Promising Compilation to ARMv8 POP[*]

Anton Podkopaev[1], Ori Lahav[2], and Viktor Vafeiadis[2]

1   a.podkopaev@2009.spbu.ru, SPbU, JetBrains Research, St. Petersburg, Russia
2   {orilahav,viktor}@mpi-sws.org, MPI-SWS, Kaiserslautern, Germany

─────── **Abstract** ───────

We prove the correctness of compilation of relaxed memory accesses and release-acquire fences from the "promising" semantics of Kang et al. [12] to the ARMv8 POP machine of Flur et al. [9]. The proof is highly non-trivial because both the ARMv8 POP and the promising semantics provide some extremely weak consistency guarantees for normal memory accesses; however, they do so in rather different ways. Our proof of compilation correctness to ARMv8 POP strengthens the results of the Kang et al., who only proved the correctness of compilation to x86-TSO and Power, which are much simpler in comparison to ARMv8 POP.

1   Introduction

One of the major unresolved topics in the semantics of programming languages has to do with giving semantics to concurrent shared-memory programs. While it is well understood that such programs cannot follow the naive paradigm of *sequential consistency* (SC) [15], it is not completely clear what the right semantics of such concurrent programs should be.

At the level of machine code, the semantics varies a lot depending on the hardware architecture, which is only loosely specified by the vendor manuals. In the last decade, academic researchers have produced formal models for the mainstream hardware architectures (e.g., x86-TSO [23], Power [22, 5], ARMv8 POP [9]) by engaging in discussions with hardware architects and subjecting existing hardware implementations to extensive tests.

In this paper, we will focus on the ARMv8 POP model due to Flur et al. [9], which is arguably the most advanced such hardware memory model.[1] Operational in nature, it models many low-level hardware features that affect the execution of concurrent programs. These include the hardware topology, the non-uniform propagation of messages to other processors, the reordering of messages, processor-level out-of-order instruction execution, branch prediction, local decisions on the coherence of overwritten writes, and so on. The ARMv8 POP model is in certain ways substantially weaker than other hardware memory models. For example, it allows the outcome $a = 1$ of the following program (see [13]):

$$
\begin{array}{l|l|l}
a := [x]; \ //1 & b := [x]; & c := [y]; \\
[x] := 1; & [y] := b; & [x] := c;
\end{array}
\qquad\text{(ARM-weak)}
$$

─────────────

[*] An extended version of this paper with a technical appendix can be found in [20].

[1] We would like to point out that the ARMv8 POP model is not the latest model for ARM. In March 2017, version 8.2 of the ARM reference manual [1] introduced a substantially stronger "*multi-copy-atomic*" model, whose formal axiomatic definition became available on 27 April 2017 [3]. The new model disallows the weak behaviors of the ARM-weak and WRC+data+addr examples discussed in this paper.

where all variables are initially 0. In essence, as we will explain in Section 2, the hardware may propagate the $[x] := 1$ store to the second thread, then write $[y] := 1$ and propagate it to the third thread, execute the third thread, and propagate its store to the first thread before the $a := [x]$ load returns. In contrast, Power and x86-TSO both forbid this outcome.

At the level of programming languages, the main problem is to design a memory model that enables efficient compilation across a wide range of hardware platforms and yet provides suitable high-level guarantees, such as reduction to SC in the absence of data races and type safety even in the presence of racy code. While many attempts to solve this problem have been made in the past [8, 19, 25, 11, 21], including the Java [18] and C/C++11 [7] memory models, almost all have been found to be lacking in one way or another, either not supporting certain compiler optimizations or allowing "out of thin air" behaviors.

Recently, however, Kang et al. [12] made a breakthrough and introduced a memory model that claims to satisfy both desiderata. Their model is also operational, but includes a rather non-standard step, according to which a thread can promise to perform a write in the future. While such promise steps are suitably restricted, once a promise is made, other threads can read from the promised write, even before the promise is fulfilled. Promises allow the weak behavior of the ARM-weak program: intuitively, the first thread may promise to write $[x] := 1$, the second and third threads may then execute writing 1 to y and x respectively, and the first thread can then execute reading $a = 1$ from the third thread and finally fulfilling its promise to write $[x] := 1$.

While the Promise machine allows this surprisingly weak behavior of the ARM-weak example, compilation from the promise semantics to the ARMv8 POP machine has not yet been shown to be sound. In their paper, Kang et al. mention the ARM-weak program, but do not verify compilation to ARMv8 POP; they only prove compilation correctness to the substantially simpler x86-TSO and Power models.

In this paper, we fill this gap and prove the correctness of compilation from a subset of the promise model to the ARMv8 POP model. The subset of promise model we handle is quite minimal—it contains relaxed loads and stores, as well as release and acquire fences—but exposes the following three main challenges we had to overcome in the compilation proof.

- Firstly, the two machines are very different. The ARM machine executes instructions possibly out of program order and in multiple steps: it issues the instruction, propagates it to one thread at a time, satisfies read instructions—all in different steps. In contrast, the Promise machine executes instructions in a single step and according to program order (except for promised writes).

- Secondly, the key technical device used in the compilation proof to the Power model is not applicable to the ARMv8 POP model because it can execute anti-dependent instructions out of order as in the ARM-weak program (see discussion in Section 10).

- Thirdly, although both memory models are operational, compilation correctness cannot be shown by a standard forward simulation. The reason is that in the ARM machine writes to a specific location are not necessarily totally ordered during the execution; they only become totally ordered once they are all propagated to the memory, which may happen at the very end of the execution. In the Promise machine, however, writes are totally ordered by timestamps from the point they are issued (or promised); so a simulation proof would have to "guess" the correct ordering of the writes.

To overcome this final challenge, we introduce an intermediate machine, which extends the ARM machine recording timestamps for each write, and views for each threads and message. We show that this intermediate machine has the same behaviors as the ARM machine, and that the Promise machine can simulate the intermediate machine's behaviors.

A secondary contribution of this paper is to provide a number of results about the ARMv8 POP model, which may be of general interest, e.g., in compiling from other language-level memory models to ARMv8 POP.

In the remainder of this paper, we first introduce the ARMv8 POP and Promise models informally (Section 2), and present the high-level structure of the proof (Section 3). Then, in Sections 4 to 9 we present the ARMv8 POP, intermediate, and Promise machines formally, and relate them to one another. We conclude with a discussion of related and future work.

2 Models through Examples

We start by discussing the ARM [9] and the Promise [12] machines on a couple of small programs, *litmus tests*, like this:

$$
\begin{array}{l|l}
[x] := 1; & a := [y]; \;\; //1 \\
[y] := 1; & b := [x]; \;\; //0
\end{array}
\tag{MP}
$$

Here a, b stand for local variables (registers) and x, y are distinct memory locations shared between threads. The program syntax of the ARM machine programs slightly differs from the syntax of the Promise machine programs. We apply the following compilation scheme:

Promise:	$[x]_{\mathsf{rlx}} := a$	$a := [x]_{\mathsf{rlx}}$	fence(acquire)	fence(release)
ARM:	$[x] := a$	$a := [x]$	dmb LD	dmb SY

As the compilation scheme is a bijection, we present programs in the ARM syntax only. For every program we suppose that locations are initialized with 0. To refer to a specific behavior of the program, we annotate read instructions with values expected to be read (e.g., //1).

The ARM Machine The ARM machine [9] consists of two components, *a thread subsystem* and *a storage subsystem*. Roughly speaking, the former corresponds to processors' per-thread control units [10], which fetch and execute instructions, and send store/load memory requests to the storage subsystem. The latter represents the memory hierarchy including caches, store/load buffers, and the main memory. A state of the storage subsystem can be represented graphically as a hierarchy of *buffers*, which are lists of memory requests.

Let's execute the MP program in the ARM machine and get $a = 1$ and $b = 0$. One way of getting this behavior is for the thread subsystem to issue the write (or read) requests *out-of-order* to the storage subsystem. However, there is another way in which the outcome $a = 1, b = 0$ is possible. First, the thread subsystem issues all requests in program order.
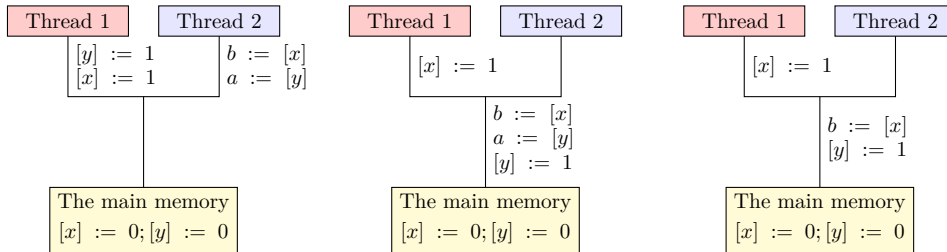
After that, the storage subsystem reorders the independent requests $[x] := 1$ and $[y] := 1$, and flows the requests $[y] := 1$, $a := [y]$, and $b := [x]$ from the bottom of the corresponding buffers to the common buffer. Once the read request $a := [y]$ follows $[y] := 1$ directly in a buffer, the storage subsystem is able to satisfy the read from the write and send a read

response, $a = 1$, to the thread subsystem. Finally, the storage subsystem flows $[y] := 1$ and $b := [x]$ to the main memory, satisfying the latter from the initial write $[x] := 0$.

Suppose that the outcome $a = 1, b = 0$ is undesirable. To outlaw it, one can put dmb SY, a *full fence*, between the writes[2] in the first thread and dmb LD, a *load fence*, between the reads in the second thread:

$$
\begin{array}{l|l}
[x] := 1; & a := [y]; \ \ //1 \\
\text{dmb SY}; & \text{dmb LD}; \\
[y] := 1; & b := [x]; \ \ //0 - \text{impossible}
\end{array}
\qquad \text{(MP-SY-LD)}
$$

The fence in the first thread forces the thread to issue $[x] := 1$, dmb SY, and $[y] := 1$ to the storage subsystem in order. Reordering of $[x] := 1$ and $[y] := 1$ in the storage subsystem is also impossible, as the request dmb SY is not reorderable with any request and stays between them. It guarantees that once $[y] := 1$ is propagated to the common buffer, $[x] := 1$ is propagated there as well. The fence dmb LD forbids to issue $b := [x]$ until $a := [y]$ is satisfied. So if $a := [y]$ is satisfied from $[y] := 1$, $[x] := 1$ is propagated to the common buffer or to the main memory, and $b := [x]$ can be satisfied only from it. So, if $a = 1$ then $b = 1$.

In the discussed executions, it is very important that the storage subsystem is able to reorder some requests but not others. The definition of the reordering relation $\hookrightarrow$ is following:

▶ **Definition.** A request e_{old} and a request e_{new} are *reorderable*, denoted $e_{\text{old}} \hookrightarrow e_{\text{new}}$, if neither of them is an SY fence and they operate on different locations.

In this paper, we consider a slightly weakened version of Flur et al.'s model [9], which issues dmb SY requests to the storage subsystem but not dmb LD requests. This allows more behaviors than the original model.[3] As we managed to prove compilation correctness to a weaker model, the result is also applicable to the original model.

The Promise Machine   The Promise machine [12] is very different from the ARM machine. There is no sophisticated storage subsystem. The *memory*, M, is simply a set of annotated writes, so-called *messages*, issued by all threads up to the moment. Each message has a *timestamp*, an element of a totally ordered set, which is unique among messages to the same location in the memory. Except for *promises*, which we discuss later in the section, the Promise machine executes instructions in program order. However, reads have a nondeterministic semantics: when a thread performs a read, it chooses any message from the memory subject to some conditions. The message has to be to the corresponding location and not to be too "old": if a thread has observed (i.e., read from) a message to location x with timestamp τ, it cannot subsequently read from a message to x with timestamp $\tau' < \tau$.

To enforce this restriction, as well as similar restrictions on timestamps of read messages that arise from the use of fences, the Promise machine uses so-called *views*—maps from locations to timestamps. Each message in the memory is annotated with a *message view*, and each thread maintains three views: $view_{\text{cur}}$, $view_{\text{acq}}$, and $view_{\text{rel}}$. The main thread view, $view_{\text{cur}}$, maps each location to the greatest timestamp of all messages of this location that were observed by the thread. For example, if a thread's $view_{\text{cur}}$ equals to $[z@\mathbf{2}, w@\mathbf{4}]$, it

---

[2]   One could equivalently put a store fence, dmb ST, but that does not correspond to anything in the promise model, as well as in the C/C++ model.

[3]   To show this, consider that dmb LD requests are issued, but are reorderable with any request. That would weaken the original semantics. But this is the same as not issuing the dmb LD requests at all.

means the thread has observed the write to the location w with the timestamp **4**. The other views— those included in messages, as well as the thread views $view_{\text{acq}}$ and $view_{\text{rel}}$— are used in combination with fences.

The weak behavior of MP is observable on the Promise machine quite easily. At the beginning of the execution, the memory contains only initial writes:

$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle\}$
$T1.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]$

The first thread $T1$ may perform the writes:

$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [x@\mathbf{0}]\rangle,$
$\qquad \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{0}, y@\mathbf{1}]\rangle\}$
$T1.view_{\text{cur}} = [x@\,\mathbf{1}\,, y@\,\mathbf{1}\,]; \quad T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]$

Now the second thread $T2$ can read from the newly added message $\langle y : 1@\mathbf{1}, [x@\mathbf{0}, y@\mathbf{1}]\rangle$ and the initial write $\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle$.

The Promise machine counterparts of the SY and LD fences, *release* and *acquire* fences correspondingly, are sufficient to outlaw $a = 1, b = 0$ as in the case of the ARM machine. The fence(release) in the first thread $T1$ enforces the message view of $[y] := 1$ to include the timestamp of $[x] := 1$, and if the second thread $T2$ reads from the message, then the fence(acquire) updates the second thread's $view_{\text{cur}}$ with the message view. Let's see how it works.

In the beginning, all views are the same. When the first thread $T1$ performs the first write, it updates $view_{\text{cur}}$, but $view_{\text{rel}}$ remains the same. The message view of the newly added write equals to the pointwise maximum of $view_{\text{rel}}$ and the timestamp of the write itself, $[x@\mathbf{1}] = (\lambda\ell.\,\underline{\text{if}}\,\ell = x\,\underline{\text{then}}\,\mathbf{1}\,\underline{\text{else}}\,\mathbf{0})$ (the latter is called a *singleton* view).

$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle\}$
$T1.view_{\text{cur}} = [x@\,\mathbf{1}\,, y@\mathbf{0}]; \quad T1.view_{\text{acq}} = [x@\,\mathbf{1}\,, y@\mathbf{0}]; \quad T1.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$
$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$

After that the first thread $T1$ executes the release fence, which makes its $view_{\text{rel}}$ to be equal to $view_{\text{cur}}$:

$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle, \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle\}$
$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{0}]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{0}]; \quad T1.view_{\text{rel}} = [x@\,\mathbf{1}\,, y@\mathbf{0}];$
$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}]$

Then, the first thread $T1$ performs the second write, again attaching to it a view which is the pointwise maximum of $T1.view_{\text{rel}}$ and the timestamp of the write itself:

$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle,$
$\qquad \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\}$
$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\,\mathbf{1}\,]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\,\mathbf{1}\,]; \quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$
$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$

When the second thread $T2$ reads from the newly added write, it updates its $view_{\text{acq}}$ with the message view:

$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle,$
$\qquad \langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\}$
$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$
$T2.view_{\text{cur}} = [x@\mathbf{0}, y@\mathbf{0}]; \quad T2.view_{\text{acq}} = [x@\,\mathbf{1}\,, y@\,\mathbf{1}\,]; \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}];$

The execution of the acquire fence makes the second thread's $view_{\text{cur}}$ to be equal to its $view_{\text{acq}}$:

$$M = \{\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle, \langle y : 0@\mathbf{0}, [y@\mathbf{0}]\rangle,$$
$$\langle x : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{0}]\rangle, \langle y : 1@\mathbf{1}, [x@\mathbf{1}, y@\mathbf{1}]\rangle\}$$
$$T1.view_{\text{cur}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}]; \quad T1.view_{\text{rel}} = [x@\mathbf{1}, y@\mathbf{0}];$$
$$T2.view_{\text{cur}} = [x@\boxed{\mathbf{1}}, y@\boxed{\mathbf{1}}] \quad T2.view_{\text{acq}} = [x@\mathbf{1}, y@\mathbf{1}] \quad T2.view_{\text{rel}} = [x@\mathbf{0}, y@\mathbf{0}]$$

And now thread $T2$ is not able to read from $\langle x : 0@\mathbf{0}, [x@\mathbf{0}]\rangle$, as $T2.view_{\text{cur}}(x) = \mathbf{1} > \mathbf{0}$, which rules out the outcome $a = 1, b = 0$.

2.1 A More Complex Behavior

Both machines guarantee that a read instruction cannot be satisfied from a same thread's write which follows the read in program order. They do, however, allow to get $a = 1$ during an execution of the program presented in Section 1:

$$
\begin{array}{l|l|l}
a := [x]; \; //1 & b := [x]; & c := [y]; \\
[x] := 1; & [y] := b; & [x] := c;
\end{array}
\qquad\qquad \text{(ARM-weak)}
$$

The ARM Machine   The behavior may be reproduced by the ARM machine if the first and the second threads share a common buffer, which is not observable by the third thread. The machine issues the two requests of the first thread and the read request of the second thread, and propagates them to the common buffer. Then, the storage subsystem satisfies read $b := [x]$ from $[x] := 1$, and the second thread issues $[y] := 1$. The storage subsystem propagates it to the common buffer, reorders it with the first thread's requests, and propagates it to the lowest buffer and to the memory.
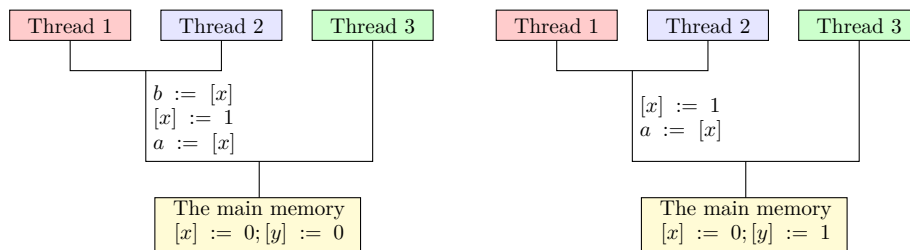
Next, the third thread issues $c := [y]$, and the storage propagates it to the memory, where it is satisfied with response $c = 1$. Now the storage sends response $c = 1$ to the third thread, it issues $[x] := 1$, and the storage subsystem propagates it to the lowest buffer.
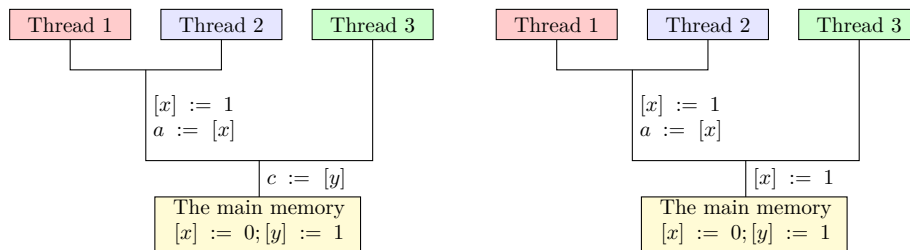
The storage subsystem propagates $a := [x]$ to the lowest buffer and satisfies it from $[x] := 1$.

The Promise Machine   Conforming to its name, the Promise machine uses *promises* to achieve the same behavior: a thread T may nondeterministically promise to write a value *val* to a location ℓ at some point in the future. When a thread T makes a promise, it adds $\langle \ell : val@\tau, \_\rangle$, where τ has not been used as a timestamp yet and is greater than $T.view_{\text{cur}}(\ell)$, to the memory, making the promise available to read from for other threads. The promise transition also adds the promise to a thread's set of promises, $T.promises$, but it does not update the thread's views. After each transition of the machine, the thread which makes a step has to *certify* that it is able to fulfill all promises it made in the current state of the memory running in isolation. The certification is used to outlaw self-satisfaction and causality cycles [8] in an execution.

To get $a = 1$ in the program ARM-weak, the first thread has to promise, e.g., $\langle x : 1@\mathbf{2}, [x@\mathbf{2}]\rangle$. The thread can certify the promise—to read from the initial write to x with timestamp $\mathbf{0}$ and then fulfill the promise by the second instruction. After the first thread promised a write to x, the second thread reads from the promise, and adds $\langle y : 1@\mathbf{1}, [y@\mathbf{1}]\rangle$ to the memory. The third thread reads from it, and adds $\langle x : 1@\mathbf{1}, [x@\mathbf{1}]\rangle$. Now the first thread can read from $\langle x : 1@\mathbf{1}, [x@\mathbf{1}]\rangle$ getting $a = 1$ and fulfill the promise $\langle x : 1@\mathbf{2}, [x@\mathbf{2}]\rangle$.

2.2   More Abstract Storage Subsystem: POP

Flur et al. [9] present two versions of the storage subsystem for the ARM machine: Flowing and POP (partial order propagation). We used the Flowing model to describe the previous examples because it is more intuitive and easier to understand. However, the Flowing model has a couple of features that make it hard to reason about the model. First, it is much easier to have a partial order on requests inside of a buffer than to keep track of reorderings inside it. Second, if we want to show that for every execution of a program in the ARM machine some invariant holds, we have to consider every possible topology of buffers.

The POP model solves the aforementioned obstacles. There are no linear buffers and fixed topologies. The state of the POP storage consists of three components: *Evt*—a set of requests observed by the storage, *Ord*—a partial order on requests from *Evt*, and *Prop*—a function mapping each thread identifier to a subset of *Evt* requests propagated to the thread. If two requests e and e' are ordered by *Ord*, $Ord(e, e')$, we write $e <_{Ord} e'$.
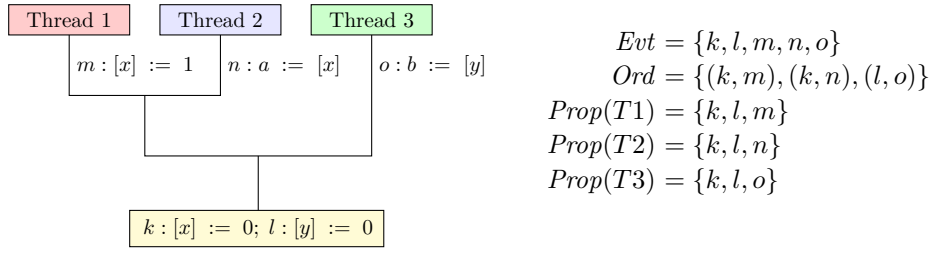
To understand how the POP model works and its connection to the Flowing model, consider an execution of the following program:

$$[x] \; := \; 1; \; \left\| \; \begin{array}{l} a \; := \; [x]; \; //1 \\ [y] \; := \; a; \end{array} \right. \left\| \; \begin{array}{l} b \; := \; [y]; \; //1 \\ c \; := \; [x + b * 0]; \; //0 \end{array} \right. \qquad \text{(WRC+data+addr)}$$

Here is a fake address dependency between reads in the third thread, so the thread does not know the target address of the second read until $b := [y]$ is satisfied. For this reason, the third thread cannot issue the second read request before the first one is satisfied. Nevertheless, the behavior $a = 1, b = 1, c = 0$ is allowed due to the storage subsystem.
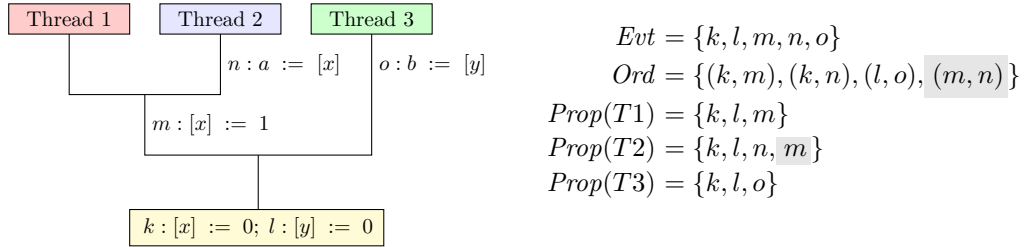
To reproduce the intended behavior in the Flowing model we have to choose the same topology as in the previous example, which has a buffer observable to the first and the second threads, but non-observable for the third one. Suppose that each thread issued a request corresponding to its first instruction. Then we get the following Flowing state (on the left) and the following POP state (on the right):[4]

---

[4]   For the sake of brevity we annotate the requests with labels and use the labels in the POP components.

$$Evt = \{k, l, m, n, o\}$$
$$Ord = \{(k, m), (k, n), (l, o)\}$$
$$Prop(T1) = \{k, l, m\}$$
$$Prop(T2) = \{k, l, n\}$$
$$Prop(T3) = \{k, l, o\}$$

When a request e is issued to the storage by a thread T, we add an Ord-edge (e', e) for each e', which is propagated to T and is not reorderable with e, $e' \not\rightarrow e$. That is why there are entries in Ord.
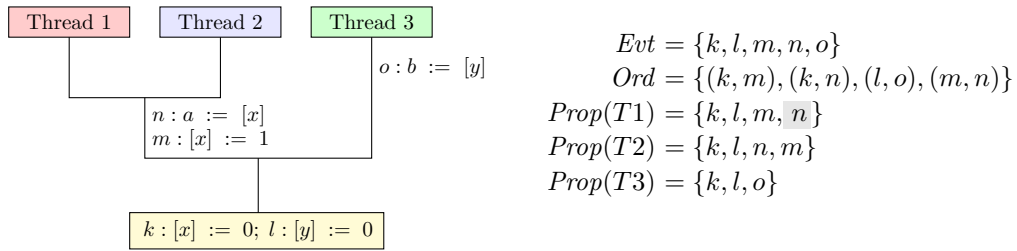
At this point the request $m : [x] := 1$ might be propagated to the $(T1, T2)$ common buffer. In terms of the POP model, this step corresponds to propagation of $m : [x] := 1$ to $T2$:
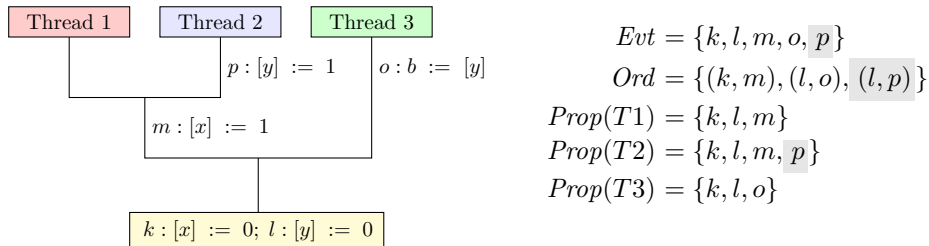
$$Evt = \{k, l, m, n, o\}$$
$$Ord = \{(k, m), (k, n), (l, o), (m, n)\}$$
$$Prop(T1) = \{k, l, m\}$$
$$Prop(T2) = \{k, l, n, m\}$$
$$Prop(T3) = \{k, l, o\}$$

The propagation step adds m to $Prop(T2)$ and the (m, n) edge to Ord.

In general, when a request e issued by a thread T is propagated to a thread T', we add (e, e') to Ord for every request e', which is propagated to T' but not to T, if e and e' are not reorderable (i.e., $e \not\rightarrow e'$) and there is no backward edge (e', e) in Ord. In the execution, the m and n requests are not reorderable because they operate on the same location x.
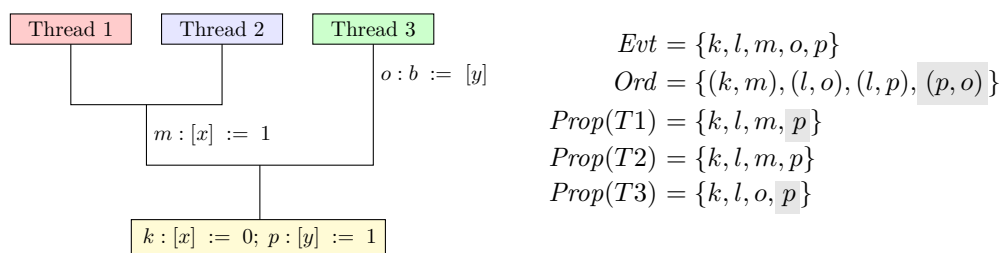
The storage subsystem may propagate $n : a := [x]$ to the common buffer (in the Flowing model) or, correspondingly, to thread $T1$ in the POP model:

$$Evt = \{k, l, m, n, o\}$$
$$Ord = \{(k, m), (k, n), (l, o), (m, n)\}$$
$$Prop(T1) = \{k, l, m, n\}$$
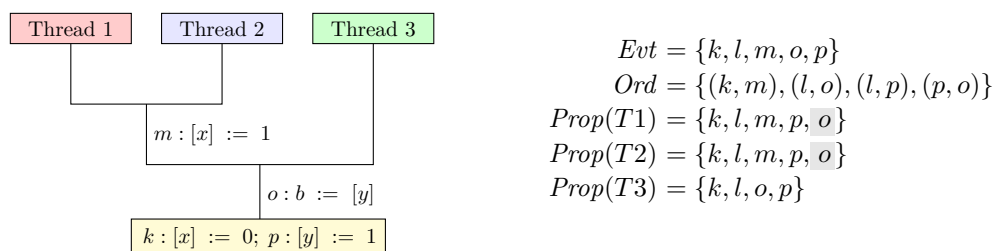$$Prop(T2) = \{k, l, n, m\}$$
$$Prop(T3) = \{k, l, o\}$$

Now $n : a := [x]$ can be satisfied from $m : [x] := 1$, as the former request follows the latter directly in the common buffer (the Flowing model), they are propagated to the same set of threads $(T1, T2)$ and there is no request in between them according to the Ord relation (the POP model). After the read is satisfied, the second thread $T2$ issues the write $p : [y] := 1$:

$$Evt = \{k, l, m, o, p\}$$
$$Ord = \{(k, m), (l, o), (l, p)\}$$
$$Prop(T1) = \{k, l, m\}$$
$$Prop(T2) = \{k, l, m, p\}$$
$$Prop(T3) = \{k, l, o\}$$

The storage propagates $p : [y] := 1$ to the common buffer and then to the lowest buffer and to the main memory in the Flowing model, and to $T1$ and $T3$ in the POP model:[5]

$$Evt = \{k, l, m, o, p\}$$
$$Ord = \{(k, m), (l, o), (l, p), (p, o)\}$$
$$Prop(T1) = \{k, l, m, p\}$$
$$Prop(T2) = \{k, l, m, p\}$$
$$Prop(T3) = \{k, l, o, p\}$$

Then, $o : b := [y]$ is propagated to the lowest buffer in the Flowing model, and to $T1$ and $T2$ in the POP model:

$$Evt = \{k, l, m, o, p\}$$
$$Ord = \{(k, m), (l, o), (l, p), (p, o)\}$$
$$Prop(T1) = \{k, l, m, p, o\}$$
$$Prop(T2) = \{k, l, m, p, o\}$$
$$Prop(T3) = \{k, l, o, p\}$$

After $o : b := [y]$ is satisfied from $p : [y] := 1$, the machine may issue $q : c := [x]$, propagate it to the main memory before $m : [x] := 1$ and satisfy it from $k : [x] := 0$.

At the end of the POP execution, all write and fence requests are propagated to all threads. As a result, every pair of write requests to the same location is ordered by Ord, which induces a total order on writes to one location. We use this observation in our proof.

As the POP model is a sound relaxation of the Flowing model and it is more abstract and easier to reason about, we use the POP model in our proof.

3 Main Challenges and High-Level Proof Structure

A compilation scheme from one machine AM to another machine AM' is correct, if for any program P and its compiled version P' each execution of P' on AM' corresponds to some execution of P on AM. The standard way to prove it is to exhibit a simulation between the machines. This may be done by introducing execution invariants and mapping transitions of AM' to transitions of AM.

There are three main problems one has to cope with to prove that the Promise machine simulates the ARM machine:

1. Although all writes to a specific location are totally ordered in the end of an ARM execution, they aren't ordered *during* the execution. In the Promise machine, however, timestamps induce a total order on writes, which is decided much earlier—at the point writes are issued (or promised).

2. In the ARM machine, while reading from a write request imposes restrictions on following reads, there is no explicit counterpart of message views of the Promise machine.

3. The ARM machine allows out-of-order execution of instructions, whereas the Promise machine, except for promises themselves, supports only in-order execution.

---

[5] As there is no fixed topology in the POP model, $p : [y] := 1$ can even be propagated to $T3$ before $T1$.

To address the first two challenges, we introduce an instrumented version of the ARM machine, the ARM+τ machine. In this machine, each write request is annotated with *(i)* a timestamp, *(ii)* a set of writes and fences which are guaranteed to be observed by a thread once it reads from the write request, and *(iii)* a view corresponding to the aforementioned set. The timestamps of writes to a specific location have to reflect a total order in which the writes are propagated to the main memory (in terms of the Flowing model) or to all threads (in terms of the POP model). However, at the moment when the thread subsystem issues a store request, the ARM machine cannot guarantee that the request will take any specific position in the total order. It is, therefore, impossible to assign a timestamp to the request at that moment. To solve this problem, the ARM+τ machine's steps have additional preconditions which guarantee acyclicity of the union of the partial order on requests in the storage (the relation *Ord*) and the per-location timestamp order. These restrictions mean that the ARM+τ machine may have potentially less behaviors than the ARM machine for a given program. So we have to prove that the ARM+τ machine simulates the ARM machine to be able to use it in the compilation correctness proof.

The third challenge makes it impossible to define a simple one-to-one or one-to-many correspondence between steps of the ARM and the Promise machines. To address this problem, we allow the Promise machine to "lag behind" the ARM machine. Consider the following program fragment:

$$
\begin{aligned}
[x] &:= 1; \\
&\mathsf{dmb\ LD}; \\
a &:= [y]; \\
[z] &:= 1;
\end{aligned}
$$

The ARM machine may first commit the fence $\mathsf{dmb\ LD}$ (*step 1*), then issue (propagate if needed and satisfy) the read $a := [y]$ (*step 2*), commit the write $[z] := 1$ (*step 3*), and only after that commit the write $[x] := 1$ (*step 4*). The Promise machine cannot perform the corresponding steps in the same order. According to the simulation we propose for the compilation correctness proof, the Promise machine does nothing when the ARM machine performs the steps 1 and 2, so it starts to lag behind the ARM machine at this point. Then it promises $[z] := 1$ at step 3. Finally, at step 4, it promises and fulfills the write $[x] := 1$ and does everything left, as it is no longer blocked by the instruction $[x] := 1$.

To represent the lagging of the Promise machine, we have two simulation relations in our proof, $\mathcal{I}$ and $\mathcal{I}_{\mathrm{pre}}$. The former relation forbids the Promise machine to lag behind too much: if states of the ARM and the Promise machines are connected by $\mathcal{I}$, then each thread of the Promise machine has executed all instructions from the maximal committed prefix of the corresponding ARM thread execution and is waiting for the next instruction to be committed by the ARM thread. The latter relation states that there is one thread which is able to (and has to) execute the next instruction, as it is committed by the ARM thread. We show that once states of the machines are related by $\mathcal{I}_{\mathrm{pre}}$, there is a finite number of steps, which the Promise machine has to make, to get to a state which satisfies $\mathcal{I}$.

In our proof, we consider only terminating executions of the ARM machine, because otherwise we would have to introduce "fairness" conditions on its speculative execution. For instance, there is an execution of the following program, which infinitely issues read requests to the storage without satisfying them:

$$
\begin{aligned}
a &:= [x]; \\
&\mathsf{if}\ a \neq 0\ \mathsf{goto}\ -1;
\end{aligned}
$$

$$
\begin{aligned}
cmds &\quad: List\ S \\
S &\quad ::= reg\ :=\ [expr] \\
&\quad\ |\ [expr_0]\ :=\ expr_1 \\
&\quad\ |\ \mathsf{dmb}\ ftype_{\mathrm{ARM}} \\
&\quad\ |\ \mathsf{if}\ expr\ \mathsf{goto}\ k \\
&\quad\ |\ reg\ :=\ expr\ |\ \mathsf{nop} \\
ftype_{\mathrm{ARM}} &\quad ::= \mathsf{SY}\ |\ \mathsf{LD} \\
expr &\quad ::= reg\ |\ \ell\ |\ uop\ expr \\
&\quad\ |\ bop\ expr_0\ expr_1 \\
&\quad\ : Expr \\
reg : Reg &\quad -\ a, b, c, ... \quad \text{(local variables)} \\
\ell : Loc &\quad -\ x, y, z, ... \quad \text{(locations)} \\
uop, bop &\quad -\ \text{arithmetic operations} \\
k &\quad \in \mathbb{Z}
\end{aligned}
$$

◼ **Figure 1** Syntax of ARM programs.

$$
\begin{aligned}
tapecell &\quad ::= \mathsf{R}\ st_{\mathrm{read}}\ |\ \mathsf{W}\ st_{\mathrm{write}} \\
&\quad\ |\ \mathsf{F}\ st_{\mathrm{fence}}\ ftype_{\mathrm{ARM}} \\
&\quad\ |\ \mathsf{If}\ st_{\mathrm{ifgoto}}\ k\ |\ \mathsf{Assign}\ |\ \mathsf{Nop} \\
&\quad\ : TapeCell \\[4pt]
sat\text{-}state &\quad ::= \mathsf{pln}\ |\ \mathsf{inflight}\ |\ \mathsf{com} \\
st_{\mathrm{read}} &\quad ::= \mathsf{none}\ |\ \mathsf{requested}\ \ell \\
&\quad\ |\ \mathsf{sat}\ sat\text{-}state\ \langle tid, path, \mathsf{wr}\ \ell : val\rangle \\
st_{\mathrm{write}} &\quad ::= \mathsf{none} \\
&\quad\ |\ \mathsf{pending}\ \ell\ val \\
&\quad\ |\ \mathsf{com}\ bool\ \ell\ val \\
st_{\mathrm{fence}} &\quad ::= \mathsf{none}\ |\ \mathsf{com} \\
st_{\mathrm{ifgoto}} &\quad ::= \mathsf{none}\ |\ \mathsf{taken}\ |\ \mathsf{ignored} \\
val : Val &\quad = Loc \cup \mathbb{Z}
\end{aligned}
$$

◼ **Figure 2** ARM instruction state.

As we said in Section 2, the considered compilation scheme is bijection, so we assume that programs for both machines are the same. Our compilation correctness theorem states that for every program and every terminating execution of the ARM machine there is a terminating execution of the Promise machine which ends with the *same memory*, i.e., the last writes to each location are the same for both machines.

▶ **Theorem 3.1.** *For all Prog and* **s***, if* $Prog \vdash \mathbf{s}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}]{}{}^* \mathbf{s}$ *and* $\mathsf{Final}^{\mathrm{ARM}}(\mathbf{s}, Prog)$, *then there exists* **p** *such that* $Prog \vdash \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}{}^* \mathbf{p}$ *where* $\mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, Prog)$ *and* $\mathsf{same\text{-}memory}(\mathbf{s}, \mathbf{p})$.

4   The ARM Machine

In this section, we formalize the semantics of ARM POP machine of Flur et al. [9]. The syntax of ARM machine programs is presented in Fig. 1. A program for the machine, $Prog : Tid \rightarrow List\ S$, consists of a list of instructions for each thread. Instructions are reads ($reg\ :=\ [expr]$), writes ($[expr_0]\ :=\ expr_1$), fences ($\mathsf{dmb}\ ftype_{\mathrm{ARM}}$), conditional relative jumps ($\mathsf{if}\ expr\ \mathsf{goto}\ k$), local variable assignments ($reg\ :=\ expr$), and no-operation instructions (nop).

The thread subsystem of the ARM machine allows out-of-order and speculative execution of instructions. Moreover, it executes instructions non-atomically, i.e., many instructions might be in the middle of their execution at the same moment. We represent a state of an instruction instance via *tapecell* (see Fig. 2). Its syntax reflects the instruction syntax.
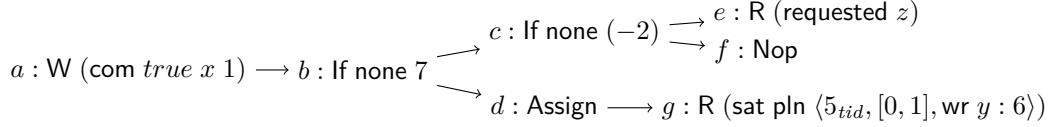
A read instance, st_{read}, might be in one of the three following states: (i) none, the read is fetched or restarted; (ii) $\mathsf{requested}\ \ell$, the read has a load request from the location ℓ in the storage subsystem; (iii) $\mathsf{sat}\ sat\text{-}state\ \langle tid, path, \mathsf{wr}\ \ell : val\rangle$, the read has been satisfied from a write instance ($tid, path$) with a value val. The *sat-state* field denotes if the read is satisfied by an in-flight, i.e., not yet committed to the storage, write ($\mathsf{inflight}$), the read is satisfied from the committed write (pln), or the read is satisfied from the committed write and is committed itself (com).

A write instance, st_{write}, might be (i) none, similarly to the read state; (ii) $\mathsf{pending}\ \ell\ val$, the address and the value of the write is determined and a read from the same thread may

read from it; (iii) com *bool* ℓ *val*, the write is committed and, if *bool = true*, issued to the storage subsystem (otherwise, it is observable only by same thread read instructions).

A fence instance, st_{fence}, might be either committed (com) or not (none). A conditional branch instance, st_{ifgoto}, signifies that the control flow either jumps to k positions ahead (taken), proceeds to the next instruction (ignored), or is still undecided (none). Assignments and nops instances are just fetched, but not executed.

Similarly to earlier work on the Power memory model [22], we may represent the instruction state of a specific thread as a labeled direct acyclic graph (DAG), e.g.:

$$a : \mathsf{W}\ (\mathsf{com}\ true\ x\ 1) \longrightarrow b : \mathsf{If\ none}\ 7 \overset{\nearrow}{\searrow} \begin{matrix} c : \mathsf{If\ none}\ (-2) \overset{\longrightarrow}{\longrightarrow} \begin{matrix} e : \mathsf{R}\ (\mathsf{requested}\ z) \\ f : \mathsf{Nop} \end{matrix} \\ d : \mathsf{Assign} \longrightarrow g : \mathsf{R}\ (\mathsf{sat\ pln}\ \langle 5_{tid}, [0,1], \mathsf{wr}\ y : 6\rangle) \end{matrix}$$

where vertices denote instruction instances, arrows represent the program order relation between the instances, and vertices with two outgoing arrows signify branch instruction instances, where the execution path has not yet been determined.

We identify an instruction instance by its thread identifier, *tid*, and a *path* : $Path \triangleq List\ \mathbb{N}$ from the root of the thread's instruction DAG. It is encoded as a list of instruction positions corresponding to the instruction instances on the *path*. For example, in the DAG above the instruction instance a has a path $[0]$, b—$[0,1]$, and f—$[0,1,8,9]$.

The instruction DAG of the thread is represented by a *tape* : $Tape \triangleq Path \rightharpoonup TapeCell$, a map from *path*s to *tapecell*s. As a *tape* represents an instruction DAG, it is *prefix-closed*: if a *tape* is defined for *path*, then it is defined for every (non-empty) prefix of *path*.

As multiple instructions may be in flight at any given moment, it is not possible to define one per-thread state of local variables for a given moment of an execution. Consider the following execution fragment:

i	$cmds[i]$	$path$	$tape(path)$
0	$a := [x];$	0	R none
1	$[y] := a;$	$0,1$	W none
2	$a := [z];$	$0,1,2$	R (sat pln $\langle 8_{tid}, [0,1,2,3], \mathsf{wr}\ z : 9\rangle$)
3	$[w] := a;$	$0,1,2,3$	W none

Here the read $a := [z]$ is satisfied with a value 9, but $a := [x]$ isn't even issued to the storage. It means that a is defined for the fourth instruction, but not for the second one.

To cope with these subtleties, we introduce two state functions $\mathsf{regf}, \mathsf{regf}_{\mathsf{com}} : (List\ S \times Tape \times Path) \rightarrow (Reg \rightharpoonup Val)$, where $\mathsf{regf}(cmds, tape, path)$ and $\mathsf{regf}_{\mathsf{com}}(cmds, tape, path)$ represent the state of the local variables just *before* the instruction instance indexed by *path*. Their only difference is in the way they process satisfied but not yet committed reads (where $path : i$ denotes the extension of *path* with the instruction index i):[6]

$$\forall i, j.\ cmds[i] = ``reg := [expr]"\wedge$$
$$\quad tape(path) = \mathsf{R}\ (\mathsf{sat}\ sat\text{-}state\ \langle \_, \_, \mathsf{wr}\ \_ : val\rangle) \wedge sat\text{-}state \neq \mathsf{com} \Rightarrow$$
$$\mathsf{regf}\quad(cmds, tape, path : i : j) = \mathsf{regf}\quad(cmds, tape, path : i)[reg \mapsto val] \wedge$$
$$\mathsf{regf}_{\mathsf{com}}(cmds, tape, path : i : j) = \mathsf{regf}_{\mathsf{com}}(cmds, tape, path : i)[reg \mapsto \bot].$$

---

[6] The full inductive definition of the regf and $\mathsf{regf}_{\mathsf{com}}$ functions is given in the extended version of the paper [20].

For the previous example, the functions have the following values:

path	regf($cmds$, $tape$, $path$)	regf$_{com}$($cmds$, $tape$, $path$)
0	$\perp$	$\perp$
$0, 1$	$\perp$	$\perp$
$0, 1, 2$	$\perp$	$\perp$
$0, 1, 2, 3$	$[a \mapsto val]$	$\perp$

The variable maps is naturally extended to expression evaluators of type $Expr \rightharpoonup Val$. For the sake of brevity, we write $[\![-]\!]^{path}$ and $[\![-]\!]^{path}_{com}$ (or $[\![-]\!]$ and $[\![-]\!]_{com}$) for the evaluators when values of the $cmds$, $tape$ (and $path$) parameters are clear from the context.

The state of the storage subsystem, $M_{POP} = \langle Evt, Ord, Prop \rangle$, contains three components: $Evt \subseteq ReqSet$—a set of memory requests in the storage; $Ord \subseteq Evt \times Evt$—a partial order on memory requests; and $Prop \subseteq Tid \times Evt$—the set of requests that have been propagated to each thread. A request, req, itself contains the thread, tid, and the instruction instance, $path$, that issued the request, as well as some information, $reqinfo$, about the request:

$$reqinfo ::= \text{rd } \ell \mid \text{wr } \ell : val \mid \text{dmb}.$$

Specifically, read requests record the location to be read, while write requests record the location and the value to be written.

The full state of the ARM machine State_{ARM} is a tuple $\langle M_{POP}, iordf, tapef \rangle$, where M_{POP} is the memory state, $iordf : Tid \rightarrow List\ ReqSet$ is a per-thread issuing order of read requests, and $tapef : Tid \rightarrow Tape$ records the tape of each thread. The ARM machine allows to issue read requests to the same location out-of-order, so it uses the issuing order to preserve coherence among the reads (discussed in the **Read satisfy** description).

The initial state of the ARM machine contains initial writes to all locations, $Evt^{init} \triangleq \{\langle 0_{tid}, [], \text{wr } \ell : 0 \rangle \mid \ell \in Loc\}$, the writes are not ordered and propagated to all threads:

$$\mathbf{s}^{init} \triangleq \langle M_{POP} = \langle Evt^{init}, \emptyset, Tid \times Evt^{init} \rangle, iordf = \lambda tid.\ [], tapef = \lambda tid.\ \perp \rangle.$$

Our version of the ARM machine has twelve possible transitions, which are shown in the extended version of the paper [20]. For simplicity, we present the transitions informally and do not separate them into storage and thread transitions.

Fetch instruction *tid path* adds a new instruction instance with a none state to the *tape* of the thread *tid*.

Propagate *e tid* adds e to a set of requests propagated to *tid*. It has to check that all requests e' which are ordered before e by Ord, i.e., $e' <_{Ord} e$, are propagated to *tid* as well. It also adds Ord-edges (e, e'') for every e'', which isn't reorderable with e and propagated to $e.tid$ but not to *tid*, to acknowledge that e is Ord-before e''.

Branch commit *tid path* processes an if − goto instruction instance and chooses which execution branch to drop, i.e., deletes instruction instances and storage requests belonging to the branch.

Fence commit LD *tid path* checks if previous reads are committed.

Fence commit SY *tid path* checks if previous instruction instances in general are committed, and issues a fence request to the storage.

Write pending *tid path ℓ val* sets the write instruction instance to pending ℓ val, where ℓ and *val* are an address and a value calculated by the corresponding evaluator.

Write commit *tid path ℓ val* sets the write instruction instance to com __ ℓ val. It issues a write request to the storage in case there is no following committed writes to the same

location in thread's *tape*. It restarts some satisfied load instances, which read from the same location, and their dataflow dependents to preserve coherence. Previous branch operators and fences have to be committed. All previous instructions must have fully determined addresses, i.e., each address in a previous instruction instance has to be determined by the corresponding *com*-evaluator.

Read issue *tid path ℓ* sends a read request to the storage, and adds it to the list of issued read requests (the *iordf*(*tid*) component). It requires that previous fences are committed.

Read satisfy *tid path tid′ path′ ℓ val* and **Read satisfy (fail)** *tid path tid′ path′ ℓ val* get the read request $\langle tid, path, \mathsf{rd}\ \ell \rangle$ satisfied from the write request $\langle tid', path', \mathsf{wr}\ \ell : val \rangle$, if there are no requests between them in the storage. The transitions delete the read request from the storage. If there are no previous read instances, which issued a read request to the same location after the (*tid, path*) instance (according to *iordf*(*tid*)) and have been satisfied from a different write, the former transition might be applied. It assigns the instruction instance to $\mathsf{sat\ pln}\ \langle tid', path', \mathsf{wr}\ \ell : val \rangle$ and restarts some *path*-following reads from the same location (and their dataflow dependents) to preserve coherence. Otherwise, the latter transition might be applied, which restarts the (*tid, path*) instruction instance.

Read satisfy from in-flight write *tid path path′ ℓ val* assigns the corresponding instruction instance to $\mathsf{sat\ inflight}\ \langle tid, path', \mathsf{wr}\ \ell : val \rangle$, if there is a previous pending write (*tid, path′*) and there are no writes to the same location in between the write and the read, as well as there are no same location reads satisfied from another write. It restarts some *path*-following reads as in the case of the transition **Read satisfy**.

Read commit *tid path* checks that previous branches and fences are committed, all previous instruction instances have a fully determined addresses, and assigns the instruction instance to $\mathsf{sat\ com}\ \_$.

5 The Promise Machine

As mentioned in Section 2, the compilation scheme from Promise to ARM is a bijection; so we may skip the definition of the Promise program syntax and use the ARM syntax.

The state of the Promise machine $\mathsf{State}_{\mathrm{Promise}}$ is a tuple $\langle M_{\mathrm{Promise}}, tsf \rangle$. The memory, $M_{\mathrm{Promise}} \subset Msg$, is a set of write messages, $\langle \ell : val@\tau, view \rangle : Msg$, which records the write's location, $\ell : Loc$, value, $val : Val$, timestamp, $\tau : Time = \mathbb{Q}$, and message view, $view : View = Loc \to Time$. The memory includes writes which are promised but not yet fulfilled. In turn, $tsf : Tid \to TS$ is a per-thread state. A thread state, $ts : TS$, is a tuple $\langle path, st, V, promises \rangle$, where $path$ is a pointer to the next instruction to be executed; $st : Reg \rightharpoonup Val$ is a variable state function; $V = \langle view_{\mathrm{cur}}, view_{\mathrm{acq}}, view_{\mathrm{rel}} \rangle : View \times View \times View$ is a current, an acquire, and a release views of the thread; and $promises \subset Msg$ is a set of promises which the thread made but has not fulfilled yet.

The initial memory of the Promise machine contains initial writes to all locations, $M_{\mathrm{Promise}}^{\mathrm{init}} \triangleq \{\langle \ell : 0@\mathbf{0}, view^{\mathrm{init}} \rangle \mid \ell \in Loc\}$, where $view^{\mathrm{init}} \triangleq \lambda\ell.\ \mathbf{0}$. The initial thread state's $path$ points to the first instruction, variables are not defined, and the set of promises is empty:

$$\mathbf{p}^{\mathrm{init}} \triangleq \langle M_{\mathrm{Promise}}^{\mathrm{init}}, \lambda tid.\ \langle path = [0], st = \bot, V = \langle view^{\mathrm{init}}, view^{\mathrm{init}}, view^{\mathrm{init}} \rangle, promises = \emptyset \rangle \rangle.$$

The main transition of the Promise machine is global:

$$
\frac{
\begin{array}{c}
Prog(tid) \vdash \langle M_{\mathrm{Promise}}, ts \rangle \xrightarrow[\mathrm{Promise}\ tid]{label} \langle M'_{\mathrm{Promise}}, ts' \rangle \\
Prog(tid) \vdash \langle M'_{\mathrm{Promise}}, ts' \rangle \xrightarrow[\mathrm{Promise}\ tid]{}{}^{*} \langle M''_{\mathrm{Promise}}, ts'' \rangle
\end{array}
\qquad ts''.promises = \emptyset
}{
Prog \vdash \langle M_{\mathrm{Promise}}, tsf[tid \mapsto ts] \rangle \xrightarrow[\mathrm{Promise}]{label\ tid} \langle M'_{\mathrm{Promise}}, tsf[tid \mapsto ts'] \rangle
}
$$

Other transitions ($\xrightarrow[\mathrm{Promise}\ tid]{}$) are defined for a specific thread. The main transition requires a thread tid, which makes a transition, to certify that it is able to fulfill its promises, i.e., there is an isolated execution of the thread with the current memory, which fulfills all thread's promises.[7]

The exact definitions of thread transitions might be found in the extended version of the paper [20]. Here we present the transitions informally. All of them, except for **Promise write**, execute the instruction pointed by the thread's *path* component, and update *path* to point to the next instruction.

Acquire fence commit makes the current view, $view_{\mathrm{cur}}$, of the thread to be equal to its acquire view, $view_{\mathrm{acq}}$, which accumulates message views of writes read by the thread up to the current point.

Release fence commit updates the release view, $view_{\mathrm{rel}}$, of the thread to match its current view. Consequently, the message view of writes issued after executing the fence will incorporate information about writes observed by the thread before the fence. In the original version of the Promise machine [12] the **Release fence commit** transition has a precondition that all unfulfilled promises of the thread must have empty message views. In our version, the machine is even more restrictive: the thread cannot have any unfulfilled promises. This restriction is easier to work with, and it is not too restrictive for the compilation proof—the release fence is compiled to the full fence in the ARM machine, which forbids to commit following writes before the fence itself is committed.

Read from memory ℓ chooses a message, $\langle \ell : val@\tau, view \rangle$, from memory with a timestamp, τ, greater than or equal to the current view value, $view_{\mathrm{cur}}(\ell)$. The transition updates thread's $view_{\mathrm{cur}}$ by $[\ell@\tau]$, and $view_{\mathrm{acq}}$ by $view$. It follows that such a message cannot be in the thread's set of unfulfilled *promises* as it would make it impossible for the thread to fulfill the corresponding promise. Also, the transition updates the thread's local variable map, st.

Promise write $\langle \ell : val@\tau, view \rangle$ adds the message to the memory and to the thread's set of promises. The target location, ℓ, and the value, val, can be chosen arbitrarily. The timestamp, τ, has to be unique among writes to the location. The message view equals to a composition of the release view, $view_{\mathrm{rel}}$, and a singleton view $[\ell@\tau]$.[8] The transition does not update the thread's views. As we see, this transition is very non-deterministic. However, it is restricted by certification.

Fulfill promise $\langle \ell : val@\tau, view \rangle$ removes the message from the thread's *promises*, if (*i*) the current instruction is a write, (*ii*) its target location and value are ℓ and val, (*iii*) τ is less than $view_{\mathrm{cur}}(\ell)$, and (*iv*) $view$ equals to $view_{\mathrm{rel}}$ updated by $[\ell@\tau]$. The transition updates $view_{\mathrm{cur}}$ and $view_{\mathrm{acq}}$ by $[\ell@\tau]$.

---

[7] In the original model, certification has to be made for all possible "future" memories. In the absence of Read-Modify-Write operations, however, we can simplify that condition and perform certifications starting only from the current memory.

[8] This is more restrictive than in the original presentation [12], which allows to promise a write with an arbitrary message view.

The other rules (**Branch commit**, **Local variable assignment**, and **Execution of nop**) have standard semantics.

6 Basic Properties of the ARM Storage

In this section we prove some properties of the ARM storage subsystem, which we use to introduce timestamps to the ARM machine in the following section. In all lemmas we assume some program *Prog* implicitly.

▶ **Lemma 6.1.** $\forall \mathbf{s}. \ \mathbf{s}^{\text{init}} \xrightarrow[\text{ARM}]{}^* \mathbf{s} \Rightarrow \mathbf{s}.Ord = (\mathbf{s}.Ord \setminus \hookrightarrow)^+ \wedge \mathbf{s}.Ord \text{ is acyclic.}$

Proof. The statement holds for the initial state, $\mathbf{s}^{\text{init}}$. Consider possible mutations of the storage. There are three types of storage operations. We assume that operations make a transition $\langle Evt, Ord, Prop \rangle \rightarrow \langle Evt', Ord', Prop' \rangle$. Let's check them:

Delete a read request e :

$$Evt' = Evt \backslash \{e\}, Prop' = Prop \backslash \{(tid, e) \mid tid\}, Ord' = (Ord \backslash (\{e\} \times Evt) \backslash (Evt \times \{e\}) \backslash \hookrightarrow)^+$$

Accept a request e **from** tid :

$$Evt' = Evt \cup \{e\}, Prop' = Prop \cup \{(tid, e)\}, Ord' = (Ord \cup \{(e', e) \mid Prop(tid, e'), e' \not\hookrightarrow e\})^+$$

Propagate a request e **to** tid :

$$Evt' = Evt, Prop' = Prop \cup \{(tid, e)\},$$
$$Ord' = (Ord \cup \{(e, e') \mid Prop(tid, e'), \neg Prop(e.tid, e'), e \not\hookrightarrow e', \neg(e' <_{Ord} e)\})^+$$

Obviously, $Ord' = (Ord' \backslash \hookrightarrow)^+$. As $Ord' \subseteq Ord$ for the delete transition, the accept transition adds edges only to a new request, and the propagate transitions checks if there is an edge (e, e') in transitively closed Ord before adding (e', e), Ord' is acyclic. ◀

The next two lemmas are proved in the similar way.

▶ **Lemma 6.2.** $\forall \mathbf{s}, e, e', tid. \ \mathbf{s}^{\text{init}} \xrightarrow[\text{ARM}]{}^* \mathbf{s} \wedge e \not\hookrightarrow e' \wedge \mathbf{s}.Prop(tid, e) \wedge \mathbf{s}.Prop(tid, e') \Rightarrow e = e' \vee \mathbf{s}.Ord(e, e') \vee \mathbf{s}.Ord(e', e).$

▶ **Lemma 6.3.** $\forall \mathbf{s}, \mathbf{s}'. \ \mathbf{s} \xrightarrow[\text{ARM}]{}^* \mathbf{s}' \Rightarrow \mathbf{s}.Evt \setminus \{e \mid e \text{ is a read request}\} \subseteq \mathbf{s}'.Evt.$

▶ **Lemma 6.4.** $\forall \mathbf{s}, \mathbf{s}'. \ \mathbf{s} \xrightarrow[\text{ARM}]{}^* \mathbf{s}' \Rightarrow \mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt) \subseteq \mathbf{s}'.Ord.$

Proof. The following weaker version of the lemma holds as there is no storage transition which deletes an *Ord*-edge between non-reorderable requests:

$$\forall \mathbf{s}, \mathbf{s}'. \ \mathbf{s} \xrightarrow[\text{ARM}]{}^* \mathbf{s}' \Rightarrow (\mathbf{s}.Ord \cap (\mathbf{s}'.Evt \times \mathbf{s}'.Evt)) \setminus \hookrightarrow \subseteq \mathbf{s}'.Ord$$

Now let's prove the original statement. Fix e, e' such that $\mathbf{s}.Ord(e, e')$. If $e \not\hookrightarrow e'$, then the statement holds as we have just shown. Otherwise, e and e' are read or write requests to different locations. As $\mathbf{s}.Ord(e, e')$ holds, by Lemma 6.1, there is a finite path in $\mathbf{s}.Ord$ from e to e' such that each edge along the path connects non-reorderable requests.

Suppose that there is a fence request e'' in the path. Then, by transitivity of $\mathbf{s}.Ord$, $\{(e, e''), (e'', e')\} \subseteq \mathbf{s}.Ord$. By the weaker version of the lemma and transitivity of $\mathbf{s}'.Ord$, $\{(e, e''), (e'', e'), (e, e')\} \subseteq \mathbf{s}'.Ord$.

Consider that there is no fence request in the path. Then, by definition of $\hookrightarrow$, the path comprises only requests to the same location. It contradicts that $e \not\hookrightarrow e'$. ◀

7 Introduction of Timestamps to the ARM Machine

In this section, we show how to assign timestamps (τ) represented by rational numbers to all write requests in a terminating execution of the ARM machine. Let us fix some program *Prog*, and consider a terminating execution:

$$Prog \vdash \mathbf{s}^0 \xrightarrow[\text{ARM}]{} \mathbf{s}^1 \xrightarrow[\text{ARM}]{} \ldots \xrightarrow[\text{ARM}]{} \mathbf{s}^n$$

where $\mathbf{s}^0 = \mathbf{s}^{\text{init}}$, an initial state of the ARM machine, and $\mathbf{s}^n = \mathbf{s}$ is a final state, i.e., there are no read requests in the storage, all requests are propagated to all threads, all instruction instances are committed, and it is impossible to fetch any new instruction instance.

For a location ℓ and a set of memory requests Evt, we define Evt_ℓ to be the set of all write requests to the location ℓ in Evt. Formally,

$$Evt_\ell \triangleq \{\langle tid, path, \mathsf{wr}\ \ell : val\rangle \in Evt \mid tid, path, val\}.$$

There is no transition which deletes write requests from the storage, so $\mathbf{s}.Evt_\ell$ is the set of all writes to a location ℓ which have been issued to the storage subsystem during the execution.

Fix a location ℓ. We know that each request e from $\mathbf{s}.Evt_\ell$ is propagated to all threads, as $\mathbf{s}$ is a final state. We also know that two different writes to the same location are not reorderable. As a consequence of it and Lemma 6.2, we have that

$$mo_\ell \triangleq \mathbf{s}.Ord{\restriction}_{\mathbf{s}.Evt_\ell} \qquad \text{where } R{\restriction}_S \triangleq R \cap (S \times S)$$

is a total order on the writes to the location ℓ. We define $mo \triangleq \bigcup_\ell mo_\ell$ to be the union of mo_ℓ for all locations mentioned in the execution. Using request indexes in the corresponding mo_ℓ sets, we define a timestamp mapping function:

$$\mathsf{map}_\tau(e) \triangleq \begin{cases} \mathsf{index}(mo_\ell, e) & \text{if } e = \langle tid, path, \mathsf{wr}\ \ell : val\rangle \in \mathbf{s}.Evt; \\ \bot & \text{otherwise.} \end{cases}$$

Finally, we show that for every state $\mathbf{s}^i$ of the execution $mo{\restriction}_{\mathbf{s}^i.Evt} \cup \mathbf{s}^i.Ord$ is acyclic.

▶ **Theorem 7.1.** $\forall i \leq n, mo{\restriction}_{\mathbf{s}^i.Evt} \cup \mathbf{s}^i.Ord$ *is acyclic.*

Proof. The statement obviously holds for $\mathbf{s}^0$. Suppose that there exists j such that for all $i < j$ the relation $mo{\restriction}_{\mathbf{s}^i.Evt} \cup \mathbf{s}^i.Ord$ is acyclic, but $mo{\restriction}_{\mathbf{s}^j.Evt} \cup \mathbf{s}^j.Ord$ has a cycle. We know that $mo{\restriction}_{\mathbf{s}^n.Evt} \cup \mathbf{s}^n.Ord = \mathbf{s}^n.Ord$ has no cycles. So if there is a cycle in $mo{\restriction}_{\mathbf{s}^j.Evt} \cup \mathbf{s}^j.Ord$, it has to be "destructed" on the subexecution $Prog \vdash \mathbf{s}^j \xrightarrow[\text{ARM}]{}^* \mathbf{s}^n$.

From this point on, we'll distinguish Ord- and mo-edges. We call an edge an Ord-edge, if it is in $\mathbf{s}^j.Ord$, and we call it an mo-edge, if it is in $mo{\restriction}_{\mathbf{s}^j.Evt} \setminus \mathbf{s}^j.Ord$.

Consider a shortest cycle in $mo{\restriction}_{\mathbf{s}^j.Evt} \cup \mathbf{s}^j.Ord$. It has to contain an mo-edge, because $\mathbf{s}^j.Ord$ is acyclic. The mo-edge (e, e') connects two writes to some location ℓ and $\mathsf{map}_\tau(e) < \mathsf{map}_\tau(e')$. This edge is a part of the cycle, so there is a path from e' to e by Ord- and mo-edges. We can break the path into mo-subpaths and Ord-subpaths. Let's check that each aforementioned Ord-subpath contains only one edge.

We pick an Ord-subpath, $\{e_i''\}_{i \in [0..k]}$. e_0'' and e_k'' are write requests, as they are connected to other subpaths via mo-edges. By transitivity of $\mathbf{s}^j.Ord$ (Lemma 6.1), $\mathbf{s}^j.Ord(e_0'', e_k'')$ holds, so the subpath can be reduced to these two requests.

Thus, the shortest path from e' to e in $mo{\restriction}_{\mathbf{s}^j.Evt} \cup \mathbf{s}^j.Ord$ contains only write requests. $\mathbf{s}^n.Ord$ contains all mo-edges from the path by definition of mo, and it contains each Ord-edge from the path by Lemma 6.3 and Lemma 6.4. It contradicts acyclicity of $\mathbf{s}^n.Ord$. ◀

$$Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{label} \mathbf{s}'$$
$$label \notin \{\textbf{Propagate \_\_},\textbf{Write commit \_\_\_\_}\}$$
$$\overline{Prog \vdash \langle \mathbf{s}, H \rangle \xrightarrow[\text{ARM}+\tau]{label} \langle \mathbf{s}', H \rangle}$$

$$Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{\textbf{Propagate } e \; tid} \mathbf{s}'$$
$$\mathbf{s}'.Ord \cup \text{tedges}(\mathbf{s}'.Evt, H_\tau) \text{ is acyclic}$$
$$\overline{Prog \vdash \langle \mathbf{s}, H \rangle \xrightarrow[\text{ARM}+\tau]{\textbf{Propagate } e \; tid} \langle \mathbf{s}', H \rangle}$$

$$Prog \vdash \mathbf{s} \xrightarrow[\text{ARM}]{\textbf{Write commit } tid \; path \; \ell \; val} \langle M'_{\text{POP}}, iordf', tapef' \rangle$$
$$tapef'(tid, path) = \mathsf{W} \; (\text{com } im \; \ell \; val) \qquad tape \triangleq \mathbf{s}.tapef(tid)$$
$$\text{time-range}(im, \ell, \tau, tid, path, tape, H) \qquad \text{coherent-thread}(\ell, \tau, path, tape, H)$$
$$\text{uniq-time-loc}(\ell, \tau, \mathbf{s}.tapef, H) \qquad path^{\mathsf{SY}} \triangleq \mathsf{lastSY}(tape, path)$$
$$S \triangleq \underline{\text{if }} path^{\mathsf{SY}} \neq [] \underline{\text{ then }} \{\langle tid, path^{\mathsf{SY}}, \mathsf{dmb} \rangle\} \cup \text{prev-Ord-req}(tid, path^{\mathsf{SY}}, tape, H) \underline{\text{ else }} \emptyset$$
$$S' \triangleq \underline{\text{if }} im \underline{\text{ then }} S \cup \{\langle tid, path, \mathsf{wr} \; \ell : val \rangle\} \underline{\text{ else }} S$$
$$view \triangleq [\ell @ \tau] \sqcup \text{viewf}(tid, path^{\mathsf{SY}}, path^{\mathsf{SY}}, tape, H)$$
$$H' \triangleq H[(tid, path) \mapsto (\tau, S', view)] \qquad M'_{\text{POP}}.Ord \cup \text{tedges}(M'_{\text{POP}}.Evt, H'_\tau) \text{ is acyclic}$$
$$\overline{Prog \vdash \langle \mathbf{s}, H \rangle \xrightarrow[\text{ARM}+\tau]{\textbf{Write commit } tid \; path \; \ell \; val \; \tau} \langle \langle M'_{\text{POP}}, iordf', tapef' \rangle, H' \rangle}$$

Figure 3 Transitions of the ARM+τ machine.

8 The ARM+τ Machine

In the previous section, we showed that one may assign timestamps for every write or fence request in a terminating execution of the ARM machine. Here, we introduce an instrumented version of the ARM machine, the ARM+τ machine, which assigns timestamps to the requests when it issues them to the storage.

8.1 Definition of the ARM+τ Machine

The ARM+τ machine state has one additional component $H : Tid \times Path \rightharpoonup Time \times 2^{ReqSet} \times View$. The H component is defined for committed write instruction instances. For each committed write, it assigns (i) a timestamp, (ii) a set of write and fence requests in the storage, which are guaranteed to be Ord-before the committed write request, if there is one, and (iii) a Promise-style message view representation of the write and fence request set—it maps a location to the greatest timestamp among write requests to the location in the set. For the sake of brevity, we define projections for H—$H_\tau : Tid \times Path \rightharpoonup Time$, $H_\leq : Tid \times Path \rightharpoonup 2^{ReqSet}$, and $H_{\mathsf{view}} : Tid \times Path \rightharpoonup View$. The H map component of the ARM+τ initial state, $\mathbf{a}^{\text{init}} \triangleq \langle \mathbf{s}^{\text{init}}, H^{\text{init}} \rangle$, assigns zero timestamps to all initial writes: $H^{\text{init}} \triangleq [(0_{tid}, []) \mapsto \langle \mathbf{0}, \lambda \ell. \; \mathbf{0}, \emptyset \rangle]$.

Transitions of the ARM+τ machine match the transitions of the plain ARM machine (see Fig. 3). There is a generic rule, which lifts all of the ARM transitions, except for **Propagate** and **Write commit**, to ARM+τ ones, leaving the timestamp component unchanged. The **Write commit** transition of ARM+τ mutates the timestamp component; and both **Propagate** and **Write commit** transitions have a common additional precondition: the union of the Ord relation and the order induced by timestamps, tedges, has to remain acyclic.

▶ **Definition.** $\text{tedges}(Evt, H_\tau) =$
$\{(e, e') \in Evt \times Evt \mid e, e' \text{ are writes}, e.loc = e'.loc, H_\tau(e.tid, e.path) < H_\tau(e'.tid, e'.path)\}$

The other rules cannot introduce cycles in the union, so we do not have to insert the additional precondition to them.

Let's take a closer look to the **Write commit** transition. It chooses a timestamp τ, which has to be unique among writes to the same location (the predicate uniq-time-loc). Also, the timestamp has to be consistent with timestamps of thread's committed writes to the location (the predicate coherent-thread): it has to be bigger than timestamps of the preceding writes and smaller than timestamps of the following writes. The **Write commit** transition of the original machine does not issue a write request to the storage in case there is a following committed write to the location ($im = false$). Nevertheless, the ARM+τ machine assigns a timestamp to it. To distinguish write instances that have write requests in the storage from those that do not, timestamps of instances with requests in the storage are integers, while timestamps of instances without requests are in a range $(\tau' - 1, \tau')$, where τ' is a timestamp of the closest following write to the same location, which has a write request in the storage (the predicate time-range).

If the write request is issued ($im = true$) and there is a preceding SY fence ($path^{\text{SY}} \neq []$), then requests guaranteed to be *Ord*-before the issued write request (its $H_<$ entry) are the last preceding fence request, $\langle tid, path^{\text{SY}}, \text{dmb}\rangle$, and prev-Ord-req($tid, path^{\text{SY}}, tape, H$)—the write requests issued by the thread before $path^{\text{SY}}$, and elements of $H_<(e.tid, e.path)$, for every write request e which is read by the thread before $path^{\text{SY}}$.

> prev-Ord-req($tid, path^{\text{SY}}, tape, H$) $\triangleq$
> $\quad \{\langle tid : path'@\ell', val'\rangle \mid path' < path^{\text{SY}}, tape(path') = \text{W (com}\_\ell' \, val')\} \cup$
> $\quad \bigcup\{H_<(e.tid, e.path) \mid path'' < path^{\text{SY}}, tape(path'') = \text{R (sat com } e)\}.$

Why are these requests *Ord*-before the added write request?

First, all these requests, except for the fence request itself, are *Ord*-before the fence request $\langle tid, path^{\text{SY}}, \text{dmb}\rangle$, because when the storage accepts a fence request e, it adds (e', e) edges to the *Ord* relation for all requests e' propagated to the thread, since no requests are reorderable with a fence request. Each write e', which was issued by the thread before the fence, was propagated to that thread, so the corresponding edge to $\langle tid, path^{\text{SY}}, \text{dmb}\rangle$ is added to *Ord*. Each write e'', which was read by the thread before the fence, was propagated to that thread as well, so edges from e'' itself and elements of its $H_<$-entry to $\langle tid, path^{\text{SY}}, \text{dmb}\rangle$ are added to *Ord*. Second, when the storage subsystem accepts the write request, it adds an edge from the fence request to it, as the latter is issued by the same thread (i.e., propagated to the thread). The others are *Ord*-before the write request by transitivity of *Ord*.

The H_{view} entry is equal to a pointwise maximum (the $\sqcup$ operation) of a write timestamp map $[\ell@\tau]$ and viewf($tid, path^{\text{SY}}, path^{\text{SY}}, tape, H$), where

> viewf($tid, path^{\text{write}}, path^{\text{read}}, tape, H$) $\triangleq$
> $\quad \bigsqcup \text{com-writes-time}(tid, path^{\text{write}}, tape, H) \sqcup \bigsqcup \text{sat-reads-view}(path^{\text{read}}, tape, H)$

which captures a composition of views corresponding to the elements of the $H_<$ entry:

> com-writes-time($tid, path, tape, H$) $\triangleq$
> $\quad \{[\ell@\tau] \mid path' < path, tape(path') = \text{W (com}\_\ell\_), \tau = H_\tau(tid, path')\} \cup$
> $\quad \{[\ell@\tau] \mid tid', path', path'' < path, \tau = H_\tau(tid, path') \neq \bot,$
> $\qquad tape(path'') = \text{R (sat } sat\text{-}state \, \langle tid', path', \text{wr } \ell : \_\rangle), sat\text{-}state \neq \text{inflight}\}.$

> sat-reads-view($path, tape, H$) $\triangleq$
> $\quad \{H_{\text{view}}(tid', path') \neq \bot \mid \exists \ell, tid', path', path'' < path,$
> $\qquad tape(path'') = \text{R (sat } sat\text{-}state \, \langle tid', path', \text{wr } \ell : \_\rangle), sat\text{-}state \neq \text{inflight}\}.$

8.2 Simulation of the ARM Machine

As we have just seen, the transitions of the ARM+τ machine are more restrictive than the ARM transitions, which may potentially lead to fewer possible behaviors of the instrumented machine. So, if we want to use the instrumented machine in the compilation proof, we have to show that it is possible to simulate the original machine.

▶ **Theorem 8.1.** $\forall Prog, \{\mathbf{s}^i\}_{i \in [0..n]}. \mathbf{s}^0 = \mathbf{s}^{\text{init}} \wedge \mathsf{Final}^{\text{ARM}}(\mathbf{s}^n, Prog) \wedge$
$Prog \vdash \mathbf{s}^0 \xrightarrow[\text{ARM}]{} \ldots \xrightarrow[\text{ARM}]{} \mathbf{s}^n \Rightarrow \exists \{H^i\}_{i \in [0..n]}. H^0 = \mathbf{a}^{\text{init}}.H \wedge$
$\quad Prog \vdash \langle \mathbf{s}^0, H^0 \rangle \xrightarrow[\text{ARM}+\tau]{} \ldots \xrightarrow[\text{ARM}+\tau]{} \langle \mathbf{s}^n, H^n \rangle.$

Proof. In Section 7, we constructed the relation mo and the function $\mathsf{map}_\tau : req \rightharpoonup \tau$ from the final state of an execution. Here, we do the same for $\mathbf{s}^n$, with a minor change: we suppose that the domain of map_τ is instruction instance identifiers $tid \times path$ instead of req. It is a stylistic change, as each req in the storage is uniquely identified (except for initial writes) by the instruction instance that issued it.

We construct $\{H^i\}_{i \in [0..n]}$ inductively. The initial map H^0 is equal to $\mathbf{a}^{\text{init}}.H$. We introduce an invariant for the ARM+τ execution we are constructing:

$\mathsf{inv}(\mathbf{s}, H) \triangleq \forall tid, path.$
$\quad (\mathbf{s}.tapef(tid, path) = \mathsf{W} \ (\mathsf{com} \ true \ \_ \ \_) \Rightarrow H_\tau(tid, path) = \mathsf{map}_\tau(tid, path)) \vee$
$\quad (\mathbf{s}.tapef(tid, path) \neq \mathsf{W} \ (\mathsf{com} \ \_ \ \_ \ \_) \Rightarrow H_\tau(tid, path) = \bot).$

The invariant says that the timestamps introduced during the instrumented execution are given by the map_τ function. We will prove that the invariant is maintained while constructing $\{H^i\}_{i \in [0..n]}$. Suppose that we made the first i transitions and the invariant holds for the corresponding states. Let's perform a case analysis of the $Prog \vdash \mathbf{s}^i \xrightarrow[\text{ARM}]{} \mathbf{s}^{i+1}$ step.

Propagate: We choose H^{i+1} to be equal to H^i. Then, $\mathsf{inv}(\mathbf{s}^{i+1}, H^{i+1})$ holds as $\mathbf{s}^{i+1}.tapef = \mathbf{s}^i.tapef$. In Section 7, we proved that for all $j \in [0..n]$, $mo\restriction_{\mathbf{s}^j.Evt} \cup \mathbf{s}^j.Ord$ is acyclic. $\mathsf{inv}(\mathbf{s}^{i+1}, H^{i+1})$ guarantees that $mo\restriction_{\mathbf{s}^{i+1}.Evt}$ is equal to $\mathsf{tedges}(\mathbf{s}^{i+1}.Evt, H_\tau^{i+1})$. Then $\mathbf{s}^{i+1}.Ord \cup \mathsf{tedges}(\mathbf{s}^{i+1}.Evt, H_\tau^{i+1})$ is acyclic. The additional precondition of the **Propagate** transition holds, and the ARM+τ machine can make the same step.

Write commit *tid path***:** There are two subcases to consider.

If the write request is issued to the storage, then we choose τ, a parameter of the ARM+τ transition, to be equal to $\mathsf{map}_\tau(tid, path)$. We choose H^{i+1} as it is defined in the **Write commit** transition of ARM+τ. The invariant is obviously preserved. By definition of map_τ, τ is unique among writes to the same location. The acyclicity of $\mathbf{s}^{i+1}.Ord \cup \mathsf{tedges}(\mathbf{s}^{i+1}.Evt, H_\tau^{i+1})$ holds by the same reason as in the previous case. By the acyclicity, τ is greater than timestamps of all writes to the same location, which are issued by *tid* to the storage. It is also greater than timestamps of previous writes, which do not have requests in the storage, as for each such write, there is a committed write with a larger timestamp. There are no following committed writes to the same location by the same thread, as the transition issues the request to the storage. Thus the timestamp is coherent with other thread writes.

If there is no write request issued to the storage, then $\mathsf{map}_\tau(tid, path) = \bot$. We know that there is a following write by the same thread to the same location with some timestamp τ'. We may choose the timestamp τ to be in $(\tau' - 1, \tau')$ in a way that it does not violate the transition preconditions. We choose H^{i+1} as it is defined in the **Write commit** transition. The invariant is obviously preserved.

Other transitions: We choose H^{i+1} to be equal to H^i. As there are no additional preconditions in the instrumented machine rules, and no changes in the additional components of the state, the instrumented machine can take the corresponding transition and the invariant is preserved.   ◀

8.3   View of the ARM+τ Machine

As we have seen for the MP-SY-LD example in Section 2, once a thread of the Promise machine reads from a write and then executes an acquire fence, the view of the thread gets updated with the message view of the write. The view update forbids subsequent reads to read from too old writes (with too small timestamps). To show a simulation between the Promise and the ARM+τ machines, we have to show a similar result for the ARM+τ machine.

We start with introducing $\mathsf{view}_{\mathrm{ARM}}$, an analog of $view_{\mathrm{cur}}$:

$$\mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, tid, path) \triangleq$$
$$\bigsqcup \mathsf{com\text{-}writes\text{-}time}(tid, path, \mathbf{a}.tapef(tid), \mathbf{a}.H_\tau) \sqcup$$
$$\bigsqcup \mathsf{sat\text{-}reads\text{-}view}(\mathsf{lastCF}(tape, path), \mathbf{a}.tapef(tid), \mathbf{a}.H_{\mathsf{view}})\,.$$

Unlike $view_{\mathrm{cur}}$ of the Promise machine, which is defined for a thread, $\mathsf{view}_{\mathrm{ARM}}$ is additionally parametrized by $path$ for the same reason that the variable state of the ARM machine is parametrized by $path$—the machine executes instructions out-of-order, so different instructions which might be executed at the same time have different τ-related restrictions. The definition itself is very similar to the definition of the H_{view} entry in the **Write commit** transition: it is a composition of singleton views corresponding to writes committed by the thread before $path$ and H_{view} entries corresponding to writes read by the thread before the last committed fence ($\mathsf{lastCF}(tape, path)$). We count reads up to any fence as both SY and LD ARM fences are strong enough to be a result of compilation of an acquire fence of the Promise machine.

Having this definition, we can define the aforementioned restrictions. If a read is satisfied from a committed write, the write has a timestamp which is greater than or equal to the corresponding value of $\mathsf{view}_{\mathrm{ARM}}$ at the read instruction instance. We do not restrict reads satisfied from not yet committed writes this way, as such writes do not have timestamps until they are committed. Similarly, each committed write has to have a timestamp which is greater than the value of $\mathsf{view}_{\mathrm{ARM}}$ at the write instruction instance:

▶ **Theorem 8.2.** $\forall Prog, \mathbf{a}, tid, tape = \mathbf{a}.tapef(tid), path.\ Prog \vdash \mathbf{a}^{\mathrm{init}} \xrightarrow[\mathrm{ARM+\tau}]{}^* \mathbf{a} \Rightarrow$

$(\forall e.\ tape(path) = R\ (\mathsf{sat}\ \_\ e) \wedge \mathbf{a}.tapef(e.tid, e.path)\ is\ committed \Rightarrow$
$\quad \mathbf{a}.H_\tau(e.tid, e.path) \geq \mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, tid, path, e.\ell)) \wedge$
$(\forall \ell.\ tape(path) = W\ (\mathsf{com}\ \_\ \ell\ \_) \Rightarrow \mathbf{a}.H_\tau(tid, path) > \mathsf{view}_{\mathrm{ARM}}(\mathbf{a}, tid, path, \ell)).$

The proof of the theorem can be found in the extended version of the paper [20].

9   The Compilation Correctness Proof

In this section, we prove the main theorem stated in Section 3.

▶ **Theorem 3.1.** *For all Prog and* $\mathbf{s}$, *if* $Prog \vdash \mathbf{s}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}]{}^* \mathbf{s}$ *and* $\mathsf{Final}^{\mathrm{ARM}}(\mathbf{s}, Prog)$, *then there exists* $\mathbf{p}$ *such that* $Prog \vdash \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}^* \mathbf{p}$ *where* $\mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, Prog)$ *and* $\mathsf{same\text{-}memory}(\mathbf{s}, \mathbf{p})$.

Here $\mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, Prog)$ means that the Promise machine cannot make a further transition (each thread's *path* points out of the thread's program instruction list) from $\mathbf{p}$ and all promises are fulfilled.

Proof. Let's fix the program *Prog*. In the remainder of the section we write "$\mathbf{s} \xrightarrow[\mathrm{ARM}]{} \mathbf{s}'$" instead of "$Prog \vdash \mathbf{s} \xrightarrow[\mathrm{ARM}]{} \mathbf{s}'$" for all machines. We apply the result of Section 8.2, and change the proof goal to the simulation for the ARM+τ machine:

$$\forall Prog, \mathbf{a}.\ \mathbf{a}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}+\tau]{}{}^{*} \mathbf{a} \wedge \mathsf{Final}^{\mathrm{ARM}+\tau}(\mathbf{a}, Prog) \Rightarrow$$
$$\exists \mathbf{p}.\ \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}{}^{*} \mathbf{p} \wedge \mathsf{Final}^{\mathrm{Promise}}(\mathbf{p}, Prog) \wedge \mathsf{same\text{-}memory}(\mathbf{a}, \mathbf{p}).$$

To prove it, we introduce a number of relations between ARM+τ and Promise states, which are parts of the simulation relation.

The $\mathcal{I}_{\mathrm{prefix}}$ relation states that every instruction instance, which has been executed by the Promise machine, has been executed by the ARM+τ machine:

$$\mathcal{I}_{\mathrm{prefix}}(\mathbf{a}, \mathbf{p}) \triangleq \forall tid, path' < \mathbf{p}.tsf(tid).path.\ \mathbf{a}.tapef(tid, path') \text{ is committed.}$$

The next relations connect the memories of the machines. $\mathcal{I}_{\mathrm{mem1}}$ states that for every write, which is committed by the ARM+τ machine, there is a message in the Promise memory to the same location with the same value and timestamp, and its view is less or equal to the corresponding view of the ARM request. If the *path* of the committed write is less than the corresponding thread's pointer to the next instruction ($\mathbf{p}.tsf(tid).path$), then the write is fulfilled, otherwise it is promised but not fulfilled:

$$\mathcal{I}_{\mathrm{mem1}}(\mathbf{a}, \mathbf{p}) \triangleq \forall tid, \ell, val, \tau, view', path.$$
$$\mathsf{W}\,(\mathsf{com}\ \_\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \langle \tau, \_, view' \rangle = \mathbf{a}.H(tid, path) \Rightarrow$$
$$\exists view \leq view'.$$
$$(path \geq \mathbf{p}.tsf(tid).path \Rightarrow \langle \ell : val@\tau, view \rangle \in \mathbf{p}.tsf(tid).promises) \wedge$$
$$(path < \mathbf{p}.tsf(tid).path \Rightarrow$$
$$\langle \ell : val@\tau, view \rangle \in \mathbf{p}.M_{\mathrm{Promise}} \setminus \bigcup_{tid} \mathbf{p}.tsf(tid).promises).$$

$\mathcal{I}_{\mathrm{mem2}}$ connects the memories in other direction: for every message in the Promise memory (except for initial ones) there is a committed write instruction instance in the ARM+τ machine:

$$\mathcal{I}_{\mathrm{mem2}}(\mathbf{a}, \mathbf{p}) \triangleq \forall \langle \ell : val@\tau, view \rangle \in \mathbf{p}.M_{\mathrm{Promise}}.\ \tau \neq \mathbf{0} \Rightarrow \exists tid, path, view' \geq view.$$
$$\mathsf{W}\,(\mathsf{com}\ \_\ \ell\ val) = \mathbf{a}.tapef(tid, path) \wedge \mathbf{a}.H(tid, path) = \langle \tau, \_, view' \rangle.$$

$\mathcal{I}_{\mathrm{mem3}}$ relates initial writes to locations:

$$\mathcal{I}_{\mathrm{mem3}}(\mathbf{a}, \mathbf{p}) \triangleq \forall \ell.\ \langle 0_{tid}, [], \mathsf{wr}\ \ell : 0 \rangle \in \mathbf{a}.M_{\mathrm{POP}} \wedge \langle \ell : 0@\mathbf{0}, \lambda\ell.\ \mathbf{0} \rangle \in \mathbf{p}.M_{\mathrm{Promise}}.$$

$\mathcal{I}_{\mathrm{view}}$ says that views of a Promise thread are restricted by the composition of singleton views of writes and reads committed by the ARM thread. For the acquire view, it counts all the writes and reads up to *path*. For the current view, it counts all the writes up to *path* and reads up to the latest committed LD fence ($\mathsf{lastLD}(tape, path)$). For the release view, it counts all writes up to the latest committed SY fence ($\mathsf{lastSY}(tape, path)$) and reads up to

the latest committed LD fence before the SY fence ($\mathsf{lastLDSY}(tape, path)$).

$\mathcal{I}_{\mathrm{view}}(\mathbf{a}, \mathbf{p}) \triangleq \forall tid, tape = \mathbf{a}.tapef(tid), path = \mathbf{p}.tsf(tid).path.$
   $\underline{\text{let }} path^{\mathsf{LD}}, path^{\mathsf{SY}} \triangleq \mathsf{lastLD}(tape, path), \mathsf{lastSY}(tape, path) \underline{\text{ in}}$
   $\underline{\text{let }} path^{\mathsf{LDSY}} \triangleq \mathsf{lastLDSY}(tape, path) \underline{\text{ in}}$
   $(\mathbf{p}.tsf(tid).view_{\mathrm{acq}} \leq \bigsqcup \mathsf{viewf}(tid, path, path, tape, \mathbf{a}.H)) \wedge$
   $(\mathbf{p}.tsf(tid).view_{\mathrm{cur}} \leq \bigsqcup \mathsf{viewf}(tid, path^{\mathsf{LD}}, path, tape, \mathbf{a}.H)) \wedge$
   $(\mathbf{p}.tsf(tid).view_{\mathrm{rel}} \leq \bigsqcup \mathsf{viewf}(tid, path^{\mathsf{LDSY}}, path^{\mathsf{SY}}, tape, \mathbf{a}.H).$

$\mathcal{I}_{\mathrm{state}}$ declares that a variable state of a Promise thread is the same as the committed state function up to the corresponding *path* of the ARM thread:

$\mathcal{I}_{\mathrm{state}}(\mathbf{a}, \mathbf{p}) \triangleq \forall tid, regf = \mathsf{regf}_{\mathrm{com}}(Prog(tid), \mathbf{a}.tapef(tid), \mathbf{p}.tsf(tid).path).$
   $\forall reg, \mathbf{p}.tsf(tid).st(reg) = regf(reg).$

$\mathcal{I}_{\mathrm{com\text{-}SY}}$ says that if an ARM thread committed a write, then all *path*-previous SY fences are executed by the corresponding Promise thread:

$\mathcal{I}_{\mathrm{com\text{-}SY}}(\mathbf{a}, \mathbf{p}) \triangleq \forall tid, tape = \mathbf{a}.tapef(tid),$
   $path_{write} = \mathsf{last\text{-}write\text{-}com}(tape), path_{\mathsf{SY}} < path_{write}.$
   $tape(path_{\mathsf{SY}}) = \mathsf{F} \_ \mathsf{SY} \Rightarrow path_{\mathsf{SY}} < \mathbf{p}.tsf(tid).path.$

where $\mathsf{last\text{-}write\text{-}com}(tape)$ is a *path* of a last write committed by the thread. The relation is necessary for certification of the Promise machine steps.
$\mathcal{I}_{\mathrm{reach}}$ asserts that states are reachable:

$\mathcal{I}_{\mathrm{reach}}(\mathbf{a}, \mathbf{p}) \triangleq \mathbf{a}^{\mathrm{init}} \xrightarrow[\mathrm{ARM}]{}^{*} \mathbf{a} \wedge \mathbf{p}^{\mathrm{init}} \xrightarrow[\mathrm{Promise}]{}^{*} \mathbf{p}.$

The relation $\mathcal{I}_{\mathrm{base}}$ combines the aforementioned relations:

$\mathcal{I}_{\mathrm{base}} \triangleq \mathcal{I}_{\mathrm{prefix}} \cap \mathcal{I}_{\mathrm{mem1}} \cap \mathcal{I}_{\mathrm{mem2}} \cap \mathcal{I}_{\mathrm{mem3}} \cap \mathcal{I}_{\mathrm{view}} \cap \mathcal{I}_{\mathrm{state}} \cap \mathcal{I}_{\mathrm{com\text{-}SY}} \cap \mathcal{I}_{\mathrm{reach}}.$

In the simulation, either the Promise machine is waiting for the next step of the ARM$+\tau$ machine, or there is a Promise thread which should make at least one non-**Promise write** step (corresponding to an instruction which the thread's *path* component is pointing to). A Promise thread *tid* is waiting for the corresponding ARM thread, if the next command to be executed, which is pointed by *path*, is not fetched or committed in the ARM thread.

$\mathcal{I}_{\mathrm{Promise\ is\ up\ to\ ARM}}^{tid}(tid, \mathbf{a}, \mathbf{p}) \triangleq \underline{\text{let }} tape, path \triangleq \mathbf{a}.tapef(tid), \mathbf{p}.tsf(tid).path \underline{\text{ in}}$
   $tape(path) = \bot \vee tape(path) \text{ is not committed.}$
$\mathcal{I}_{\mathrm{Promise\ is\ up\ to\ ARM}}\quad (\mathbf{a}, \mathbf{p}) \triangleq \forall \ tid. \ \ \mathcal{I}_{\mathrm{Promise\ is\ up\ to\ ARM}}^{tid}(tid, \mathbf{a}, \mathbf{p}).$
$\mathcal{I}_{\mathrm{Promise\ isn't\ up\ to\ ARM}}\ (\mathbf{a}, \mathbf{p}) \triangleq \exists! \ tid. \ \neg \mathcal{I}_{\mathrm{Promise\ is\ up\ to\ ARM}}^{tid}(tid, \mathbf{a}, \mathbf{p}).$

The relations are used to define two simulation relations:

$\mathcal{I}_{\mathrm{pre}} \triangleq \mathcal{I}_{\mathrm{base}} \cap \mathcal{I}_{\mathrm{Promise\ isn't\ up\ to\ ARM}} \qquad \mathcal{I} \triangleq \mathcal{I}_{\mathrm{base}} \cap \mathcal{I}_{\mathrm{Promise\ is\ up\ to\ ARM}}$

If the states are related by $\mathcal{I}_{\mathrm{pre}}$, there is a thread of the Promise machine which may take a step (which is not **Promise write**) by executing the next instruction its *path* is pointing to. After it either the thread has to make another step ($\mathcal{I}_{\mathrm{pre}}(\mathbf{a}, \mathbf{p}')$), or all threads of the Promise machine are waiting ($\mathcal{I}(\mathbf{a}, \mathbf{p}')$):

▶ **Lemma 9.1.** $\forall (\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\mathrm{pre}}. \ \exists \mathbf{p}'.\mathbf{p} \xrightarrow[\mathrm{Promise}]{} \mathbf{p}' \wedge (\mathbf{a}, \mathbf{p}') \in \mathcal{I}_{\mathrm{pre}} \cup \mathcal{I}.$

As at every specific state of the ARM+τ machine it has committed a finite number of instruction instances, we show that the Promise machine can make a finite number of transitions to get its state to satisfy $\mathcal{I}$:

▶ **Lemma 9.2.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}_{\text{pre}}. \ \exists \mathbf{p}'. \ \mathbf{p} \xrightarrow[\text{Promise}]{}^* \mathbf{p}' \wedge (\mathbf{a}, \mathbf{p}') \in \mathcal{I}.$

Suppose, the ARM+τ and Promise machine states are related by $\mathcal{I}$. Then, we show that the ARM+τ machine may make a step. If the step is **Write commit**, then the Promise machine has to promise the corresponding message, and the states of the machines are related by $\mathcal{I}_{\text{pre}} \cup \mathcal{I}$. If the step is not **Write commit**, then the Promise machine does not make a step, and the states are related by the same relation $\mathcal{I}_{\text{pre}} \cup \mathcal{I}$.

▶ **Lemma 9.3.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}.$
$(\forall \mathbf{a}'. \ \mathbf{a} \xrightarrow[\text{ARM+}\tau]{\neg \text{ Write commit}} \mathbf{a}' \Rightarrow (\mathbf{a}', \mathbf{p}) \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}) \wedge$
$(\forall \mathbf{a}'. \ \mathbf{a} \xrightarrow[\text{ARM+}\tau]{\text{Write commit}} \mathbf{a}' \Rightarrow \exists \mathbf{p}'. \ \mathbf{p} \xrightarrow[\text{Promise}]{\text{Promise write}} \mathbf{p}' \wedge (\mathbf{a}', \mathbf{p}') \in \mathcal{I}_{\text{pre}} \cup \mathcal{I}).$

Then, we state the following lemma:

▶ **Lemma 9.4.** $\forall(\mathbf{a}, \mathbf{p}) \in \mathcal{I}. \ \exists \mathbf{p}'. \ \mathbf{p} \xrightarrow[\text{Promise}]{}^* \mathbf{p}' \wedge (\mathbf{a}', \mathbf{p}') \in \mathcal{I}.$

It straightforwardly follows from the three previous lemmas.[9]

The theorem is proved by induction on the ARM+τ execution using Lemma 9.4. The machine memories are the same at the end due to $\mathcal{I}_{\text{mem1}}$ and $\mathcal{I}_{\text{mem2}}$. The only thing, which we need to show, is that $\mathsf{Final}^{\text{Promise}}(\mathbf{p}, \textit{Prog})$ holds.

The Promise machine cannot make a further step (each thread's *path* points out of the thread's instruction list), as otherwise the ARM+τ machine would be able to fetch a new instruction instance, and $\mathsf{Final}^{\text{ARM+}\tau}(\mathbf{a}, \textit{Prog})$ would not hold. Each thread has fulfilled its promises according to $\mathcal{I}_{\text{mem1}}$ and $\mathcal{I}_{\text{mem2}}$. ◀

10 Related Work

The most closely related work is the correctness proof of compilation from the Promise machine to the x86-TSO and Power models in the paper introducing the Promise machine [12].[10] Those proofs were much simpler than our proof essentially because these models are substantially simpler than the ARMv8 POP model. To simplify the correctness proof, Kang et al. use a result of Lahav and Vafeiadis [13], which reduces the soundness of compilation to proving soundness of certain local program transformations and of compilation with respect to stronger memory models (SC and Strong-Power respectively). Sadly, however, this reduction is not applicable to the ARMv8 POP model because of examples such as ARM-weak, in which ARM may execute anti-dependent instructions out of order. As a result, although Kang et al. do not use promise steps in the compilation part of their proof, promise steps *must* be used to justify the correctness of compilation to ARMv8 POP, which in turn renders our proof substantially more complicated than theirs.

In addition, there exist formal compilation proofs [6, 7] from the C++11 memory model to x86-TSO and Power, although the latter proof was recently found to be flawed in the case

---

[9] Proofs of the lemmas can be found in the extended version of the paper [20].

[10] Kang et al.'s proof for Power considers a compilation scheme that compiles acquire loads using Power's `lwsync`. This scheme is more expensive than the one implemented in existing compilers, which uses control dependency and `isync` for acquire loads.

of SC accesses [14, 17] indicating that the C++ semantics for SC accesses is too strong. This is also the reason why the Promise machine of Kang et al. [12] does not support SC accesses.

We introduced the intermediate ARM+τ machine to manage "lack of prescience", i.e., absence of information about a final ordering of write messages in the storage during an execution of the ARM machine. We could have used a backward simulation [16] and/or have treated the timestamp mapping component of the ARM+τ state as a prophecy variable [4] to establish a connection between the ARM and ARM+τ machines, but we found it easier to do the proof in a forward style.

Instead of proving the correctness of compilation schemes, one can resort to testing or model checking. Recently, Wickerson et al. [24] introduced an approach to automatically check different properties of weak memory models, including compilation. The tool generates all programs which size less than some given (small) parameter, and exhaustively checks all executions of those programs. Their approach, however, only works for memory models expressed in an axiomatic per-execution style, and is thus not directly applicable to neither the Promise nor the ARMv8 POP semantics.

11   Conclusion

In this paper, we have proved soundness of the compilation of relaxed loads and stores, as well as release and acquire fences, from the Promise machine to the ARMv8 POP machine. Since the proof is already significantly complex, we have not attempted to model all the features of the Promise machine. Specifically, we have not considered the compilation of release/acquire accesses, read-modify-write (RMW) instructions, and SC fences. Extending the proof to cover these instructions and mechanizing it are left for future work. In the remainder of this section, we outline the issues involved in extending our proof.

Another useful item for future work would be to consider the correctness of compilation from the Promise machine to the newer stronger ARMv8.2 model [1, 3]. As, however, the new model is in many regards substantially stronger than ARMv8 POP, the compilation proof should be much easier.

Handling Release and Acquire Accesses   There are two proposed compilation schemes for release and acquire accesses [2]. A one of them involves fences considered in the paper:

$$
\begin{array}{r|l}
\textbf{Promise:} \quad a := [x]_{\mathsf{acq}} & [x]_{\mathsf{rel}} := a \\
\textbf{ARM:} \quad a := [x];\ \mathsf{dmb\ LD} & \mathsf{dmb\ SY};\ [x] := a
\end{array}
$$

Compilation correctness for $a := [x]_{\mathsf{acq}}$ straightforwardly follows from results of Kang et al. [12] and the current paper, as a transformation $a := [x]_{\mathsf{acq}} \rightsquigarrow a := [x]_{\mathsf{rlx}};\ \mathsf{fence}(\mathsf{acquire})$ is sound for the Promise machine. To cover the aforementioned mapping of $[x]_{\mathsf{rel}} := a$, one should be able to restrict the ARM machine to commit writes, which directly follow SY fences, right after committing the fences without losing any observable behaviors. Then, the compilation correctness proof is a straightforward extension of the current proof.

Another compilation scheme uses special *acquire* ($a := [x]_{\mathsf{LDAR}}$) and *release* ($[x]_{\mathsf{STLR}} := a$) ARM instructions, which were originally introduced to the architecture to cover SC accesses:

$$
\begin{array}{r|l}
\textbf{Promise:} \quad a := [x]_{\mathsf{acq}} & [x]_{\mathsf{rel}} := a \\
\textbf{ARM:} \quad a := [x]_{\mathsf{LDAR}} & [x]_{\mathsf{STLR}} := a
\end{array}
$$

These instructions induce rather strong synchronization. For instance, the ARM acquire reads forbid program-following instructions to issue requests until the reads are satisfied, and

any satisfied acquire read requests are not removed from the storage, but start to act as a fence request, i.e., become impossible to reorder with anything. To cover them one would need to extend our definition of the ARM machine and Theorem 8.2.

Handling Read-Modify-Writes   The Promise machine with RMW instructions represents message timestamps as ranges of rational numbers, and maintains the invariant that all messages in memory have disjoint timestamp ranges. If there is a message with a timestamp $(\tau, \tau']$ in the memory and a thread T executes an RMW operation, which reads from that message, the RMW message gets a timestamp range $(\tau', \tau'']$ for some $\tau'' > \tau$, which prevents other threads from adding a message "in-between" in future.

RMWs are compiled to ARM as a combination of an *exclusive* load followed by an *exclusive* store, typically inside a loop. The ARM-POP model [9] guarantees that when an exclusive store issues a write request e_{excl} to the storage, there is no write request to the same location *Ord*-between this write request and the write request e_{prev} read by the corresponding exclusive load. The machine guarantees that the property is preserved during an ongoing execution as well. This enables us to keep the same timestamp representation in the ARM+τ machine: as all write requests in the storage of the ARM+τ machine have integer timestamps, it is easy to show that the timestamp of e_{excl} is equal to the timestamp of e_{prev} increased by one. In the simulation, when the ARM+τ machine commits a *exclusive* store with a timestamp τ, the Promise machine will promise a RMW with a timestamp range $(\tau - 1, \tau]$.

A slight difficulty is that once RMWs are added to the source language, the compilation scheme is no longer bijective, as RMWs get compiled to a sequence of ARM instructions. For example, a compare-and-swap instruction $\mathsf{cas}(\ell, val_{old}, val_{new})$ may be compiled to the following loop (on the left):

Loop :	$a := \mathsf{load_{excl}}(\ell);$		Loop :	$a \ := \ [\ell];$	
	if $a = val_{old}$ goto Exit;			if $a = val_{old}$ goto Exit;	
	$\mathsf{store_{excl}}(flag, \ell, val_{new});$			$flag := \mathsf{cas_{restricted}}(\ell, val_{old}, val_{new});$	
	if $flag = 0$ goto Loop;			if $flag = 0$ goto Loop;	
Exit :			Exit :		

For the sake of preserving a simple mapping between source and target programs, one might introduce a restricted version of the CAS instruction in the Promise machine. This restricted CAS would be allowed to read only from a write read by a previous load instruction, i.e., the write whose timestamp is equal (not greater or equal) to the corresponding value of the thread's current view. After that, one may show that the program transformation that replaces $\mathsf{cas}(\ell, val_{old}, val_{new})$ with the loop shown above (on the right) is sound for the extended Promise machine and prove compilation correctness for the extended machine.

Handling SC Fences   The Promise machine uses a global view to support SC fences. When a thread executes an SC fence, it synchronizes its own views with the global view, i.e., assigns to all of them (including the global one) the pointwise maximum. This models an existence of a global order on SC fences.

SC fences are compiled to $\mathsf{dmb}\ \mathsf{SY}$ fences in the ARM machine. As with write requests, $\mathsf{dmb}\ \mathsf{SY}$ fences are definitely ordered by *Ord* only at the end of an execution. Consequently, one needs to extend the ARM+τ machine to calculate timestamps for $\mathsf{dmb}\ \mathsf{SY}$ fences.

This, however, does not solve all problems. Currently in the simulation, when the ARM+τ machine commits a $\mathsf{dmb}\ \mathsf{SY}$ instruction, the Promise machine executes the corresponding

release fence instruction. Doing this is necessary, because it enables promising program-following writes, when the ARM+τ machine commits them to the storage. In the same situation, however, the Promise machine may not be able to execute the corresponding SC fence, because the Promise machine has to execute them in the newly introduced timestamp order, which may not coincide with a commit order of the ARM+τ execution.

—— **References** ——

1 ARM architecture reference manual: ARMv8, for ARMv8-A architecture profile. Available at `https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile` [Online; accessed 16-May-2017].

2 C/C++11 mappings to processors. Available at `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`. [Online; accessed 16-May-2017].

3 The `herdtools7` repository. Available at `https://github.com/herd/herdtools7` [Online; accessed 16-May-2017].

4 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. `doi:10.1109/LICS.1988.5115`.

5 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. `doi:10.1145/2627752`.

6 Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *ESOP*, volume 9032 of *LNCS*, pages 283–307. Springer, 2015. `doi:10.1007/978-3-662-46669-8_12`.

7 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL 2011*, pages 55–66. ACM, 2011. `doi:10.1145/1925844.1926394`.

8 Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC 2014*, pages 7:1–7:6. ACM, 2014. `doi:10.1145/2618128.2618134`.

9 Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL 2016*, pages 608–621. ACM, 2016. `doi:10.1145/2837614.2837615`.

10 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

11 Alan Jeffrey and James Riely. On thin air reads: Towards an event structures model of relaxed memory. In *LICS 2016*. IEEE, 2016. `doi:10.1145/2933575.2934536`.

12 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL 2017*. ACM, 2017. `doi:10.1145/3009837.3009850`.

13 Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In *FM 2016*. Springer, 2016. `doi:10.1007/978-3-319-48989-6_29`.

14 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*. ACM, 2017.

15 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. `doi:10.1109/TC.1979.1675439`.

16   Nancy Lynch and Frits Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. `doi:10.1006/inco.1995.1134`.

17   Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016. URL: `http://arxiv.org/abs/1611.01507`.

18   Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL 2005*, pages 378–391. ACM, 2005. `doi:10.1145/1040305.1040336`.

19   Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, pages 622–633. ACM, 2016. `doi:10.1145/2837614.2837616`.

20   Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Promising compilation to ARMv8 POP. Extended version, 2017. Available at `http://podkopaev.net/armpromise` [Online; accessed 16-May-2017].

21   Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016. URL: `http://arxiv.org/abs/1606.01400`.

22   Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI 2011*, pages 175–186. ACM, 2011. `doi:10.1145/1993498.1993520`.

23   Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. `doi:10.1145/1785414.1785443`.

24   John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL 2017*. ACM, 2017. `doi:10.1145/3009837.3009838`.

25   Yang Zhang and Xinyu Feng. An operational happens-before memory model. *Frontiers of Computer Science*, 10(1):54–81, 2016. `doi:10.1007/s11704-015-4492-4`.