ \text{xclb} : \text{option } \langle \text{time} : \mathbb{T}; \text{view} : \mathbb{V}\rangle \end{array} \right\rangle \qquad \begin{array}{r} T \in \text{Thread} \overset{def}{=} \text{St} \times \text{TState} \\ \vec{T} \in \text{TPool} \overset{def}{=} \text{TId} \to \text{Thread} \\ \langle \vec{T}, M\rangle \in \text{Machine} \overset{def}{=} \text{TPool} \times \text{Memory} \end{array}$$

**Figure 4.** Types in the semantics

***read*** read$(M, l, t)$ gives the result of reading location $l$ at timestamp $t$ in memory $M$. For $t = 0$, this is the initial value $v_{\text{init}}$, here 0; otherwise either the value of the message in $M$ at timestamp $t$ if its location is $l$, otherwise *none*.

***read-view*** read-view$(a, rk, f, t)$ returns either the timestamp $t$ of the read message or the forward view of the message $f$ in the forward bank, subject to certain constraints on the architecture $a$ and read kind $rk$ (**R13**, **R14**, **R15**, **R16**, **ρ13**).

***atomic*** atomic$(M, l, tid, t_r, t_w)$ checks whether an exclusive write to $l$ at timestamp $t_w$ by thread $tid$ can become successful, and so atomic with respect to its earlier exclusive read with read message at timestamp $t_r$ in the current memory $M$ (**ρ8**, **ρ9**, **ρ11**).

Now we define thread-local steps $T, M \xrightarrow{[t]}_{tid} T'$, which do not change the memory.

***EXCLUSIVE-FAILURE*** A store exclusive that has not been executed is always allowed to fail. It sets $r_{\text{succ}}$ to $\mathsf{v}_{\text{fail}}$ (here 1) to signal failure, with 0 timestamp, and sets xclb to *none* (**ρ10**).

***FULFIL*** starts with the pre-condition (from top to bottom). First evaluate address and data expressions. Rule **ρ11** explains the condition for exclusive writes. Since we assume writes always promise first and then fulfil, this step requires the write to have been promised.

Rules **R10**, **R6**, **R21**, **ρ1**, **ρ14** describe the components contributing to the pre-view. The pre-view and coherence view have to be less than $t$ (**R19**); the post-view is the timestamp $t$ (**R3**). **ρ12** explains the view $\nu_{\text{succ}}$ placed on the register write indicating the success. The post-condition removes the promise (**R19**); writes $\mathsf{v}_{\text{succ}}$ (here 0) to the "success register" (**ρ12**); and updates the coherence view to include $t$ (**R11**), certain views (**R5**, **R22**, **ρ3**); the forward bank (**R14**, **ρ13**); and the exclusives bank (**ρ8**, **ρ10**).

***READ*** also starts with the pre-condition (from top to bottom). First evaluate the address $l$; in order to read $v$ it must be $v = \text{read}(M, l, t)$ as described above. The pre-view calculation is described in **R10**, **R6**, **ρ4**. The pre-view (**R2**) and the coherence view (**R12**) constrain the read. The post-view is defined in **R3**, **R16**. The post-condition updates the register with value and post-view (**R9**); coherence with post-view as in rule **R11**; views as in **R5**, **ρ2**, **R22**; and exclusives bank as in **ρ9**.

***FENCE*** This defines a single rule for all other ARMv8 and RISC-V fences in a format matching RISC-V's `fence` instruction. $\texttt{fence}_{K_1,K_2}$ has two arguments: $K_1$ indicates whether the fence creates ordering with respect to program-order preceding reads ($R$), writes ($W$), or both ($RW$); similarly $K_2$ indicates which program-order later instructions are ordered with it ($R$, $W$, or $RW$). It then updates $\mathsf{v}_{\mathsf{rNew}}$ and/or $\mathsf{v}_{\mathsf{wNew}}$ (depending on $K_2$), to include $\mathsf{v}_{\mathsf{rOld}}$ and/or $\mathsf{v}_{\mathsf{wOld}}$ (depending on $K_1$) according to the intuition given in **R5**, **R6**. We define ARMv8's full barrier `dmb.sy` = $\texttt{fence}_{RW,RW}$, its load barrier `dmb.ld` = $\texttt{fence}_{R,RW}$, its store barrier `dmb.st` = $\texttt{fence}_{W,W}$, and moreover RISC-V's "TSO fence" as
`fence.tso` = $\texttt{fence}_{R,R}; \texttt{fence}_{RW,W}$. With these definitions, the behaviour of the ARM barriers is as explained with rules **R5**, **R6** **R7**, **ρ5**, **ρ6**.

***REGISTER*** A register assignment updates the register with the expressions and view from the evaluation of its expression (**R9**).

***BRANCH*** The pre-condition evaluates the condition, branches as determined by this value, and updates $\mathsf{v}_{\mathsf{CAP}}$ (**R22**).

***ISB*** Executes an `isb` by merging $\mathsf{v}_{\mathsf{CAP}}$ into $\mathsf{v}_{\mathsf{rNew}}$ (**ρ7**).

***SKIP, SEQ, and WHILE*** Mostly as expected. `while` is expressed using a branch.

We can then define thread steps $\langle T, M\rangle \xrightarrow{[t]}_{tid} \langle T', M'\rangle$.

***EXECUTE*** lifts a thread-local step that does not change memory to a thread step. PROMISE allows promising any write message, appending this write to memory and recording its timestamp in prom. While thread steps allow unconstrained promises, machine steps only allow certified promises. Finally, machine steps just lift *certified* thread steps (**R24**).

$$c \mathbin{?} v_1 : v_2 \overset{def}{=} \text{if } c \text{ then } v_1 \text{ else } v_2 \qquad c \mathbin{?} v \overset{def}{=} c \mathbin{?} v : 0 \qquad v_1 \sqcup v_2 \overset{def}{=} \max(v_1, v_2) \qquad v@v \overset{def}{=} \langle v, v\rangle : \mathrm{Val} \times \mathbb{V}$$

$$\llbracket(-)_1\rrbracket_{(-)_2} : \mathit{Expr} \to (\mathrm{Reg} \to \mathrm{Val} \times \mathbb{V}) \to \mathrm{Val} \times \mathbb{V}$$

$$\llbracket v\rrbracket_m \overset{def}{=} v@0 \qquad \llbracket r\rrbracket_m \overset{def}{=} m(r) \qquad \llbracket e_1 \ \mathsf{op}\ e_2\rrbracket_m \overset{def}{=} (v_1 \llbracket \mathsf{op}\rrbracket v_2)@(v_1 \sqcup v_2) \text{ with } \llbracket e_1\rrbracket_m = v_1@v_1, \llbracket e_2\rrbracket_m = v_2@v_2$$

$$\mathrm{read}(M, l, t) : \text{option Val} \overset{def}{=} \text{if } t = 0 \text{ then } \mathsf{v}_{\mathrm{init}} \text{ else if } M(t).\mathrm{loc} = l \text{ then } M(t).\mathrm{val} \text{ else } \mathit{none}$$

$$\text{read-view}(a, \mathit{rk}, f, t) \overset{def}{=} \text{if } (f.\mathrm{time} = t \wedge (f.\mathrm{xcl} \Rightarrow (a = \mathsf{ARM} \wedge \mathit{rk} \sqsubseteq \mathrm{pln}))) \text{ then } f.\mathrm{view} \text{ else } t$$

$$\mathrm{atomic}(M, l, \mathit{tid}, t_{\mathrm{r}}, t_{\mathrm{w}}) \overset{def}{=} M(t_{\mathrm{r}}).\mathrm{loc} = l \implies \forall t'.\ (t_{\mathrm{r}} < t' < t_{\mathrm{w}} \wedge M(t').\mathrm{loc} = l) \implies M(t').\mathrm{tid} = \mathit{tid}$$

$$\boxed{T, M \xrightarrow{[t]}_{a,\mathit{tid}} T'}$$

**(exclusive-failure)**

$$\frac{\mathit{xcl} = \mathtt{true} \quad \mathit{ts}' = \mathit{ts}\big[\mathrm{regs}(r_{\mathrm{succ}}) \mapsto \mathsf{v}_{\mathrm{fail}}@0, \mathrm{xclb} \mapsto \mathit{none}\big]}{\langle r_{\mathrm{succ}} := \mathtt{store}_{\mathit{xcl},\mathit{wk}}[e_1]\, e_2, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle \mathtt{skip}, \mathit{ts}'\rangle}$$

**(read)**

$$\frac{\begin{array}{l} l@v_{\mathrm{addr}} = \llbracket e\rrbracket_{\mathit{ts}.\mathrm{regs}} \\ \mathrm{read}(M, l, t) = v \\ v_{\mathrm{pre}} = v_{\mathrm{addr}} \sqcup \mathit{ts}.\mathsf{v}_{\mathrm{rNew}} \sqcup (\mathit{rk} \sqsupseteq \mathrm{acq} \mathbin{?} \mathit{ts}.\mathsf{v}_{\mathrm{Rel}}) \\ \forall t'.\ t < t' \le (v_{\mathrm{pre}} \sqcup \mathit{ts}.\mathrm{coh}(l)) \implies M(t').\mathrm{loc} \neq l \\ v_{\mathrm{post}} = v_{\mathrm{pre}} \sqcup \text{read-view}(a, \mathit{rk}, \mathit{ts}.\mathrm{fwdb}(l), t) \\ \mathit{ts}' = \mathit{ts}\left[\begin{array}{l} \mathrm{regs}(r) \mapsto v@v_{\mathrm{post}}, \\ \mathrm{coh}(l) \mapsto \mathit{ts}.\mathrm{coh}(l) \sqcup v_{\mathrm{post}}, \\ \mathsf{v}_{\mathrm{rOld}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{rOld}} \sqcup v_{\mathrm{post}}, \\ \mathsf{v}_{\mathrm{rNew}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{rNew}} \sqcup (\mathit{rk} \sqsupseteq \mathrm{wacq} \mathbin{?} v_{\mathrm{post}}), \\ \mathsf{v}_{\mathrm{wNew}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{wNew}} \sqcup (\mathit{rk} \sqsupseteq \mathrm{wacq} \mathbin{?} v_{\mathrm{post}}), \\ \mathsf{v}_{\mathrm{CAP}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{CAP}} \sqcup v_{\mathrm{addr}}, \\ \mathrm{xclb} \mapsto \mathit{xcl} \mathbin{?} \langle \mathrm{time} = t; \mathrm{view} = v_{\mathrm{post}}\rangle : \mathit{ts}.\mathrm{xclb} \end{array}\right] \end{array}}{\langle r := \mathtt{load}_{\mathit{xcl},\mathit{rk}}\ [e], \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle \mathtt{skip}, \mathit{ts}'\rangle}$$

**(fulfil)**

$$\frac{\begin{array}{l} \llbracket e_1\rrbracket_{\mathit{ts}.\mathrm{regs}} = l@v_{\mathrm{addr}} \qquad \llbracket e_2\rrbracket_{\mathit{ts}.\mathrm{regs}} = v@v_{\mathrm{data}} \\ \mathit{xcl} \implies \mathit{ts}.\mathrm{xclb} \neq \mathit{none} \wedge \mathrm{atomic}(M, l, \mathit{tid}, \mathit{ts}.\mathrm{xclb}.\mathrm{time}, t) \\ t \in \mathit{ts}.\mathrm{prom} \qquad M(t) = \langle l := v\rangle_{\mathit{tid}} \\ v_{\mathrm{pre}} = v_{\mathrm{addr}} \sqcup v_{\mathrm{data}} \sqcup \mathit{ts}.\mathsf{v}_{\mathrm{wNew}} \sqcup \mathit{ts}.\mathsf{v}_{\mathrm{CAP}} \sqcup \\ \qquad (\mathit{wk} \sqsupseteq \mathrm{wrel} \mathbin{?} (\mathit{ts}.\mathsf{v}_{\mathrm{rOld}} \sqcup \mathit{ts}.\mathsf{v}_{\mathrm{wOld}})) \sqcup \\ \qquad ((a = \mathsf{RISC\text{-}V} \wedge \mathit{xcl}) \mathbin{?} \mathit{ts}.\mathrm{xclb}.\mathrm{view}) \\ (v_{\mathrm{pre}} \sqcup \mathit{ts}.\mathrm{coh}(l)) < t \\ v_{\mathrm{post}} = t \qquad v_{\mathrm{succ}} = (a = \mathsf{RISC\text{-}V} \mathbin{?} v_{\mathrm{post}} : \bot) \\ \mathit{ts}' = \mathit{ts}\left[\begin{array}{l} \mathrm{prom} \mapsto \mathit{ts}.\mathrm{prom} \setminus \{t\}, \\ \mathrm{regs}(r_{\mathrm{succ}}) \mapsto \mathit{xcl} \mathbin{?} \mathsf{v}_{\mathrm{succ}}@v_{\mathrm{succ}} : \mathit{ts}.\mathrm{regs}(r_{\mathrm{succ}}), \\ \mathrm{coh}(l) \mapsto \mathit{ts}.\mathrm{coh}(l) \sqcup v_{\mathrm{post}}, \\ \mathsf{v}_{\mathrm{wOld}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{wOld}} \sqcup v_{\mathrm{post}}, \\ \mathsf{v}_{\mathrm{CAP}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{CAP}} \sqcup v_{\mathrm{addr}}, \\ \mathsf{v}_{\mathrm{Rel}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{Rel}} \sqcup (\mathit{wk} \sqsupseteq \mathrm{rel} \mathbin{?} v_{\mathrm{post}}), \\ \mathrm{fwdb}(l) \mapsto \langle \mathrm{time} = t; \mathrm{view} = v_{\mathrm{addr}} \sqcup v_{\mathrm{data}}; \mathrm{xcl} = \mathit{xcl}\rangle \\ \mathrm{xclb} \mapsto \mathit{xcl} \mathbin{?} \mathit{none} : \mathit{ts}.\mathrm{xclb} \end{array}\right] \end{array}}{\langle r_{\mathrm{succ}} := \mathtt{store}_{\mathit{xcl},\mathit{wk}}\ [e_1]\ e_2, \mathit{ts}\rangle, M \xrightarrow{t}_{a,\mathit{tid}} \langle \mathtt{skip}, \mathit{ts}'\rangle}$$

**(fence)**

$$\frac{\begin{array}{l} v_1 = (\mathsf{R} \sqsubseteq K_1 \mathbin{?} \mathit{ts}.\mathsf{v}_{\mathrm{rOld}}) \sqcup \\ \qquad (\mathsf{W} \sqsubseteq K_1 \mathbin{?} \mathit{ts}.\mathsf{v}_{\mathrm{wOld}}) \end{array} \qquad \mathit{ts}' = \mathit{ts}\left[\begin{array}{l} \mathsf{v}_{\mathrm{rNew}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{rNew}} \sqcup (\mathsf{R} \sqsubseteq K_2 \mathbin{?} v_1), \\ \mathsf{v}_{\mathrm{wNew}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{wNew}} \sqcup (\mathsf{W} \sqsubseteq K_2 \mathbin{?} v_1) \end{array}\right]}{\langle \mathtt{fence}_{K_1,K_2}, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle \mathtt{skip}, \mathit{ts}'\rangle}$$

**(register)**

$$\frac{\mathit{ts}' = \mathit{ts}\big[\mathrm{regs}(r) \mapsto \llbracket e\rrbracket_{\mathit{ts}.\mathrm{regs}}\big]}{\langle r := e, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle \mathtt{skip}, \mathit{ts}'\rangle}$$

**(branch)**

$$\frac{\llbracket e\rrbracket_{\mathit{ts}.\mathrm{regs}} = v@v \qquad \mathit{ts}' = \mathit{ts}[\mathsf{v}_{\mathrm{CAP}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{CAP}} \sqcup v]}{\langle \mathtt{if}\ (e)\ s_1\ s_2, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle v \neq 0 \mathbin{?} s_1 : s_2, \mathit{ts}'\rangle}$$

**(isb)**

$$\frac{\mathit{ts}' = \mathit{ts}[\mathsf{v}_{\mathrm{rNew}} \mapsto \mathit{ts}.\mathsf{v}_{\mathrm{rNew}} \sqcup \mathit{ts}.\mathsf{v}_{\mathrm{CAP}}]}{\langle \mathtt{isb}, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle \mathtt{skip}, \mathit{ts}'\rangle}$$

**(skip)**

$$\frac{}{\langle \mathtt{skip}; s, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle s, \mathit{ts}\rangle}$$

**(seq)**

$$\frac{\langle s_1, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle s_1', \mathit{ts}'\rangle}{\langle s_1; s_2, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle s_1'; s_2, \mathit{ts}'\rangle}$$

**(while)**

$$\frac{s' = \mathtt{if}\ (e)\ (s; \mathtt{while}\ (e)\ s)\ \mathtt{skip}}{\langle \mathtt{while}\ (e)\ s, \mathit{ts}\rangle, M \to_{a,\mathit{tid}} \langle s', \mathit{ts}\rangle}$$

$$\boxed{\langle T, M\rangle \xrightarrow{[t]}_{a,\mathit{tid}} \langle T', M'\rangle}$$

**(execute)**

$$\frac{T, M \to_{a,\mathit{tid}} T'}{\langle T, M\rangle \to_{a,\mathit{tid}} \langle T', M\rangle}$$

**(promise)**

$$\frac{w.\mathrm{tid} = \mathit{tid} \quad t = |M| + 1 \quad \mathit{ts}' = \mathit{ts}[\mathrm{prom} \mapsto \mathit{ts}.\mathrm{prom} \cup \{t\}]}{\langle\langle s, \mathit{ts}\rangle, M\rangle \xrightarrow{t}_{a,\mathit{tid}} \langle\langle s, \mathit{ts}'\rangle, M +\!\!+ [w]\rangle}$$

$$\boxed{\langle \vec{T}, M\rangle \to_a \langle \vec{T}', M'\rangle}$$

**(machine-step)**

$$\frac{\langle \vec{T}[\mathit{tid}], M\rangle \to_{a,\mathit{tid}} \langle T', M'\rangle \qquad \langle T', M'\rangle \text{ certified}}{\langle \vec{T}, M\rangle \to_a \langle \vec{T}\,[\mathit{tid} \mapsto T'], M'\rangle}$$

$$\langle T, M\rangle \text{ certified} \overset{def}{=} \exists T', M'.\ \langle T, M\rangle \xrightarrow{\mathrm{seq}}{}^{*}_{a,\mathit{tid}} \langle T', M'\rangle \wedge T'.\mathrm{prom} = \{\}$$

$$\boxed{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{a,\mathit{tid}} \langle T', M'\rangle}$$

**(seq-exec)**

$$\frac{T, M \to_{a,\mathit{tid}} T'}{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{a,\mathit{tid}} \langle T', M\rangle}$$

**(seq-write)**

$$\frac{\begin{array}{c} \langle T, M\rangle \xrightarrow{t}_{a,\mathit{tid}} \langle T', M'\rangle \\ T', M' \xrightarrow{t}_{a,\mathit{tid}} T'' \end{array}}{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{a,\mathit{tid}} \langle T'', M'\rangle}$$

**Figure 5.** Thread-local steps, thread steps, and machine steps

## B Algorithm

In the following, we describe the *algorithm* used by the executable model of §7 to implement the certification and promise enumeration: a function that enumerates the legal next (promise or non-promise) steps of a thread. The certification algorithm has to handle two tasks. Given a thread state and the current memory state, first it has to decide which of the possible next instructions steps of the thread allow fulfilling the promises the thread has already made. Second, it has to enumerate the possible new promises the thread should be allowed to make. These promises have to correspond to feasible writes by store instructions of this thread but also be compatible with the set of promises the thread has already “committed to”. The main challenge in developing the certification algorithm is in the latter, *computing* the new promise steps that should be enabled in the current thread configuration.

The model of §5 (and §A.3) allows adding arbitrary new promises *during the certification*. Doing the same in the executable model would make promise enumeration and certification computationally infeasible.

- The algorithm therefore forbids early promises during certification (*i.e.* only allows “normal writes”), using the fact that this does not change the model behaviour.

Moreover, in general, given an arbitrary program, fulfilling a promise may take arbitrarily many steps by this thread, in particular in the presence of loops whose execution is not statically bounded. For the sake of executability, the executable model necessarily bounds these, and the certification algorithm takes a *fuel* argument, limiting the number of thread steps the certification is allowed to take. The idea of the algorithm is then to enumerate all legal traces of this thread in isolation under current memory, of a length bounded by this argument, that lead to a state in which all of this thread’s promises have been fulfilled. Then:

1. Any such trace’s first (non-promise) step is immediately certified.
2. Moreover, any normal write done by this thread on such a trace corresponds to a legal promise step if it has the *pre-view and coherence-view (at its location)* less than or equal to the maximal timestamp of current memory (the memory before the start of the certification).

$$
\begin{array}{l||l}
(a)\ \mathtt{r}_1 := \mathtt{load}\ [\mathtt{w}]; & \\
(b)\ \mathtt{store}\ [\mathtt{x}]\ 1; & \ \dots \\
(c)\ \mathtt{store}_{\mathsf{rel}}\ [\mathtt{y}]\ 1; & \\
(d)\ \mathtt{store}\ [\mathtt{z}]\ \mathtt{r}_1 &
\end{array}
$$

To illustrate the algorithm, consider the above (partial) program. Assume that the memory is $[1{:}\ \langle w := 1\rangle_2, 2{:}\ \langle z := 1\rangle_1]$, that the promise set of Thread 1 is $\mathrm{prom} = \{2\}$, and that Thread 1 has not yet executed $a$. The certification algorithm first enumerates all traces of Thread 1 leading to states in which all its promises have been fulfilled. Here, there is only one such trace:

- $a$ reads 1 from $w$: otherwise $d$ would write $z = 0$, which cannot fulfil the outstanding promise $2{:}\ \langle z := 1\rangle_1$.
- $b$ writes $x = 1$ at timestamp 3 with pre-view 0 and coherence-view 0, leading to post-view 3.
- $c$ writes $y = 1$ at timestamp 4 with pre-view 3 and coherence-view 0: as a store release $c$’s pre-view includes $b$’s post-view, via $\mathsf{v}_{\mathsf{wOld}}$.
- And $d$ fulfils $2{:}\ \langle z := 1\rangle_1$.

Therefore:

1. The next-instruction step in which $a$ reads 1 from $w$ is a certified step for Thread 1. (The one where $a$ reads 0 is not, due to the outstanding promise $2{:}\ \langle z := 1\rangle_1$.)
2. Promising the write $x = 1$ at timestamp 3 is also certified: it is a possible write by Thread 1 on a path fulfilling its promises, and with a pre-view and coherence view both less than or equal to the current maximal timestamp 2 in memory (before the certification run).
3. Promising the write $y = 1$, however, is not a certified step, since $c$’s pre-view in the only possible certification trace is $3 \not\leq 2$.

## C Certification with ARMv8 store exclusives

In §4.3 and §5 we introduced a simple certification definition to avoid model executions in which not all promises are fulfilled. This certification is sound for RISC-V and ARMv8, and precise for RISC-V programs and for ARMv8 programs without store exclusives. In the presence of ARMv8 store exclusive instructions, however, it is imprecise: matching ARMv8’s architecturally intended weak semantics of store exclusive instructions in Promising-ARM leads to executions in which the model gets stuck due to unfulfilled promises, of a similar sort as those present in the Flat model [39]. In this section, we extend the model's machine state with locks and a certification that takes the locking into account to prevent these executions and make certification sound and precise for ARMv8 programs even with store exclusives, and makes the model deadlock-free.

### C.1 The challenge of certification with ARMv8 store exclusives

One of the main simplifications of the recent revision of ARMv8 was that, where the architecture previously distinguished between notions of "true" and "false" dependencies, it now makes no such distinction. Now syntactic dependencies of the right kind induce memory ordering, with no consideration of whether the result of the register computation varies as a function of its input or not. Therefore, in the revised ARMv8, it is not always sound to replace an expression by another expression that performs the same register computation. However, ARMv8's specification of store exclusives intends to allow processors to treat a load/store exclusive pair as a single atomic operation that is guaranteed to succeed, e.g. treating $\mathtt{r}_1 := \mathtt{load}_{\mathtt{ex}}\ [\mathtt{x}]; \mathtt{r}_2 := \mathtt{store}_{\mathtt{ex}}\ [\mathtt{x}]\ (\mathtt{r}_1 + 1)$ as $\mathtt{r}_1 := \mathtt{fetch\text{-}and\text{-}add}\ [x]; \mathtt{r}_2 := 0$. Now, $r_2$ still has to be set to indicate success ($\mathsf{v}_{\mathsf{succ}}$), but does not syntactically depend on the store. Therefore, in ARMv8, a dependency on $r_2$ in the program above does not induce ordering (and Promising-ARM sets its associated view to 0). But the value of $r_2$ after the store exclusive still depends on the success of the store exclusive!

This leads to surprising behaviours: in the following program despite the dependency from $b$ to $c$ they can be reordered. In particular, Thread 1 may (1.) execute $a$ and read the initial value 0 (so $r_1 = 0$) and, (2.) assume the success of $b$ (so $r_2 = \mathsf{v}_{\mathsf{succ}} = 0$) and write $p = 1$ with $c$, *before $b$ is in memory*: the architecture allows that $d$ reads $p = 1$ and $f$ reads $x = 0$ after the barrier $e$. (While in RISC-V the dependency from $b$ to $c$ means $b$ propagates before $c$, forbidding this behaviour.)

$$\begin{array}{l||l||l}
a : \mathtt{r}_1 := \mathtt{load}_{\mathtt{ex}}\ [\mathtt{x}]; & d : \mathtt{r}_3 := \mathtt{load}\ [\mathtt{p}];\ /\!/\ 1 & g : \mathtt{store}\ [\mathtt{x}]\ 2 \\
b : \mathtt{r}_2 := \mathtt{store}_{\mathtt{ex}}\ [\mathtt{x}]\ (\mathtt{r}_1 + 1); & e : \mathtt{dmb.sy}; & \\
c : \mathtt{store}\ [\mathtt{p}]\ (1 - \mathtt{r}_1 - \mathtt{r}_2) & f : \mathtt{r}_4 := \mathtt{load}\ [\mathtt{x}]\ /\!/\ 0 &
\end{array}$$
$$r_3 = 1 \wedge r_4 = 0 \text{ allowed}$$

This means whereas in all other cases a store may propagate to memory only when all its dependencies are "fixed", dependencies on the success of a store exclusive are special. In order to allow the above re-ordering of $b$ and $c$, an operational model has to do extra work, since it has to ensure the success of $b$ and therefore its atomicity with respect to the write $a$ read from, until $b$ is done. For the simple case above, mimicking the behaviour of a processor and replacing $\mathtt{r}_1 := \mathtt{load}_{\mathtt{ex}}\ [\mathtt{x}]; \mathtt{r}_2 := \mathtt{store}_{\mathtt{ex}}\ [\mathtt{x}]\ (\mathtt{r}_1 + 1)$ with $\mathtt{r}_1 := \mathtt{fetch\text{-}and\text{-}add}\ [x]; \mathtt{r}_2 := 0$ is easy. However, handling the dependency relaxation in its full generality (without deadlocks) is difficult.

This problem manifests in Promising-ARM in the following way: in the initial state Thread 1 is allowed to promise $p = 0$. Since $c$ can produce a write $p = 0$ only if $a$ reads $x = 0$ and $b$ succeeds, the ability of Thread 1 to fulfil the promise now depends $b$'s success, and so on whether $b$'s write can enter memory as the next write to location $x$ (after the initial write $x = 0$). If, however, $g$ now writes to $x$, $b$ will fail and the model gets stuck, with $c$ unfulfilled. Since $c$'s early promise has to be allowed to match ARMv8 semantics, the model must instead prevent $g$ from writing until $b$'s write, since $g$ would break Thread 1's promise.

The certification definition presented in §4 and 5 works thread-locally: it takes into account only a thread's state and current memory in order to decide whether a thread step should be allowed or not. But whether $g$'s write should be allowed cannot be determined based only on the state of Thread 3 and on the list of messages in memory: it is Thread 1's promises due to which $g$ must not write. So a precise certification algorithm for ARMv8 needs to take into account some information about other threads. In this example, Thread 1 effectively requires "locking" location $x$, to constrain the behaviour of the other threads. We leverage this intuition of a thread locking a location and extend memory with a lock state, in order to still allow certifying a thread by taking into account only its own state and the (extended) memory state.

### C.2 Extended certification, take 1

The example indicates a pattern: in the problematic execution a thread's ability to fulfil its promises depends on both, a read exclusive, and the success of a paired write exclusive that has been promised. More precisely, the state in which a Thread $n$ requires a lock on a location $x$ is the following:

- Thread $n$ depends on a load exclusive $l$ to location $x$. This is if:
  - Thread $n$ has already executed $l$, or
  - Thread $n$ has an outstanding promise whose fulfilment depends on $l$ due to register dataflow or due to coherence or view requirements.
- And Thread $n$ relies on the success of the write of a store exclusive $s$ to $x$ that is paired with $l$: Thread $n$ has an outstanding promise whose fulfilment depends on $s$ via register dataflow.

- And the write $w'$ that $l$ read from is already in memory, whereas the write of $s$ is not.

If the above holds, Thread $n$'s dependency on $l$ "fixes" the write $w'$ that $l$ reads from, and due to the success dependency on $s$ requires the write of $s$ to succeed and be atomic with respect to $w'$.

The idea underlying the extended model is to precisely detect cases when this condition holds, and to then lock $x$ for Thread $n$ in memory for as long as the condition holds and prevent other threads from writing to locked locations. The main challenge here is in detecting the above condition. The model handles this by extending the certification and generalising the views of the thread state. During the certification the model tracks dependencies from load and store exclusive instructions to other stores, in order to detect when the fulfilment of a write promise by some store depends on such load/store exclusive pairs as described in the condition. To this end, the extended certification uses views that in addition to timestamps carry *taints* that keep track of the load/store exclusive instruction dependencies, including information about their memory location and pairing.

In the example above, in the state after Thread 1 has read $x = 0$ with $a$ and promised $p = 1$ with $c$, the extended certification will work as follows:

- the only certifying execution of Thread 1 alone under current memory is one where $a$ reads $x = 0$, and $b$ succeeds. In this execution:
- $a$ taints $r_1$ to indicate it is from a load exclusive $a$ to location $x$ reading from a value in memory.
- $b$ taints $r_2$ to indicate it is from a successful store exclusive to location $x$ whose write is not in memory yet and that is paired with $a$.
- when fulfilling $p = 0$ with $c$, $c$'s pre-view includes a taint with information about both $a$ and $b$: $a$ and $b$ are paired and to location $x$; $a$ reads at timestamp 0, so from a write in memory; $b$ is not propagated yet.
- Therefore Thread 1 requires locking $x$.

The information returned by the certification is then, which locations have to be locked in order to *guarantee* a thread can fulfil its promises, and a machine step is only allowed if the step is compatible with the current lock state in memory: not writing to a location locked by another thread, and not locking already-locked locations.

The taint tracking introduces complexity to the certification. Importantly however, during "normal" execution the extended model's views are simple timestamps just as before (§5), and taints are not persistent but local to the certification.

### C.3 Extended certification, take 2

As the following example illustrates, unfortunately the ideas on certification above are still insufficient. In this example Thread 1's store $d$ depends on the load exclusive $b$ to $x$, and the "success register write" of the store exclusive $c$ to location $x$. New here is that $c$ has release ordering, and so $c$ is ordered after the write $a$ to $y$. Symmetrically in Thread 2 $h$ depends on the load and store exclusive instructions $f$ and $g$ to $y$, and $g$ is a store exclusive release ordered after a store $e$ to $x$.

$$\begin{array}{l||l}
a : \mathtt{store}\ [\mathtt{y}]\ 1; & e : \mathtt{store}\ [\mathtt{x}]\ 1; \\
b : \mathtt{r}_1 := \mathtt{load}_{\mathtt{ex}}\ [\mathtt{x}]; & f : \mathtt{r}_3 := \mathtt{load}_{\mathtt{ex}}\ [\mathtt{y}]; \\
c : \mathtt{r}_2 := \mathtt{store}_{\mathtt{ex,rel}}\ [\mathtt{x}]\ 1; & g : \mathtt{r}_4 := \mathtt{store}_{\mathtt{ex,rel}}\ [\mathtt{y}]\ 1; \\
d : \mathtt{store}\ [\mathtt{p}]\ (1 - \mathtt{r}_1 - \mathtt{r}_2) & h : \mathtt{store}\ [\mathtt{q}]\ (1 - \mathtt{r}_3 - \mathtt{r}_4)
\end{array}$$

Now assume an execution in which Thread 1 promises $p = 1$. Since this depends on $b$ reading $x = 0$ and $c$ eventually successfully writing $x = 1$, our extended certification requires a lock on $x$ for Thread 1 to prevent Thread 2 from breaking its promise. Now Thread 2 could analogously promise $q = 1$, after which the model also locks location $y$ for Thread 2, which does not contradict Thread 1 locking $x$. But now the model is stuck again: Thread 1 cannot execute $a$, since $y$ is locked by Thread 2; $c$'s write $x = 1$ would release the lock on $x$, but since $c$ is a store release, this requires first promising $a$. Thread 2 in turn cannot execute $e$ due to the lock on $x$, and cannot promise $g$'s $y = 1$ (and unlock $y$) before executing $e$.

In order to avoid such executions, the extended certification has to be improved to take into account some information about the thread-internal ordering requirements in order to prevent model deadlocks. To this end, the extended model's taints carry additional information about stores preceding store exclusive release instructions and the lock state captures rely-guarantee style lock information per thread; a machine step is then only allowed if the rely-guarantee lock information of all thread states is consistent. Then in the previous bad execution:

- For the promise $p = 1$ the certification returns information of the form $([y]; x)$, meaning that for this step Thread 1 requires a lock on $x$, and that it *relies* on being able to write to $y$ before releasing the lock on $x$. Promising $p = 1$ adds this to the memory's lock state.
- Since there are no other locks, Thread 1 can promise.
- In the following state, for the promise $q = 1$ by Thread 2, symmetrically, the certification returns the information $([x], y)$.

- Now $([x], y)$ is incompatible with $([y], x)$ due to the cyclic rely-guarantee dependency. Thus Thread 2 is not allowed to promise $q = 1$.

Moreover, certain sequences of multiple such store exclusive release instructions can lead to *nesting* of these rely-guarantee locks, making the consistency checking difficult. In particular, our current algorithm for checking the consistency is exponential in the nesting depth. We believe, in practice, sequences with nesting depth greater than 1 do not occur "naturally". Hence, for the purpose of exhaustive state space exploration, the executable model approximates the lock information and consistency checking up to depth one. The model may then still get stuck in cases requiring depth more than 1, but consistency checking becomes linear in the size of the lock information. (Irrespective, the model remains sound.)

For lack of space we omit the details of the extended certification and refer the interested reader to the Coq formalisation in the supplementary material.

## D Equivalence with the reference axiomatic memory model

The argument *arch* switches between ARMv8 and RISC-V. For simplicity of the formalisation, the barriers here are `dmb.rw`, `dmb.rr`, `dmb.wr`, `dmb.ww`. All others are just "macros": combinations of these. For example: ARMv8's `dmb.ld` = `dmb.rw`; `dmb.rr`. AQ is for strong read acquire, AQpc for the weak read acquire, RL for the strong write release, RLpc for the weak write release.

```
let obs = rfe | fr | co
let dob  = addr | data
         | (addr | data); rfi
         | (ctrl | (addr; po)); [W]
         | (ctrl | (addr; po)); [isb]; po; [R]
let aob = [range(rmw)]; rfi;
(if arch = RISC-V then [R] else [AQ|AQpc])
let bob  = [R]; po; [dmb.rr]; po; [R]
         | [R]; po; [dmb.rw]; po; [W]
         | [W]; po; [dmb.wr]; po; [R]
         | [W]; po; [dmb.ww]; po; [W]
         | [RL]; po; [AQ]
         | [AQ|AQpc]; po
         | po; [RL|RLpc]
         | if arch = RISC-V then rmw
let ob = obs | dob | aob | bob
acyclic po-loc | fr | co | rf as internal
acyclic ob as external
empty rmw & (fre; coe) as atomic
```

**Figure 6.** ARMv8 and RISC-V axiomatic memory models

The revised ARMv8 has an official axiomatic concurrency model, written in herd [11], by Will Deacon [20]. RISC-V has an axiomatic model closely following ARM's, produced by the RISC-V Memory Model Task Group, chaired by Daniel Lustig. The models work in a two-step process. The models first enumerate the set of all *candidate executions*. Each candidate execution is one potential full execution of the program, specified by relations on its memory accesses $\langle po, co, rf, rmw\rangle$.

- *po* (program order) is a control flow unfolding of the threads of the program.
- *co* is the coherence order, the sequencing of writes to the same address in memory.
- *rf* is the reads-from relation, relating a write access $w$ with a read access $r$ that reads from $w$.
- *rmw* relates a read and write access of successfully paired load and store exclusive instructions.

In the second step the model checks each candidate execution for whether it satisfies its *axioms*, and only allows such *legal* executions that do. Typically the axioms require the acyclicity of certain relations of the full candidate executions.

In ARMv8 there are three axioms: a standard coherence axiom and an axiom concerning the atomicity guarantees of load/store exclusive instructions, and the "main" axiom. For the main axiom, the relation obs describes the interaction between memory accesses of different threads (using reads-from and coherence), and the relation ob describes the thread-local ordering due to dependencies and barriers every execution must preserve. The main axiom requires that the interaction between threads is compatible with this thread-internal ordering, by requiring the acyclicity of the relations. For RISC-V, the axiomatic herd model [42] is similar. The proof currently assumes known simplifications of the axiomatic models to unify them in the Coq formalisation. We call this model Axiomatic for both cases, ARM or RISC-V (see supplementary material for the definitions).

We now define two variants of the Promising semantics. To this end, first define a *valid execution* as an execution in which the threads in the final state have no outstanding promises. Then we call Promising the model as defined in §A.3 accepting only such valid executions. Second, as an intermediate model for the proof, we define Global-Promising to be the same as Promising, except where machine-step requires no certification. These Promising model variants are equivalent. Moreover Promising for RISC-V has no deadlocks (*i.e.* every execution is valid).

The following statements all assume finite executions.

**Theorem D.1.** *For a program $p$, $\vec{R}$ is a final register state of a legal candidate execution of $p$ in Axiomatic if and only if it is that of a valid execution of $p$ in Global-Promising.*

*Proof.* Proved in Coq (for both ARM and RISC-V). □

**Theorem D.2.** *For a program $p$, $\vec{R}$ is a final register state of a valid execution of $p$ in Global-Promising if and only if it is that of a valid execution of $p$ in Promising.*

*Proof.* Proved in Coq (for both ARM and RISC-V). □

**Theorem D.3** (Dead-lock freedom for RISC-V)**.** *For every certified state in Promising for RISC-V, either it is a final state with no outstanding promise, or there exists a step to another certified state.*

Proved in Coq.

## E Full evaluation results

| *Test* | *Promising* | *Flat* |
|---|---:|---:|
| SLA-1 | 0.27 | 0.41 |
| SLA-2 | 0.30 | 3.38 |
| SLA-3 | 0.33 | 21.57 |
| SLA-4 | 0.39 | 110.18 |
| SLA-5 | 0.44 | 526.76 |
| SLA-6 | 0.52 | 2277.72 |
| SLA-7 | 0.61 | 9108.53 |
| SLA-8 | 0.73 | ooT |
| SLA-9 | 0.86 | ooT |
| SLA-10 | 1.01 | ooT |
| SLC-1 | 3.21 | 8.63 |
| SLC-2 | 4.69 | 121.98 |
| SLC-3 | 6.58 | 1472.74 |
| SLR-1 | 2.47 | 3.70 |
| SLR-2 | 3.50 | 17.51 |
| SLR-3 | 4.88 | 52.52 |
| PCS-1-1 | 0.26 | 0.33 |
| PCS-2-2 | 0.40 | 10.33 |
| PCS-3-3 | 1.36 | 249.26 |
| PCM-1-1-1 | 0.30 | 23.58 |
| PCM-2-2-2 | 1.70 | ooT |
| PCM-3-3-3 | 71.12 | ooT |
| TL/(opt)-1 | 10.16 / 10.28 | 456.12 / 1180.33 |
| TL/(opt)-2 | 13.72 / 14.54 | 2202.12 / 7115.31 |
| TL/(opt)-3 | 18.08 / 20.13 | ooT / ooT |
| STC/(opt)-100-010-000 | 0.36 / 0.36 | 35.26 / 104.57 |
| STC/(opt)-100-010-010 | 0.42 / 0.42 | 2144.52 / 5943.50 |
| STC/(opt)-100-100-010 | 8.70 / 8.70 | ooT / ooT |
| STC/(opt)-110-011-000 | 7.64 / 8.13 | ooT / ooT |
| STC/(opt)-110-100-010 | 21.84 / 22.48 | ooT / ooT |
| STC/(opt)-200-020-000 | 7.16 / 7.12 | ooT / ooT |
| STC/(opt)-210-011-000 | 615.41 / 637.98 | ooT / ooT |
| STR-100-010-000 | 0.35 | 4.61 |
| STR-100-010-010 | 0.39 | 77.21 |
| STR-100-100-010 | 7.30 | 8940.03 |
| STR-110-011-000 | 6.55 | ooT |
| STR-110-100-010 | 18.09 | ooT |
| STR-200-020-000 | 5.80 | 11325.87 |
| STR-210-011-000 | 522.19 | ooT |
| DQ/(opt)-100-1-0 | 0.30 / 0.30 | 2.93 / 2.97 |
| DQ/(opt)-110-1-0 | 0.44 / 0.44 | 1042.88 / 1114.39 |
| DQ/(opt)-110-1-1 | 0.66 / 0.65 | ooT / ooT |
| DQ/(opt)-111-1-1 | 1.76 / 2.44 | ooT / ooT |
| DQ/(opt)-211-1-1 | 9.51 / 37.10 | ooT / ooT |
| DQ/(opt)-211-2-1 | 28.55 / 111.54 | ooT / ooT |
| QU/(opt)-100-000-000 | 1.34 / 2.95 | 2983.11 / ooT |
| QU/(opt)-100-010-000 | 2.55 / 5.66 | ooT / ooT |
| QU/(opt)-100-010-010 | 4.53 / 10.00 | ooT / ooT |
| QU/(opt)-100-100-010 | 712.57 / 4984.94 | ooT / ooT |
| QU/(opt)-110-011-000 | 589.50 / ooT | ooT / ooT |
| QU/(opt)-110-100-010 | 2108.12 / ooT | ooT / ooT |
| QU/(opt)-200-010-010 | 531.41 / ooT | ooT / ooT |
| QU/(opt)-200-020-000 | 286.99 / 10585.10 | ooT / ooT |

**Table 3.** Runtimes in seconds. ooT = more than four hours.