Promising-ARM/RISC-V: a simpler and faster operational concurrency model for ARMv8 and RISC-V

CHRISTOPHER PULTE, University of Cambridge, United Kingdom
JEAN PICHON-PHARABOD, University of Cambridge, United Kingdom
JEEHOON KANG, Seoul National University, Korea
SUNG-HWAN LEE, Seoul National University, Korea
CHUNG-KIL HUR*, Seoul National University, Korea

ARM have recently revised the ARMv8 architecture with two major simplifications: a shift to a syntactic definition of preserved dependencies and multi-copy-atomicity. Moreover, ARM has introduced an official, formal memory model. This axiomatic memory model was designed alongside and informed by an operational model with an "abstract microarchitectural flavour", to relate to hardware implementations and enable its validation. Both these two memory models serve their purposes for specification and validation. In both of them, however, execution order is divorced from program order, making them difficult to understand. In particular, in the operational model instructions execute out-of-order, non-atomically, and speculatively, and the model needs to sometimes discard or restart already-executed instructions.

In this paper, we introduce an equivalent simpler operational memory model for ARMv8. This model emphasises the execution of instructions in program order: instructions execute in order, without speculation, and atomically, with the exception of a notion of write promises. RISC-V has recently adopted a memory model closely following that of ARMv8. We extend our model to also handle RISC-V concurrency, in an even simpler model. Moreover, we integrate this model with ISA models for ARMv8 and RISC-V to obtain a more efficient tool for interactive or exhaustive exploration of small algorithm examples. With two simple optimisations it performs better than the two previous models for ARMv8.

The main results are formalised in Coq.

Additional Key Words and Phrases: Relaxed Memory Models, Operational Semantics, ARM, RISC-V

1 INTRODUCTION

After years of extensive research to clarify the semantics and develop precise models for the relaxed memory behaviour of POWER and ARM [3–6, 9, 10, 12–14, 21, 22, 25–28], ARM have recently revised their concurrency architecture with two major simplifications. First, whereas it previously introduced a semantic notion of "true" and "false" dependencies, it now considers syntactic dependencies. Second, writes are now *multi-copy-atomic*: when a write propagates from its thread to another, it also propagates to all other threads. Moreover, ARM now has an official formal memory model [7, Chapter B2][11]. This axiomatic model was co-designed with an equivalent operational model, Flat [25]. Both models are simpler than the operational or axiomatic models for the previous non-multi-copy-atomic ARM (and POWER) architectures. Recently, RISC-V has adopted a concurrency model closely following that of ARM, and has a similar axiomatic model, and an adaptation of Flat for an operational model.

The simplifications that these models present notwithstanding, we argue ARMv8 and RISC-V still lack a *simple programmer-focused operational* model. The ARMv8 and RISC-V axiomatic models are simple and concise. As axiomatic models, they directly specify properties of full executions and allow reasoning in terms of these global guarantees of the concurrency behaviour, but do not lend

*Hur is the corresponding author.

Authors' addresses: 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, Seoul National University, Korea, jeehoon.kang@sf.snu.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.

themselves well to reasoning locally in terms of program invariants. The axiomatic models first enumerate all full *candidate executions* of a given program and then discard those in which certain global relations on memory accesses are acyclic. Thus, they do not execute "step-by-step" and do not offer an obvious notion of a program state during execution.

Flat — taking advantage of the multi-copy-atomicity — simplifies previous operational ARM models by expressing the concurrency behaviour purely in terms of the threads' actions. The model has an "abstract microarchitectural flavour", offering a clearer relation to hardware implementations and helping its validation by ARM architects. The model, however, is complex, and to understand the possible concurrency behaviours of a program, it requires thinking in terms of abstractions of real hardware: in the model, instructions execute *out-of-order*, *speculatively* (the model has explicit branch speculation), and *non-atomically* (with several transitions per instruction), and the rules for the ordering of these transitions are intricate. In order to handle branch mispredictions, and coherence violations from speculatively executed instructions, the Flat model sometimes discards or restarts already-executed instructions.

Consider, for instance, the following example program. Here Thread 0 writes 37 to x, and 42 to y. The stores are separated by a strong barrier `dmb.sy`, keeping them ordered. Thread 1 reads y; if the value read is 42 it does a store and a load to some location z, and subsequently loads x. For the last load of x, computing the address involves the value returned from the load g of z (here using the calculation $x + r_1 - r_1$). Even though one might think, the chain of control flow and dataflow dependencies from d to h would keep them ordered, ARMv8 and RISC-V permit them to execute out of order and allows h to reading the initial memory value $x = 0$ after reading $y = 42$ with d.

Thread 0	Thread 1
(a) `store` [x] 37;	(d) r_0 := `load` [y]; // 42
(b) `dmb.sy`;	(e) `if` (r_0 = 42)
(c) `store` [y] 42	(f) `store` [z] 51;
	(g) r_1 := `load` [z]; // 51
	(h) r_2 := `load` [$x + (r_1 - r_1)$] // 0
	(i) `else` ...

The (only) Flat execution allowing this behaviour is the following: in the initial state, Thread 1 can speculate that e's condition may evaluate to "true", and fetch instructions f, g, and h; f cannot propagate to memory yet due to the unresolved conditional branch e, but it can take enough transitions to determine its address and value, making f available locally; g observes this state in which f is "half executed" and can read from f by local forwarding, and resolve the address of h; and h can read $x = 0$; now a can propagate to Thread 1; c can propagate to Thread 1; d can read $y = 42$ and determine that the branch speculation of e was correct, allowing f through h to finish; had e read differently and the speculation would have turned out incorrect, it would have discarded f through h and executed i.

In this paper we propose a concurrency model for ARMv8 and RISC-V in a different style altogether. The model combines the abstractness and conciseness of the axiomatic models with a clear operational intuition that does not aim to relate to hardware concepts but *emphasises the execution of threads in program-order*. The model has an abstract state and executes instructions *atomically* and *without branch speculation*, and with the exception of *write promises* executes instructions *in order*. As a result, the model does not need instruction restarts or discards for coherence violations or branch mispredictions and offers a simpler way of thinking about the concurrency behaviours. We return to how our model explains the above example later.

This model, called Promising-ARM/RISC-V, builds on the work of Kang et al. [16] on the Promising semantics for C/C++11 concurrency. In our model (1) loads execute in order; early load execution is captured by allowing reading from older writes in the *write history*. (2) Writes can be *promised*, capturing the out-of-order execution of stores, but every such promise is justified thread-locally and not subject to restarts or discards. (3) A concept of *views* uniformly provides

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

Fig. 1. The language, with ARMv8 and RISC-V memory accesses and barriers

coherence guarantees and the ordering resulting from dependencies or barriers. This model is equivalent to the axiomatic ARMv8 and RISC-V models.

Moreover, we produce an executable version, integrated with an ISA model for ARMv8 and RISC-V into the rmem tool of [12–14, 25, 27] that allows interactively stepping through the possible concurrent executions, pseudorandom exploration, and exhaustive enumeration of possible final outcomes for smaller examples. With two simple optimisations we achieve significantly better performance than Flat and ARM's axiomatic herd model on two ARMv8 spinlock examples.

To summarise, the contributions of this paper are:

- Promising-ARM, a simpler ARMv8 operational memory model (§§3,4). This model is formalised in Coq using a small language calculus.
- Promising-RISC-V, an operational memory model of RISC-V, a simple adaptation of Promising-ARM (§§3,4), formalised in Coq with the same calculus.
- An executable version of the models integrated with models for large parts of the ARMv8 [12] and RISC-V [24] user-mode ISA, into a tool for interactive, pseudorandom, and exhaustive exploration of small programs. With two simple optimisations, this tool achieves better performance than Flat and ARM's axiomatic model in herd (§6).
- A proof of equivalence of Promising-ARM and Promising-RISC-V and the ARMv8 and RISC-V axiomatic models. Coq mechanisation in progress, most important result mechanised (§8).
- A solution to the operational model deadlocks resulting from the relaxed semantics of ARMv8's store exclusive instructions, such as also present in Flat, for Promising-ARM. This extends the ARMv8 model and certification with locks (not yet covered by the Coq proof). (§7)

Caveats. Our model does not handle all aspects of ARMv8 and RISC-V: we do not model mixed-size accesses whose semantics is still being clarified; the model includes the well-established concurrency primitives of load/store exclusives (load reserve/store conditional in RISC-V) but not currently the similar atomic memory operations/AMOs. Finally, the ISA model only handles user-mode non-vector, non-floating-point instructions, systems features, or supervisor mode.[1]

We start with the definition of the sequential calculus and an informal illustration of the main aspects of the behaviour of Promising-ARM, before giving a precise definition of Promising-ARM and Promising-RISC-V.

2 LANGUAGE

To focus on the concurrency aspects we consider the small imperative language of Fig. 1; the executable tool of Section 6 handles user-mode parts of the real ARMv8 and RISC-V instruction semantics. A program is the parallel composition of statements. Statements include loads, stores,

[1]Currently the executable model's ISA semantics (not the concurrency model) is missing the weaker load acquire LDAPR introduced in ARMv8.3., due to time constraints.

barriers, register assignment, sequential composition, conditionals, and loops. Statements operate on thread-local 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) with a read or write kind, respectively indicating whether it is a plain access or has special acquire or release ordering. A store exclusive writes a bit to a register indicating whether it succeeds or fails. For uniformity of the syntax and the rules, we make non-exclusive stores also write the success bit, to an otherwise unused register, and omit it in the syntax. Following the ARM ISA, success is indicated by 0 (here called v_{succ}), and failure by 1 (v_{fail}). Only store exclusives can fail; non-exclusive stores always succeed. Whenever a load or store command is not annotated with a memory kind, we mean plain loads and stores; whenever it is not annotated to be exclusive, we assume it is non-exclusive. So r := `load` $[e]$ is a plain, non-exclusive load, and `store` $[e_1]$ e_2 is a plain, non-exclusive store.[2]

We treat '(`if` (e) s_1 s_2); s_3' as equivalent to '`if` (e) $(s_1; s_3)$ $(s_2; s_3)$': control flow is not delimited in assembly programs, and so in our language the instructions in s_3 are control-dependent on expression e.[3] As usual, our executable model necessarily bounds loops.

3 PROMISING-ARM AND PROMISING-RISC-V, INFORMALLY

ARMv8 and RISC-V have relaxed memory models that allows the effects of certain processor optimisations to become observable in the programmer-visible concurrency behaviour. This includes the out-of-order and speculative execution of instructions (past unresolved conditional branches or computed jumps), thread-internal write forwarding, and store queues and caches. At the same time, the architectures gives programmers certain guarantees such as coherence, ordering resulting from dataflow dependencies and special memory ordering instructions, and atomicity guarantees for load/store exclusive instructions. A concurrency model for these architectures therefore has to be able to handle the subtle semantics resulting from the looseness in the allowed concurrency behaviours while maintaining these architectural guarantees.

Promising-ARM/RISC-V does this using four main building blocks inspired by those of the promising semantics for C/C++11 of Kang et al. [16], but leveraged in a novel way:

(1) Memory is the full history of writes. To account for the the effects of the early execution of loads in ARMv8 and RISC-V while still executing programs in order, Promising-ARM/RISC-V allows loads to read from "old" writes in memory.
(2) To handle the early execution of stores, threads can *promise* writes at any time as long as they can later be *fulfilled* by a store instruction executed in program order.
(3) To provide the architectural ordering and coherence guarantees, Promising-ARM/RISC-V uses *views* to constrain both reads and writes.
(4) To ensure that all promises can be fulfilled, the promises of a thread are *certified* when they are introduced and at any step the thread takes.

For presentation purposes, we use ARMv8 terminology for barriers: `dmb.sy` for full barriers, etc. We start with simple programs using only plain loads and stores and full barriers. 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. Once we have introduced this core of the model, we describe how the other barriers and exclusive accesses fit in. Finally, we describe how certification avoids executions with unfulfilled promises.[4]

[2] RISC-V has a load-reserve strong acquire-release and store-conditional strong acquire-release, in which a single instruction has both strong acquire and strong release ordering combined. For simplicity of the presentation we omit this instruction, but the executable model handles this. We plan on adding this to the Coq formalisation as well.
[3] This matters for the memory ordering from control dependencies.
[4] Where for ARMv8 the certification will later be extended, in §7, as discussed.

3.1 View semantics

The view semantics underlying and Promising-ARM/RISC-V explains the effects of the out-of-order execution of reads — while executing programs in order — by recording the full write propagation history and allowing reads to read from older writes, and not just the last same-address write [19]. 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 the corresponding thread. Each thread consists of a statement (of type St), a register state, and more components which we will gradually introduce as we proceed.

Memory. Memory is a list of writes, in the order they were propagated in. A write (message) w, written $\langle x := v \rangle_{tid}$, records the location w.loc (here x), value w.val (v), and originating thread identifier w.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.

(*a*) `store` [x] 37; (*b*) `dmb.sy`; (*c*) `store` [y] 42 ‖ (*d*) r_1 := `load` [y]; // 42 (*e*) r_2 := `load` [x] // 0

$r_1 = 42 \wedge r_2 = 0$ allowed

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

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

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

Now d can read $y = 1$. Subsequently, since loads can read not only from the last same-address write, but also older writes in memory or the initial state, e is allowed to read the initial $x = 0$.

Views. Memory barriers restore stronger memory ordering. Placing a full barrier `dmb.sy` between the loads of Thread 1 orders them and prevents the behaviour where f reads 0 after d reads 42. Our model captures this ordering using *views*:

(*a*) `store` [x] 37; (*b*) `dmb.sy`; (*c*) `store` [y] 42 ‖ (*d*) r_1 := `load` [y]; // 42 (*e*) `dmb.sy`; (*f*) r_2 := `load` [x] // ≠ 0

$r_1 = 42 \wedge r_2 = 0$ forbidden

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 and indicates that the write at position v and its predecessors in the message list including the initial writes have been "seen".

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.

r3 Before executing a load or store i, its *pre-view* is computed, which constrains its execution. After executing i, its *post-view* is computed. Intuitively, the pre-view of i captures the maximal post-view of the operations that i depends on, and the post-view of i captures the ordering constraint that any operation depending on i should obey. The post-view of a store is the timestamp of its write message, which is always strictly greater than its pre-view. The post-view of a load is the maximum of its pre-view and a *read-view*. In the examples we consider first, the read-view is simply the timestamp of the write the load reads from; we later refine this to

handle thread-local *forwarding*. We will gradually introduce how the pre-view of loads and stores are 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-after the barrier are constrained by the post-views of those program-order-before the barrier. Intuitively, `dmb.sy` prevents reordering of any load or store before it and that after it, which is the strongest form of barrier. Other fences update these two views in a similar, but weaker way.

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

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

When executing f in the resulting state, f is constrained by view $\mathsf{v}_{\mathsf{rNew}} = 2$. Since all messages are seen with view 2, f must read the last message to location x, which is $\langle x := 37\rangle_0$.

Now, while `dmb.sy` provides strong ordering guarantees, it comes at a performance cost. Therefore, concurrent ARM programs also rely on memory ordering resulting from dataflow dependencies.

Address dependencies. The next example replaces the `dmb.sy` between the loads of Thread 1 with a *syntactic* address dependency from the first load (d) to the second (e). The register r_1 containing the return value of d is used in the computation "$x + (r_1 - r_1)$" of the address of load e, and this 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 cannot read 37.

(a) `store` $[x]$ 37;	(d) r_1 := `load` $[y]$; // 42
(b) `dmb.sy`;	(e) r_2 := `load` $[x + (r_1 - r_1)]$ // $\neq 0$
(c) `store` $[y]$ 42	

$r_1 = 42 \wedge r_2 = 0$ forbidden

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

- **r8** The register state $\mathsf{regs} : \mathsf{Reg} \to (\mathsf{Val} \times \mathbb{V})$ of a thread not only maps each register of the thread to a value, but also to an associated view (of type $\mathbb{V}$). We write $v@\nu$ for a value-view pair.
- **r9** When a register is written to by any instruction, it also updates the associated view to specify which writes have to have been seen in order to produce this value. Specifically, for any arithmetic instruction, the view of the output register is the maximum of the views of the input registers (that is, registers used by it). For a load, the output register view is its post-view (that is, the maximum of its pre-view and the read message's timestamp).
- **r10** Finally, the pre-view of a load or store instruction is the maximal view of its input registers (for loads those involved in the expression computing the address, for stores also that of the data) and the $\mathsf{v}_{\mathsf{rNew}}$ or $\mathsf{v}_{\mathsf{wNew}}$ view, respectively. This will be refined later to handle more dependencies and features such as *weaker barriers*, *release/acquire*, and *exclusives*.

Assuming the previous order a, b, c, when d reads $y = 42$, Thread 1 executes as follows:

$$\langle \mathsf{regs} = \{r_1 \mapsto 0@0, \ldots\}, \mathsf{v}_{\mathsf{rOld}} = 0, \mathsf{v}_{\mathsf{rNew}} = 0\rangle \overset{(d)}{\Rightarrow} \langle \mathsf{regs} = \{r_1 \mapsto 42@2, \ldots\}, \mathsf{v}_{\mathsf{rOld}} = 2, \mathsf{v}_{\mathsf{rNew}} = 0\rangle$$

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

Coherence. Accessing the same location multiple times also induces constraints. Consider the following example, which adds a later, independent, load f to x to Thread 1. While f is not ordered with d, the execution where d reads $y = 42$, e reads $x = 37$ and f reads $x = 0$ is forbidden, since it violates the principle of *coherence*: a is ordered after the implicit initial $x = 0$ in memory. So if e has read $x = 42$, the program-order later f must not read the coherence-superseded $x = 0$.

$$\begin{array}{l|l} (a)\ \texttt{store}\ [x]\ 37; & (d)\ r_1 := \texttt{load}\ [y];\ \text{// } 42 \\ (b)\ \texttt{dmb.sy}; & (e)\ r_2 := \texttt{load}\ [x + (r_1 - r_1)];\ \text{// } 37 \\ (c)\ \texttt{store}\ [y]\ 42 & (f)\ r_3 := \texttt{load}\ [x]\ \text{// } \neq 0 \end{array}$$
$$r_1 = 42 \wedge r_2 = 37 \wedge r_3 = 0 \text{ forbidden}$$

To account for the architectural coherence requirements:

r11 Each thread state maintains the coherence view $\mathsf{coh} : \mathsf{Loc} \to \mathbb{V}$. coh maps each location x to the maximal post-view of all loads and stores on x executed so far by that thread.

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 loads from the same location have to read from writes in coherence order, loads to the same location do not have to effectively happen in order. Consider the example below, where Thread 0 is unchanged, but where Thread 1 now contains an earlier read d from y, followed by a write e of 2 to y, before our 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 $y = 37$. While g is ordered after f by the address dependency, g is still allowed to read the initial $x = 0$. In ARMv8/RISC-V, 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.

$$\begin{array}{l|l} (a)\ \texttt{store}\ [x]\ 37; & (d)\ r_0 := \texttt{load}\ [y];\ \text{// } 42 \\ (b)\ \texttt{dmb.sy}; & (e)\ \texttt{store}\ [y]\ 51; \\ (c)\ \texttt{store}\ [y]\ 42 & (f)\ r_1 := \texttt{load}\ [y];\ \text{// } 51 \\ & (g)\ r_2 := \texttt{load}\ [x + (r_1 - r_1)]\ \text{// } 0 \end{array}$$
$$r_0 = 42 \wedge r_1 = 37 \wedge r_2 = 0 \text{ allowed}$$

For d to read $y = 42$, Promising-ARM/RISC-V must execute in the order a, b, c, d and then e by coherence, leading to memory $[1{:}\ \langle x := 37\rangle_0; 2{:}\ \langle y := 42\rangle_0; 3{:}\ \langle y := 51\rangle_1]$. If f now were to read $y = 51$ at timestamp 3 then its post-view would become 3, and so would the view of register r_1; this would not allow g to read the initial $x = 0$, since it would be constrained by pre-view 3 due to r_1.

To deal with this behaviour, each thread state records information about the thread's own writes, and the model's definition of the read-view specially handles the case in which a load reads from a write from the same thread to allow it to obtain a smaller post-view than the write's timestamp:

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, information about the last write to x propagated by this thread. Specifically:

r14 Every time a thread executes a store to x, it updates the forward bank entry $\mathsf{fwdb}(x)$ to record the timestamp of the written message (time), the maximal view of the store's input registers (view), and whether it was a write exclusive (xcl). Since a store's input registers are used to compute the address and data of the store, view captures its address and data dependencies.

r15 Initially, fwdb is set to $\langle \mathsf{time} = 0, \mathsf{view} = 0, \mathsf{xcl} = \texttt{false}\rangle$.

r16 The read-view of a load on some location x is refined as follows: if the read message's timestamp is the same as $\mathsf{fwdb}(x).\mathsf{time}$ (*i.e.* the load reads the last write at x by its thread), its read-view is the associated forward view $\mathsf{fwdb}(s).\mathsf{view}$. Otherwise, it is the read message's timestamp, as before. Since the post-view of a load is the maximum of its pre-view and read-view, this means

that when a load reads from a forwarded write, the post-view contains the address and data dependencies of the write instead of its timestamp. We explain the role of the xcl bit in §3.4.

In the example above, d reads $y = 42$ at timestamp 2, updating coh(y) to 2; e writes $y = 51$ at timestamp 3, updating coh(y) to 3 and fwdb(y) to $\langle \text{time} = 3, \text{view} = 0, \text{xcl} = \texttt{false} \rangle$, 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 fwdb(y).view = 0, thereby setting the view of r_1 to 0; finally, g reads the initial $x = 0$ with pre-view 0, since its sole input register, r_1, has view 0.

It is important to note that, as we have seen in this example, in general the coherence view 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.

3.2 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 before. As a motivating example, consider the following program. Here Thread 0 reads from x, and writes the value it reads to y; Thread 1 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.

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

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 prom : 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 any other write 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 should generate a write $x = v$ with the extra condition that its pre-view and the coherence view coh(x) are strictly smaller than the promise timestamp t.

r20 We split the execution of a write in a promise and its fulfilment. A normal write that is not executed early is accounted for by promising it just before a store fulfils it. Note that the timestamp of a write is always bigger than its pre-view because it is appended at the end with a fresh timestamp and immediately fulfilled.

The pre-view of a store essentially constrains promises by constraining the fulfilment: a promise cannot be made "too early", because it cannot be fulfilled if its timestamp is not strictly larger than its pre-view. With only this rule added to the underlying view semantics, in what follows, we show how this simple notion of promise captures the out-of-order execution of stores in ARMv8/RISC-V.

Out-of-order execution of writes. The behaviour in the example above that motivated promises is explained as follows. Thread 1 first promises the write $x = 42$ at timestamp 1, resulting in promise set prom = {1} and memory [1: $\langle x := 42 \rangle_1$]. Now, in Thread 0, a can read $x = 42$ from memory and write $y = 42$ (by a normal write), resulting in memory [1: $\langle x := 42 \rangle_1$, 2: $\langle y := 42 \rangle_0$]. Then, c can

read $y = 42$ from memory, and d can fulfil the promise $x = 42$ at timestamp 1, yielding prom = {}; d's pre-view and coh(x) are 0, so strictly smaller than the promise timestamp 1, as required.

Memory barriers. Placing a barrier on Thread 1 prevents the out-of-order execution of writes. Promising-ARM/RISC-V handles this using views. As before, consider the state after Thread 1 promising $x = 42$ and Thread 0 executing a, b, resulting in memory $[1: \langle x := 42\rangle_1, 2: \langle y := 42\rangle_0]$. Here, c is not allowed to read $y = 42$, because it would not be able to fulfil promise at timestamp 1. Suppose c does read $y = 42$ at timestamp 2. Then Thread 1 has $\mathsf{v}_{\mathsf{rOld}} = 2$ after c and $\mathsf{v}_{\mathsf{wNew}} = 2$ by `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.

(a) r_1 := `load` $[x]$; // 42	(c) r_2 := `load` $[y]$; // $\neq 42$
(b) `store` $[y]$ r_1	(d) `dmb.sy`;
	(e) `store` $[x]$ 42

$r_1 = r_2 = 42$ forbidden

Coherence. Also, replacing d by r_3 := `load` $[x + (r_2 - r_2)]$ 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 coh(x) to its post-view 2 and e is constrained by coh(x) = $2 \not< 1$.

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

(a) r_1 := `load` $[x]$; // 42	(c) r_2 := `load` $[y]$; // $\neq 42$
(b) `store` $[y]$ r_1;	(d) `store` $[x + (r_2 - r_2)]$ 42

$r_1 = r_2 = 42$ forbidden

Control and address-po dependencies. While ARMv8/RISC-V allows executing loads 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) since the address resolution could trap and prevent the store. Changing d in the previous example to a conditional branch depending on c's return value also prevents promising e early: the behaviour in which both a and c read 42 is forbidden in the example below, due to the control flow dependency of the store on c.

(a) r_1 := `load` $[x]$; // 42	(c) r_2 := `load` $[y]$; // $\neq 42$
(b) `store` $[y]$ r_1	(d) `if` $((r_2 - r_2) = 0)$
	(e) `store` $[x]$ 42

$r_1 = r_2 = 42$ forbidden

To capture control and address-po dependencies, the model introduces another view $\mathsf{v}_{\mathsf{CAP}}$.

r21 Each thread state has an additional view $\mathsf{v}_{\mathsf{CAP}} : \mathbb{V}$, initially set to 0.

r22 Every time a thread executes a conditional branch, the maximal view of the branch's input registers is merged into $\mathsf{v}_{\mathsf{CAP}}$. Similarly, when a load or store is executed, the maximal view of the input registers used to compute the address is merged into $\mathsf{v}_{\mathsf{CAP}}$.

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 1, a reads $x = 42$, b writes $y = 42$ at timestamp 2, and c reads $y = 42$ thereby setting r_2's view to 2. Then d

merges r_2's view (*i.e.* 2) into $\mathsf{v}_{\mathsf{CAP}}$ since r_2 is used to compute the branch condition. In case d is `store` $[z + (r_2 - r_2)]$ 0, register r_2's view is also merged into $\mathsf{v}_{\mathsf{CAP}}$ since r_2 is used to compute the address. Then e's pre-view includes $\mathsf{v}_{\mathsf{CAP}} = 2$, and thus e cannot fulfil the promise at timestamp 1.

Replacing d by an address-dependent load or store to an otherwise unused memory location z (e.g. `store` $[z + (r_2 - r_2)]$ 0) introduces the same ordering and also forbids the behaviour.

3.3 Release/acquire accesses and weak barriers

In addition to the full `dmb.sy` barriers and dependencies, memory model ordering can also be created in ARMv8 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 $\mathtt{fence}_{\mathtt{RW,RW}}$, $\mathtt{fence}_{\mathtt{R,RW}}$, and $\mathtt{fence}_{\mathtt{W,W}}$, respectively. `isb` has no equivalent in RISC-V: `fence.i` does not consider control and address-po dependencies. Since we do not model self-modifying code, `fence.i` is a no-op in this model. RISC-V has some additional barriers, such as $\mathtt{fence}_{\mathtt{W,R}}$ and `fence.tso`, which work analogously to the ARMv8 barriers presented so far (see §4).

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 0 promises $y = 42$ before $x = 37$, at timestamp 1. Executing a places $x = 37$ at timestamp 2, and sets $\mathsf{v}_{\mathsf{wOld}} = 2$.

$$(a)\ \mathtt{store}\ [x]\ 37; \quad \| \quad (c)\ r_2 := \mathtt{load}_{\mathtt{acq}}\ [y];\ //\ 42$$
$$(b)\ \mathtt{store}_{\mathtt{rel}}\ [y]\ 42 \quad \| \quad (d)\ r_2 := \mathtt{load}\ [x]\ //\ \neq 0$$
$$r_1 = 42 \wedge r_2 = 0 \text{ forbidden}$$

r24 A store release includes in its pre-view the view of all previous memory accesses, captured by $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$.

Therefore, after a, the pre-view of b is 2, and it cannot fulfil the promise at timestamp 1. So, in the example, when c reads $y = 42$, memory must instead be $[1{:}\ \langle x := 37\rangle_0, 2{:}\ \langle y := 42\rangle_0]$ thereby setting c's post-view to 2.

r25 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:

r26 The thread state maintains a view $\mathsf{v}_{\mathsf{Rel}} : \mathbb{V}$ containing the maximal post-view of all strong releases executed so far.

r27 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:

r28 `dmb.st` updates $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{wOld}}$.
r29 `dmb.ld` updates $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{rOld}}$.
r30 `isb` updates $\mathsf{v}_{\mathsf{rNew}}$ to include $\mathsf{v}_{\mathsf{CAP}}$.

3.4 Load/store exclusive instructions

The previously discussed instructions can only introduce intra-thread ordering. Exclusive instructions (called load reserve/store conditional in RISC-V) make it possible to provide inter-thread atomicity guarantees. If a load exclusive a and a store exclusive b are paired 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).

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 $[1\colon \langle x := 37\rangle_1, 2\colon \langle x := 51\rangle_1, 3\colon \langle x := 42\rangle_0]$ (although different-address writes and non-exclusive writes to x from Thread 0 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.

$$\begin{array}{l|l} (a)\ r_1 := \mathtt{load}_{\mathsf{ex}}\ [x];\ //\ 37 & (c)\ \mathtt{store}\ [x]\ 37; \\ (b)\ r_2 := \mathtt{store}_{\mathsf{ex}}\ [x]\ 42 & (d)\ \mathtt{store}\ [x]\ 51; \\ & (e)\ r_3 := \mathtt{load}\ [x]\ //\ \neq 42 \end{array}$$
$$r_1 = 37 \wedge r_2 = \mathsf{v}_{\mathsf{succ}} \wedge r_3 = 42 \text{ forbidden}$$

To capture the pairing of load exclusives and store exclusives:

r31 Each thread maintains an *exclusives bank* xclb : option $\langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}\rangle$, initially set to *none*, containing information about the last load exclusive when there has been no other store exclusive in that thread since then. Specifically,
r32 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.
r33 xclb is set to *none* whenever a store exclusive (successful or not) is executed.

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

r34 A store exclusive to location z at timestamp t succeeds only if xclb is not *none* and, in case the message at xclb.time is also z (so the load exclusive was to the same location), every message to z in memory between xclb.time and t is written by this thread.
r35 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 §3.5, for which we present a solution in §7.

Specifically, in Thread 0, xclb.time is 1 and d should write $x = 42$ at timestamp 3, but then the write $x = 51$ in the middle is written by Thread 1, which violates the above rule. Thus in order for b to be successful, b should be executed before d resulting in memory $[1\colon \langle x := 37\rangle_1, 2\colon \langle x := 42\rangle_0, 3\colon \langle x := 51\rangle_1]$ after d. Then e is constrained by $\mathsf{coh}(x) = 3$, which is due to d, and thus should read 51.

In addition to this atomicity guarantee, exclusives provide some ordering guarantees: 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:

r36 Recall that the forward bank fwdb : Loc → option $[\mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}; \mathsf{xcl} : \mathbb{B}]$ records in the xcl field whether the write in the forward bank is an exclusive write. The model then prevents

a load acquire on ARM, and any load in RISC-V, from a location z from obtaining the smaller forward view fwdb(z).view if fwdb(z).xcl is set.

RISC-V additionally guarantees ordering of the store exclusive with the paired load-exclusive even if the load and the store are to different addresses.[5] To this end:

r37 Recall that the exclusives bank xclb : option ⟨time : $\mathbb{T}$; view : $\mathbb{V}$⟩ records the post-view of the load exclusive in the view field. In RISC-V, this view xclb.view is included in the paired store exclusive's pre-view.

3.5 Certification

In the description of promises so far (§3.2) we focused on how promises capture the effects of stores being executed early, and how the fulfilment of promises is constrained by the views of stores. However, we did not discuss how the model ensures threads do not take steps that lead to inconsistent states in which threads cannot fulfil their promises. In the remainder of this section, we describe how we use *certification* to prevent such executions. Certification works purely thread-locally: for any given thread, certification only considers the state of this thread and the current state of memory. The resulting operational models are equivalent to the axiomatic concurrency models of ARMv8 and RISC-V, and §8 discusses the mechanised Coq proof of equivalence between the ARMv8 axiomatic model and Promising-ARM/RISC-V. Moreover, we describe an algorithm that makes certification executable, enabling the executable tool of §6 that is sufficiently efficient for exhaustively checking the possible outcomes of small concurrent algorithm examples with bounded loops.

The simple certification definition given in this section is *sound* for ARMv8 and RISC-V: it does not prevent any valid executions. For RISC-V it is also *precise*: it prevents any steps leading to executions in which a thread fails to fulfil promises. In ARMv8, however, it is not precise: — as we will discuss in §7.2 — for ARMv8 the resulting model can deadlock. However, all such deadlocks are due to load/store exclusives: in ARMv8, syntactic dependencies on the register write of a store exclusive introduce no memory ordering; any such deadlock can be *blamed* on a thread making an assumption about the success of a store exclusive that later turns out to conflict with another thread's write. In §7.2, we extend the model with locks and the certification to take locking into account to achieve a deadlock-free model of ARMv8.[6]

3.5.1 Thread-local certification. We call a machine state *globally consistent* if there exists a legal execution (involving all threads) from this machine state in which all threads fulfil there promises. Thus, the purpose of certification is to prevent any thread steps leading to globally inconsistent machine states. Somewhat surprisingly maybe, this certification can be done purely thread-locally.[7]

r38 Let T be the thread's current state, M the current memory state. The *thread configuration* $\langle T, M\rangle$ is *certified* if there exists an execution from $\langle T, M\rangle$ leading to a final thread configuration $\langle T', M'\rangle$ such that T' has no outstanding promises.

The above definition of certification is based only on its current thread state and the memory, is sufficient to preserve global consistency of machine states, without preventing executions that lead to consistent machine states (and which therefore should be allowed). Since the threads communicate by reading and writing memory one might imagine certification of a thread having to take this interaction with other threads into account. Informally speaking, the reason this is not

[5] In the case of ARMv8 the architecture specifies "constrained unpredictable" behaviour; this is still being clarified.
[6] The mechanisation of the proof of the algorithmic definition and of the extended certification are currently in progress.
[7] again, with the exception of ARMv8 load/store exclusive instructions as discussed later

the case is that "one thread cannot break another thread's promises", and that "one thread cannot help another thread fulfil its promises".

"One thread cannot break another thread's promises". Consider the following program. In the initial state, Thread 0 is allowed to promise $y = 37$ at timestamp 1 (leading to memory $[1{:}\ \langle y := 37\rangle_0]$ and adding 1 to prom) since it is certified after the promise: there is an execution of Thread 0 alone under this memory in which a reads $x = 0$, b writes $x = 42$ at timestamp 2 and c fulfils the promise $y = 37$ at timestamp 1.

(a) r_1 := load $[x]$;	(d) store $[x]$ 51;
(b) store $[x]$ 42;	(e) store $[y]$ 63
(c) store $[y]$ $(r_1 + 37)$	

To see why this certification of Thread 0 cannot be broken by other threads, suppose, for example, that Thread 1 executes d and e, leading to memory $[1{:}\ \langle y := 37\rangle_0, 2{:}\ \langle x := 51\rangle_1, 3{:}\ \langle y := 63\rangle_1]$. With this memory Thread 0's state is still certifiable as before:

- Since all views in Thread 0's state are unaffected by Thread 1's execution, a can still read $x = 0$ at timestamp 0 in the same way as in the previous certification.
- While b has to write $x = 42$ at timestamp 4, a different timestamp from the previous certification, c can still fulfil the promise $y = 37$ at timestamp 1 because c's pre-view and coh(y) are still 0.

Since the views precisely track the dependencies of a store, the only writes a promise can depend on, must already be in memory at the time of promising (Otherwise, by view constraints in the initial certification, the promise fulfilment would have failed, disallowing the promise.) In the example, since b is not in memory at the time of c's promise, Thread 0's ability to fulfil c's promise is independent of the how b executes, and so of how b's views change during the certification due to the interference by other threads.

"One thread cannot help another thread fulfil its promises". Now we discuss the opposite: due to the way dependencies introduce memory ordering in ARMv8 and RISC-V a thread cannot be "helped" in fulfilling its promises. Consider another example where in Thread 0 a loads x, b writes to y what a read, while on Thread 1 c reads y and if the value read is 42 e writes 42 to y. Consider a promise $x = 42$ from Thread 1 in the initial state. Certification does not allow this promise: e can only execute and produce this write if c reads $y = 42$. But in the initial memory there is no such write $y = 42$.

(a) r_1 := load $[x]$;	(c) r_2 := load $[y]$;
(b) store $[y]$ r_1	(d) if $(r_2 = 42)$
	(e) store $[x]$ 42;

This example program might be taken to suggest the above certification definition was insufficient, since the following *hypothetical* execution could allow Thread 1 to fulfil the promise $x = 42$ with the help of Thread 0, leading to outcome $r_1 = r_2 = 42$: first promise $x = 42$ in Thread 1, then read $x = 42$ with a and write $y = 42$ with b in Thread 0, and finally in Thread 1 read $y = 42$ with c satisfying d's branch condition and allowing e to fulfil the promise — and thereby Thread 0 "helping" Thread 1 fulfil its promise and implying a more sophisticated certification may be necessary.

Since, however, in ARMv8 and RISC-V dependencies create memory ordering, behaviours of this kind are not possible. In this particular hypothetical execution, Thread 1's promise of $x = 42$ in the initial state leads to memory $[1{:}\ \langle x := 42\rangle_1]$ and prom $= \{1\}$; executing b afterwards as above to memory $[1{:}\ \langle x := 42\rangle_1; 2{:}\ \langle y := 42\rangle_0]$; now when c reads $y = 42$ from this second write it updates r_2's view to its post-view 2; d's branch condition depends on r_2 and so d updates $\mathsf{v}_{\mathsf{CAP}}$ to 2; since in the next step e's pre-view is constrained by $\mathsf{v}_{\mathsf{CAP}} = 2$ Thread 1 cannot fulfil promise $1{:}\ \langle x := 42\rangle_1$.

More generally, when some thread i promises a new write p, the fulfilment of p cannot depend on another thread: any write to memory w by another thread after the promise of p must have a

timestamp greater than p. Since p's fulfilment cannot depend on any write with timestamp greater than p, no such write w can "help" in fulfilling p.

3.5.2 Algorithm. In the following, we describe an *algorithm* implementing the above certification definition in a function that enumerates the legal next (promise or non-promise) steps of a thread. The algorithm makes the model executable and is the basis for the definition used in the rmem tool for ARMv8 and RISC-V discussed in §6. 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 §4 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).

To illustrate the algorithm, consider the following (partial) program. Assume that the memory is $[1{:}\ \langle w := 1\rangle_1, 2{:}\ \langle z := 1\rangle_0]$, that the promise set of Thread 0 is $\mathrm{prom} = \{2\}$, and that Thread 0 has not yet executed a. The certification algorithm first enumerates all traces of Thread 0 leading to states in which all its promises have been fulfilled. Here, there is only one such trace:

$$\begin{array}{l} (a)\ r_1 := \texttt{load}\ [w]; \\ (b)\ \texttt{store}\ [x]\ 1; \\ (c)\ \texttt{store}_{\texttt{rel}}\ [y]\ 1; \\ (d)\ \texttt{store}\ [z]\ r_1 \end{array} \quad \Big\| \quad \ldots$$

- a reads 1 from w: otherwise d would write $z = 0$, which cannot fulfil the outstanding promise $2{:}\ \langle z := 1\rangle_0$.
- 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 $\mathrm{v_{wOld}}$.
- And d fulfils $2{:}\ \langle z := 1\rangle_0$.

Therefore:

(1) The next-instruction step in which a reads 1 from w is a certified step for Thread 0. (The one where a reads 0 is not, due to the outstanding promise $2{:}\ \langle z := 1\rangle_0$.)

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

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

$$ts \in \text{TState} \stackrel{def}{=} \left\langle \begin{array}{l} \text{prom} : \text{set } \mathbb{T}; \quad \text{regs} : \text{Reg} \to \text{Val} \times \mathbb{V}; \\ \nu_{\text{rOld}}, \nu_{\text{wOld}}, \nu_{\text{rNew}}, \nu_{\text{wNew}}, \nu_{\text{CAP}}, \nu_{\text{Rel}} : \mathbb{V}; \\ \text{fwdb} : \text{Loc} \to \langle \text{time} : \mathbb{T}; \text{view} : \mathbb{V}; \text{xcl} : \mathbb{B} \rangle; \\ \text{xclb} : \text{option} \langle \text{time} : \mathbb{T}; \text{view} : \mathbb{V} \rangle \end{array} \right\rangle$$

$$T \in \text{Thread} \stackrel{def}{=} \text{St} \times \text{TState} \qquad \vec{T} \in \text{TPool} \stackrel{def}{=} \text{TId} \to \text{Thread} \qquad \langle \vec{T}, M \rangle \in \text{Machine} \stackrel{def}{=} \text{TPool} \times \text{Memory}$$

Fig. 2. Types in the semantics

(2) Promising the write $x = 1$ at timestamp 3 is also certified: it is a possible write by Thread 0 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 $3 \not\leq 2$.

4 THE MODEL, FORMALLY

Fig. 2 summarises the types used by the model that were introduced in §3. 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. 3 gives the formal definition of the steps of the semantics, cross-referenced with the relevant rules in §3. First, we define auxiliary definitions.

- The interpretation function for expressions (second and third line) takes an expression and a register state m, and returns the expression's value and view. Constants have view 0; registers are looked up in m; the view for an arithmetic expression merges the views of the arguments (**r9**).
- read(M, l, t) gives the result of reading location l at timestamp t in memory M. For $t = 0$, this is the initial value v_{init}, here 0; otherwise either the value of the message in M at timestamp t if its location is l, otherwise *none*.
- read-view(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**, **r36**).
- not-blocked(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 (**r31**, **r32**, **r34**).
- Now, we define thread-local steps $T, M \to_{a,tid} T'$, which do not change the memory.
  exclusive-failure A store exclusive that has not been executed is always allowed to fail. It sets r_{succ} to ν_{fail} (here 1) to signal failure, with 0 timestamp, and sets xclb to *none* (**r33**).
- fulfil starts with the pre-condition (from top to bottom). First evaluate address and data expressions. Rule **r34** 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**, **r24**, **r37** 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**). **r35** explains the view ν_{succ} placed on the register write indicating the success. The post-condition removes the promise (**r19**); writes ν_{succ} (here 0) to the "success register" (**r35**); and updates the coherence view to include t (**r11**), certain views (**r5**, **r22**, **r26**); the forward bank (**r14**, **r36**); and the exclusives bank (**r31**, **r33**).

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

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

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

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

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

$\mathrm{atomic}(M, l, tid, t_r, t_w) \stackrel{def}{=} M(t_r).\mathrm{loc} = l \implies \forall t'.\ (t_r < t' < t_w \wedge M(t').\mathrm{loc} = l) \implies M(t').\mathrm{tid} = tid$

$\boxed{T, M \to_{a,tid} T'}$

(EXCLUSIVE-FAILURE)

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

(READ)

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

(FULFIL)

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

(FENCE)

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

(REGISTER)

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

(BRANCH)

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

(ISB)

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

(SKIP)

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

(SEQ)

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

(WHILE)

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

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

(EXECUTE)

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

(PROMISE)

$$\frac{w.\mathrm{tid} = tid \qquad ts' = ts[\mathrm{prom} \mapsto ts.\mathrm{prom} \cup \{|M| + 1\}]}{\langle \langle s, ts\rangle, M\rangle \to_{a,tid} \langle \langle s, ts'\rangle, M +\!\!+ [w]\rangle}$$

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

(MACHINE-STEP)

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

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

Fig. 3. Thread-local steps, thread steps, and machine steps

- 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**, **r27**. 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**, **r25**, **r22**; and exclusives bank as in **r32**.
- 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 $\texttt{dmb.sy} = \texttt{fence}_{RW,RW}$, its load barrier $\texttt{dmb.ld} = \texttt{fence}_{R,RW}$, its store barrier $\texttt{dmb.st} = \texttt{fence}_{W,W}$, and moreover RISC-V's "TSO fence" as $\texttt{fence.tso} = \texttt{fence}_{R,R}; \texttt{fence}_{RW,W}$. With these definitions, the behaviour of the ARM barriers is as explained with rules **r5**, **r6** **r7**, **r28**, **r29**.
- 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 expression, branches as determined by this value, and updates $\mathsf{v}_{\mathsf{CAP}}$ (**r22**).
- ISB. Executes an `isb` by merging $\mathsf{v}_{\mathsf{CAP}}$ into $\mathsf{v}_{\mathsf{rNew}}$ (**r30**).
- SKIP, SEQ, and WHILE. Mostly as expected. `while` is expressed using a branch.

We can then define thread steps $\langle T, M\rangle \rightarrow_{a,tid} \langle T', M'\rangle$. EXECUTE lifts a thread-local step that does not change memory to a thread step. PROMISE allows promising any write message to be promised, appending the 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 (**r38**).

5 EXAMPLE FROM INTRODUCTION

After having introduced PROMISING-ARM/RISC-V we now return to the original example of the introduction and show how the model allows the behaviour of interest. First Thread 0 executes a, b, and c, placing the writes into memory: $[1{:}\ \langle x := 37\rangle_0; 2{:}\ \langle y := 42\rangle_0]$. Now Thread 1 executes all instructions in program-order: d reads $y = 42$ at timestamp 2, annotating r_0 with view 2; when executing e its condition evaluates to "true" and updates $\mathsf{v}_{\mathsf{CAP}}$ to 2, due to the dependency on r_1; executing f updates memory to append 3: $\langle z := 51\rangle_1$ and sets $\text{fwdb}(z)$ to its write information: since f has no input register dependencies it records forward view 0; therefore when g reads $z = 51$ at timestamp 3 it obtains this forward view and annotates r_1 with view 0; finally, since h's pre-view only includes r_1's view not $\mathsf{v}_{\mathsf{CAP}}$, its pre-view is 0 and h is allowed to read the initial $x = 0$.

6 EXHAUSTIVE EXPLORATION

In §3, we showed an algorithmic definition for certifying thread steps and enumerating legal promise steps leading to an *executable* version of PROMISING-ARM and PROMISING-RISC-V. This section discusses this executable model that we integrated into the rmem tool with the user-mode ARMv8 and RISC-V Sail ISA models [13, 24].[8] The tool makes it possible to interactively step through executions of the model for ARMv8 and RISC-V ELF binaries [17] and litmus tests to explore their concurrency behaviours. Moreover, we show that, with two simple optimisations, our

[8] Unpublished; the RISC-V Sail model will be discussed elsewhere.

model significantly outperforms Flat and the ARM axiomatic model [11] specified in herd [6] in our comparison on two spinlock example algorithms.

6.1 Optimisation

In the following, we discuss two optimisations that improve the performance of exhaustive exploration and lead to promising early performance results in comparison with the Flat model and the herd ARMv8 axiomatic model. The first performance optimisation is based on two simple observations:

Observation 1: For every Promising-ARM trace in which a non-promise transition t is followed by a promise transition t', the trace that swaps t and t' is a valid equivalent trace. The informal reasoning is that the actions of the transitions commute, and doing t' before t at most "allows more behaviour": either t and t' are by the same thread or not. If they are from the same thread, then since t' is certifiable in the state after t it is also certifiable before t. Also, t' does not constrain t. If they are from different threads, then t' does not affect the thread state of t and the incremented memory only enables more behaviour in t's thread.

Observation 2: For every Promising-ARM trace any two consecutive non-promise transitions t and t' from different threads commute. t and t' do not change memory, only their respective thread states. Since both transition's pre and post-conditions depend only on memory and their thread states a trace that re-orders t and t' is valid, too.

The model uses these ideas to only explore traces that all follow a particular shape:

(1) The model starts in "promise-mode": in this state it allows only promise transitions, no other thread execution steps. In addition, the model allows non-deterministically leaving promise-mode with a stop-promising transition.
(2) Once there are no more promising transitions or the stop-promising transition is taken the model enters "non-promise-mode". In this mode it fixes an order of threads and fully executes each of the threads from start to end in program-order, without interleaving the thread actions. In "non-promise-mode" the model allows no more promises.

This reduces the model non-determinism to first enumerating possible "final memories" and then exploring the final thread states that are possible as a result of this memory.

In the description so far, the exploration, however, has an additional source of non-determinism: non-deterministically allowing to quit the promise-mode means the model can stop promising "too early": the execution of a thread may still involve a store whose write has not been promised yet. To avoid this, the executable model computes additional information during certification about whether it is possible to certify the thread state without propagating new writes. Then, the tool enables the stop-promising transition only if, for each thread, there is at least one possible way of certifying the thread state without propagating new writes, and in non-promise mode only explores such thread behaviours that do not produce new writes.

The tool uses a second optimisation to improve performance of non-promise-mode: after entering non-promise-mode, the tool runs the remaining thread actions without "normal" certification. Instead it uses a cheaper approximation to eagerly detect inconsistency (checking only that the next instruction step does not increase coh, v_{wNew}, or v_{CAP} to a value larger than or equal to an existing promise's timestamp) and discards deadlocking traces in which promises are not fulfilled. While in promise-mode, dropping certification would render the exploration computationally infeasible, in non-promise-mode, since threads are no longer interleaved, this improves the search complexity, since the certification of any thread's initial state under final memory already requires the effort of exploring the thread's possible final states.

6.2 Experimental validation and performance

The executable model incorporates the ideas of the formal model presented in this text. It is, however, not directly derived from the Coq formalisation, and differs in that it integrates the ARMv8 [12] and RISC-V [24] Sail ISA semantics. Therefore we validate the tool to confirm that it experimentally agrees with the outcomes allowed by the ARMv8 axiomatic model on a suite of around 6,500 litmus tests and a similar suite of around 7,000 RISC-V litmus tests.[9] Running the full suite of tests for ARMv8 takes around 84 minutes in runtime accumulated across CPU cores, compared to 171 minutes for the Flat model, on a POWER8 machine with 80 cores of 8 threads, 2.06 GHz each.

To compare performance of Promising-ARM to Flat and herd, we run two spinlock examples. The first is one of the two real-world tests used in demonstrating Flat [25]. (The other test from the paper, `spin_unlock_wait`, requires mixed-size memory accesses, which Promising-ARM does not currently support.) This test is an example taken from a Linux kernel spinlock implementation [15, 20], in a two-thread variant. We run Flat and Promising on a number of loop-unrolled versions of this spinlock example, transcribed as a litmus test (in supplementary material). herd does not support some of the arithmetic instructions. Both tools successfully verify the correctness of the loop unrolled versions, and discover the problematic behaviour of the bug-injected version the authors tested.

With the executable Promising-ARM model optimised as discussed above, the tool shows promising early performance results compared to the Flat model reporting the runtimes on a standard Linux Intel 2-core hyper-threading laptop at 2.5 GHz (see Tab. 1).[10] [11] These results indicate

unrolling #	1	2	3	4	5	6
Flat	0.50	3.75	24.04	123.27	606.48	2593.88
Promising	0.23	0.28	0.38	0.52	0.70	0.94

Table 1. Runtime in seconds per loop unrolling for the linux spinlock example (2-threads)

that in this example, Promising-ARM scales much better with increasing the loop unrollings. We speculate the reason for the difference in the asymptotic runtime behaviour is that this test particularly emphasises a performance advantage originating from the Promising-ARM model's more abstract design. In this two thread test there are four stores in total: a plain store, followed by a store exclusive executed in a loop, and another plain store; Thread 1's store is a store exclusive and it loops until it succeeds. There are several other loads and arithmetic, but in any final state the threads have propagated at most four writes (three from Thread 0 and one from Thread 1).

In the Flat model, each loop adds instructions to the Flat model's instruction tree. This includes loads with read transitions whose interleavings with other memory access transitions Flat must explore. Since states with a different number of loop unrollings differ, the tool must explore the traces enabled from those states (and cannot "prune them") even though these states could be regarded as equivalent to states with fewer unrollings.

In Promising-ARM, on the other hand, using the promise-first optimisation means the model will initially enumerate the possible "final memories". Since the promisable writes enabled in the initial state is independent of the number of loop unrollings (provided at least one unrolling), the

[9] Except for those that require atomic operations/AMOs.

[10] The Promising model bounds instruction steps for executability of infinite examples. For running these examples we set the loop bound to a value of 1,000, unreachable in these examples.

[11] The original spinlock example had a mixed-size aspect: although all writes to the same location were of the same size, one location was being accessed with loads of smaller sizes. We adapted the example to have no more mixed-size accesses and confirmed that performance numbers are not affected by this change.

non-determinism in this state does not increase when adding loop unrollings. The runtime increase in Promising-ARM is reduced to two (related) parts of the exploration: the additional cost in the certification of promises and promise enumeration in promise-only mode, and the exploration of the possible final thread states in non-promise-mode. Since both, however, only consider the execution of a single thread without interleaving with other-thread actions (in the latter case due to the non-promise-mode) the performance is less affected than in Flat where the interleaving of the memory actions of all threads' instructions of the unrolled program must be considered.

For a second performance test, we write a simple implementation of a standard spinlock example in C11, compile it to assembly, and run all three tools on this program transcribed as a litmus test in two-thread and three-thread versions and multiple loop unrollings (see supplementary material for the litmus tests). The resulting experimental numbers clearly show Promising-ARM scaling better for increased loop unrollings, although our model does not scale as well in this test as the first. These numbers from this performance comparison look promising, but more experimental

unrolling #	1	2	3	4	5	1	2	3
Flat	0.46	4.32	45.95	537.98	6443.87	3.77	219.79	8486.39
Herd	0.58	25.18	o.o.m	—	—	187.76	o.o.m	—
Promising	0.37	0.85	1.92	4.01	7.63	2.93	28.27	221.81

Table 2. Runtime in seconds per loop unrolling; 2-threads (left) and 3-threads (right); o.o.m = out of memory

validation is needed. In particular, while a realistic example of concurrent algorithms in practice, the first example's structure seems to emphasise the benefits of Promising-ARM.

For certain tests, tests with a large number of writes to otherwise unused locations (writes that are never read from) Promising does not perform well. In such tests, Promising-ARM explores all interleavings of the writes, and each interleaving leads to a different memory state and timestamps for the writes, even if the order of writes to those unused locations with respect to other writes does not affect the possible final outcomes. We plan to improve the handling of such cases. An obvious optimisation is to treat accesses to non-shared locations differently to reduce non-determinism. There are two obvious solutions to obtain the information about shared/non-shared locations: safely over-approximating the shared locations by abstract interpretation of the input program, or allowing for user-supplied information about the shared memory, and then checking and reporting violations of the user-supplied information. Our focus in this paper has been on establishing the semantics, and we leave further performance improvements, including these, for future work.

7 CERTIFICATION WITH ARMV8 STORE EXCLUSIVES

In §3.5 and §4 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 [25]. 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.

7.1 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 $r_1 := \texttt{load}_{\text{ex}}\ [x]; r_2 := \texttt{store}_{\text{ex}}\ [x]\ (r_1 + 1)$ as $r_1 := \texttt{fetch-and-add}\ [x]; r_2 := 0$. Now, r_2 still has to be set to indicate success (v_{succ}), but does not syntactically depend on the store. Therefore, 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 0 may (1.) execute a and read the initial value 0 (so $r_1 = 0$) and, (2.) assume the success of b (so $r_2 = \mathsf{v}_{\text{succ}} = 0$) and write $p = 1$ with c, *before b is in memory*: the architecture allows that d reads $p = 1$ and f reads $x = 0$ after the barrier e. (While in RISC-V the dependency from b to c means b propagates before c, forbidding this behaviour.)

$$
\begin{array}{l||l||l}
a : r_1 := \texttt{load}_{\text{ex}}\ [x]; & d : r_3 := \texttt{load}\ [p];\ \text{// } 1 & g : \texttt{store}\ [x]\ 2 \\
b : r_2 := \texttt{store}_{\text{ex}}\ [x]\ (r_1 + 1); & e : \texttt{dmb.sy}; & \\
c : \texttt{store}\ [p]\ (1 - r_1 - r_2) & f : r_4 := \texttt{load}\ [x]\ \text{// } 0 &
\end{array}
$$

$$r_3 = 1 \wedge r_4 = 0 \text{ allowed}$$

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 $r_1 := \texttt{load}_{\text{ex}}\ [x]; r_2 := \texttt{store}_{\text{ex}}\ [x]\ (r_1 + 1)$ with $r_1 := \texttt{fetch-and-add}\ [x]; r_2 := 0$ is easy. However, handling the dependency relaxation in its full generality (without deadlocks) is difficult.

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

The certification definition presented in §3 and 4 works thread-locally: it takes into account only a thread's state and current memory in order to decide whether a thread step should be allowed or not. But whether g's write should be allowed cannot be determined based only on the state of Thread 2 and on the list of messages in memory: it is Thread 0's promises due to which g must not write. So a precise certification algorithm for ARMv8 needs to take into account some information about other threads. In this example, Thread 0 effectively requires "locking" location x, to constrain the behaviour of the other threads. 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.

7.2 Extended certification, take 1

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

- Thread n depends on a load exclusive l to location x. This is if:
  - Thread n has already executed l, or
  - Thread n has an outstanding promise whose fulfilment depends on l due to register dataflow or due to coherence or view requirements.
- And Thread n relies on the success of the write of a store exclusive s to x that is paired with l: Thread n has an outstanding promise whose fulfilment depends on s via register dataflow.
- And the write w' that l read from is already in memory, whereas the write of s is not.

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

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

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

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

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

The taint tracking introduces complexity to the certification. Importantly however, during "normal" execution the extended model's views are simple timestamps just as before (§4), and taints are not persistent but local to the certification.

7.3 Extended certification, take 2

As the following example illustrates, unfortunately the ideas on certification above are still insufficient. In this example Thread 0's store d depends on the load exclusive b to x, and the "success register write" of the store exclusive c to location x. New here is that c has release ordering, and

so c is ordered after the write a to y. Symmetrically in Thread 1 h depends on the load and store exclusive instructions f and g to y, and g is a store exclusive release ordered after a store e to x.

$$\begin{array}{l|l} a : \texttt{store}\ [y]\ 1; & e : \texttt{store}\ [x]\ 1; \\ b : r_1 := \texttt{load}_{\texttt{ex}}\ [x]; & f : r_3 := \texttt{load}_{\texttt{ex}}\ [y]; \\ c : r_2 := \texttt{store}_{\texttt{ex,rel}}\ [x]\ 1; & g : r_4 := \texttt{store}_{\texttt{ex,rel}}\ [y]\ 1; \\ d : \texttt{store}\ [p]\ (1 - r_1 - r_2) & h : \texttt{store}\ [q]\ (1 - r_3 - r_4) \end{array}$$

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

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

- For the promise $p = 1$ the certification returns information of the form $([y]; x)$, meaning that for this step Thread 0 requires a lock on x, and that it *relies* on being able to write to y before releasing the lock on x. Promising $p = 1$ adds this to the memory's lock state.
- Since there are no other locks, Thread 0 can promise.
- In the following state, for the promise $q = 1$ by Thread 1, symmetrically, the certification returns the information $([x], y)$.
- Now $([x], y)$ is incompatible with $([y], x)$ due to the cyclic rely-guarantee dependency. Thus Thread 1 is not allowed to promise $q = 1$.

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.

8 EQUIVALENCE WITH THE REFERENCE AXIOMATIC MEMORY MODEL

The revised ARMv8 has an official axiomatic concurrency model, written in herd [6], by Will Deacon [11]. RISC-V has an axiomatic model closely following ARM's, produced by the RISC-V Memory Model Task Group. 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 [29] 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 several 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 §4 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. Finally, we define Extended-Promising, only for ARMv8, to be the same as Promising, except machine-step requires the extended certification as described in §7. All these Promising model variants are equivalent. Moreover Promising for RISC-V and Extended-Promising for ARMv8 have no dead locks (*i.e.* every execution is valid).

Theorem 8.1. *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. Fully mechanised for both ARM and RISC-V in Coq. □

Theorem 8.2. *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. Coq proof for both ARM and RISC-V in progress. □

Theorem 8.3 (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.*

We plan to formalise this theorem, which we think is straightforward.

Theorem 8.4 (ARM only). *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 Extended-Promising.*

The formalisation is in progress; the remaining proof is mostly technical.

Theorem 8.5 (Dead-lock freedom for ARMv8). *For every certified state in Extended-Promising for ARMv8, either it is a final state with no outstanding promise, or there exists a step to another certified state.*

We are planning to formalise this theorem, which seems technical and non-trivial but feasible.

9 RELATED WORK

Models. The reference axiomatic memory model and the operational Flat model of ARMv8, have already been discussed. The main inspiration for Promising-ARM is the promising semantics of Kang et al. for C/C++11 [16]. As described in the introduction, Promising-ARM relies on the same building blocks: 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 multi-copy-atomicity. (2) Promising C11 uses *future-memory quantification* for certification in order to capture the semantics dependencies. Here, dependencies are purely syntactic, captured by associating timestamps to registers. (3) Messages do not carry views, and barriers work purely thread-locally. (4) C11's semantic notion of dependencies, and its non-multi-copy-atomicity make exhaustive execution of Promising C11 hard.

The I^2E models aim to provide simple concurrency specifications in which instructions execute in order and atomically [8, 30, 31]. However, I^2E does not allow load-store reordering, allowed in both ARMv8 and RISC-V.

Model checking. The executability of Promising-ARM makes it possible to perform simple model checking by exhaustive state exploration for small programs, even in the presence of load buffering and load/store exclusives, with significant performance improvements over Flat memory model and the axiomatic memory model as executed by herd. The CDSChecker [23] tackles model checking for a variant of C/C++11 featuring some degree load buffering. CDSChecker interactively discovers the "future values" enabled by load buffering, and injects them by replaying executions with these new values. This is different from how our model handles out-of-order execution: their consistency check is based on the global C/C++11 consistency predicate, and they allow load buffering in the presence of syntactic dependencies, which means that they allow more behaviour (and do not track dependencies). The checker is neither sound nor precise.

Abdulla et al. [1, 2] describe efficient model-checking algorithms for hardware memory models (TSO, PSO, and Power — load buffering is architecturally allowed in the latter) by reducing bounded reachability under these models to bounded reachability under sequential consistency. To our knowledge, their tool provides an unsound approximate analysis only. Also, they do not tackle the Power ISA, but a simple calculus that does not feature load/store exclusives, which, as we described, are a major complication for ARMv8. Kokologiannakis et al. [18] describe an efficient model-checking algorithm for a variant of C/C++11 which is non-multi-copy-atomic but forbids load buffering — as opposed to ARM's multi-copy-atomic semantics with load buffering.

10 CONCLUSION AND FUTURE WORK

We have presented operational concurrency models, Promising-ARM/RISC-V, in a different style from existing architecture concurrency models. The models aim to explain the concurrency models in a simpler and more abstract way, emphasising the thread-local execution, using the concepts of views and promises from the Promising C11 semantics. The early performance results show significant performance improvements over Flat and the herd axiomatic model.

In the future we plan to do more serious performance optimisation, both by improvements enabled by the model's abstractness and applying standard model checking techniques. With some user-interface engineering we plan to also make the model more practical for use in interactive debugging concurrent programs, exploiting the advantages of Promising-ARM/RISC-V: the model has an abstract state and enumerates only valid steps, without instruction discards or restarts due to mis-speculation. Moreover, we believe Promising-ARM/RISC-V can serve as a basis for formal reasoning about concurrent software above the architecture.

ACKNOWLEDGMENTS

We are grateful to Peter Sewell. We thank Shaked Flur for interesting discussions. 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 Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2017R1A2B2007512).

REFERENCES

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. *Acta Inf.*, 54(8):789–818, 2017.
[2] P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo. Context-bounded analysis for POWER. In *TACAS*, pages 56–74, 2017.
[3] A. Adir, H. Attiya, and G. Shurekm. Information-flow models for shared memory with an application to the PowerPC architecture. In *IEEE Trans. Parallel Distrib. Syst.14, 5*, pages 502–51, 2003.
[4] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP*, 2009.
[5] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. litmus: Running tests against hardware. In *TACAS*, 2011.
[6] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *PLDI*, 2014.
[7] ARM. *ARM Architecture Reference Manual.* 2018.
[8] A. Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. *SIGARCH Comput. Archit. News*, 34(2):29–40, May 2006.
[9] N. Chong and S. Ishtiaq. Reasoning about the arm weakly consistent memory model. In *MSPC*, 2008.
[10] F. Corella, J. M. Stone, and C. M. Barton. Technical report RC18638: A formal specification of the PowerPC shared memory architecture. Technical report, 1993.
[11] W. Deacon. The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat, 2016.
[12] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL*, 2016.
[13] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *POPL*, pages 429–442, Jan. 2017.
[14] K. E. Gray, G. Kerneis, D. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *MICRO*, Dec. 2015.
[15] D. Howells, P. E. McKenney, W. Deacon, and P. Zijlstra. Documentation/memory-barriers.txt. https://www.kernel.org/doc/Documentation/memory-barriers.txt, 2018.
[16] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, pages 175–189, 2017.
[17] S. Kell, D. P. Mulligan, and P. Sewell. The missing link: explaining ELF static linking, semantically. In *OOPSLA*, pages 607–623, 2016.
[18] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018.
[19] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *POPL*, pages 649–662, 2016.
[20] Linux contributors. Documentation/locking/spinlocks.txt. https://www.kernel.org/doc/Documentation/locking/spinlocks.txt, 2014.
[21] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors, 2012.
[22] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, 2012.
[23] B. Norris and B. Demsky. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.*, 38(3):10:1–10:51, 2016.
[24] R. Norton-Wright, S. Flur, and P. Mundkur. RISC-V ISA. 2018.
[25] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. In *POPL*, 2018.
[26] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI*, 2012.

[27] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, June 2011.
[28] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, 2009.
[29] A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. 2018.
[30] S. Zhang, Arvind, and M. Vijayaraghavan. Taming weak memory models. *arXiv preprint arXiv:1606.05416*, 2016.
[31] S. Zhang, M. Vijayaraghavan, and Arvind. Weak memory models: Balancing definitional simplicity and implementation flexibility. In *PACT*, 2017.