Bounding Data Races in Space and Time

(Extended working version, with appendices)

Stephen Dolan
University of Cambridge, UK

KC Sivaramakrishnan
University of Cambridge, UK

Anil Madhavapeddy
University of Cambridge, UK

Abstract

We propose a new semantics for shared-memory parallel programs that gives strong guarantees even in the presence of data races. Our *local data race freedom* property guarantees that all data-race-free portions of programs exhibit sequential semantics. We provide a straightforward operational semantics and an equivalent axiomatic model, and evaluate an implementation for the OCaml programming language. Our evaluation demonstrates that it is possible to balance a comprehensible memory model with a reasonable (no overhead on x86, ~0.6% on ARM) sequential performance trade-off in a mainstream programming language.

CCS Concepts • **Computing methodologies → Shared memory algorithms**; • **Theory of computation → Parallel computing models**;

Keywords weak memory models, operational semantics

ACM Reference Format:
Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time: (Extended working version, with appendices). In *Proceedings of working draft ('18)*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern processors and compilers aggressively optimise programs. These optimisations accelerate without otherwise affecting sequential programs, but cause surprising behaviours to be visible in parallel programs. To benefit from these optimisations, mainstream languages such as C++ and Java have adopted complicated memory models which specify which of these *relaxed behaviours* programs may observe. However, these models are difficult to program against directly.

The primary reasoning tools provided to programmers by these models are the *data-race-freedom (DRF) theorems*. Programmers are required to mark as *atomic* all variables used for synchronisation between threads, and to avoid *data races*, which are concurrent accesses (except concurrent reads) to nonatomic variables. In return, the DRF theorems guarantee that no relaxed behaviour will be observed. Concisely, *data-race-free programs have sequential semantics*.

When programs are not data-race-free, such models give few or no guarantees about behaviour. This fits well with unsafe languages, where misuse of language constructs generally leads to undefined behaviour. Extending this to data races, another sort of misuse, is quite natural. On the other hand, safe languages strive to give well-defined semantics even to buggy programs. These semantics are expected to be *compositional*, so that programs can be understood by understanding their parts, even if some parts contain bugs.

Giving weak semantics to data races threatens this compositionality. In a safe language, when `f() + g()` returns the wrong answer even when `f()` returns the right one, one can conclude that `g` has a bug. This property is threatened by weak semantics for data races, when a correct `g` could be caused to return the wrong answer by a data race in `f`.

We propose a new semantics for shared-memory parallel programs, which gives strong guarantees even in the face of data races. Our contributions are to:

- introduce the *local DRF* property (§2), which allows compositional reasoning about concurrent programs even in the presence of data races.
- propose a memory model with a straightforward small-step operational semantics (§3), prove that it has the local DRF property (§4), and provide an equivalent axiomatic model (§6).
- show that our model supports many common compiler optimisations and provide sound compilation schemes to both the x86 and ARMv8 architectures (§7), and demonstrate their efficiency in practice in the hybrid functional-imperative language OCaml (§8).

2 Reasoning Beyond Data-Race Freedom

We propose moving from the *global DRF* property:

Data-race-free programs have
sequential semantics

to the stronger *local DRF* property:

All data-race-free parts of programs have
sequential semantics

To demonstrate the difference between global and local DRF, we present several examples of sequential program fragments, and multithreaded contexts in which a lack of local DRF causes unexpected results.

2.1 Bounding Data Races in Space

The first step towards local DRF is *bounding data races in space*, ensuring that a data race on one variable does not affect accesses to a different variable.

'18, August 2018, Cambridge, UK
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Since the C++ memory model gives semantics only to data-race-free programs, in principle it does not have this property. Still, it is not obvious how this property could fail to hold in a reasonable implementation, so we give an example.

Example 1. `b = a + 10`

Assumption There are no other accesses to `a` or `b`.
Expected result Afterwards, `b` = `a + 10`.
Possible result Afterwards, `b` ≠ `a + 10`. (C++)

Explanation Consider the following multithreaded program, where `c` is a nonatomic global variable:

```
c = a + 10;                 ||
... // some computation     || c = 1;
b = a + 10;                 ||
```

Suppose that the elided computation is pure (writes no memory and has no side-effects). The compiler might notice that `a` is not modified between its two reads, and thus occurrences of `a + 10` may be combined, optimising the first thread to:

```
t = a + 10;
c = t;
... // some computation
b = t;
```

Register pressure in the elided computation can cause the temporary `t` to be spilled. Since its value is already stored in location `c`, a clever register allocator may choose to rematerialise `t` from `c` instead of allocating a new stack slot[1], giving:

```
t = a + 10;                 ||
c = t;                      ||
... // some computation     || c = 1;
b = c;                      ||
```

However, in the transformed program, the data race between `c = a + 10` and `c = 1` may cause `c` to contain the wrong value. From the programmer's point of view, two reads of location `a` returned two different values, even though there were no concurrent writes to `a`! Indeed, the only data race is the two concurrent writes to `c`, a variable which is never read.

A data race on one variable affecting the results of reading another is far from the worst effect that compiler optimisations can bestow on racy C++ programs. Boehm [9] gives several others, but this one suffices to show bounding data races in space is a nontrivial property, and that reasonable implementations of C++ do not necessarily possess it.

2.2 Bounding Data Races in Time

In contrast to C++, the Java memory model [16] limits the allowable behaviours even in the presence of data races. In particular, the value that is returned by reading a variable must be something written to the same variable, so data races are indeed bounded in space.

However, data races in Java are not *bounded in time*: a data race in the past can cause later accesses to have non-sequential behaviour, as in the following example. Below, when we say that two accesses *happen concurrently*, we mean that neither *happens-before* the other using the ordering defined by the memory model. Roughly, this means that the two accesses occur in different threads without any synchronisation between the threads.

Example 2. `b = a; c = a;`

Assumption No accesses to `a`, `b`, `c` happen concurrently with the above.
Expected result Afterwards, `b` = `c`.
Possible result Afterwards, `b` ≠ `c`. (C++, Java)

Explanation Consider the program below, where `flag` is an atomic (`volatile` in Java) boolean, initially `false`.

```
a = 1;          || a = 2;
flag = true;    || f = flag;
                || b = a;
                || c = a;
```

Suppose that `f` is `true` afterwards. Then the read and the write of `flag` synchronise with each other (`flag` being `volatile`), so both of the `writes` to `a` happen-before both of the reads, although the writes race with each other.

In such circumstances, the assumption above does hold: there are no accesses to `a`, `b`, `c` that happen concurrently with the reads of `a`. However, Java permits the outcome `b = 1`, `c = 2`, allowing the two reads to read from the two different writes. Concretely, this can happen if the compiler optimises `b = a` to `b = 2` without making the same change to `c = a`. This situation can occur due to aliasing, if only the first read from `b = a` is statically known to be the same location as that written by `a = 2`. A concrete example triggering this behaviour under both Java 8 and 9 appears in appendix D of the technical report [11].

So, the effect of data races in Java is not *bounded in time*, because the memory model permits reads to return inconsistent values because of a data race that happened in the past. Surprisingly, non-sequential behaviour can also occur because of data races in the future, as in the following example. We assume the prior definition of `class C {int x;}`.

Example 3. `C c = new C(); c.x = 42; a = c.x;`

Assumption There are no other accesses to `a`.
Expected result Afterwards, `a` = 42.
Possible result Afterwards, `a` ≠ 42. (C++, Java)

Explanation Here, we know that there cannot be any data races in the past on the location `c.x`, since `c` is a newly-allocated object, to which no other thread could yet have a reference. So, we might imagine that this fragment will always set `a` to 42, regardless of what races are present in the rest of the program.

[1] This is an optimisation not generally implemented, because of the effort involved in preserving information about the contents of memory all the way to register allocation, but has been proposed for addition to LLVM [15].

In fact, it is possible for a to get a value other than 42, because of subsequent data races. Consider this pair of threads:

```
C c = new C();  ||
c.x = 42;       ||
a = c.x;        ||
g = c;          ||  g.x = 7;
```

The read of c.x and the write of g performed by the first thread operate on separate locations, so the Java memory model permits them to be reordered. This can cause the read of c.x to return 7, as written by the second thread.

So, providing local DRF requires us to prevent loads being reordered with later stores, which constrains both compiler optimisations and compilation to weakly-ordered hardware. We examine the performance cost of these constraints in detail in §8, and revisit the topic in §9.1.

2.3 Global and Local DRF

We propose a local DRF property which states that data races are bounded in space and time: accesses to variables are not affected by data races on other variables, data races in the past, or data races in the future. In particular, the following intuitive property holds:

> *If a location* a *is read twice by the same thread, and there are no concurrent writes to* a*, then both reads return the same value.*

We formally state the local DRF theorem for our model in §4, after introducing the operational semantics in §3. Detailed proofs appear in the appendix. Thanks to the local DRF theorem, we can prove that each of the examples above has the expected behaviour (see §5).

Using the standard global DRF theorems, we are able to prove that each of the three examples above have the expected behaviour, but only under the stronger assumption that there are no data races on any variables at any time during the program's execution. Local DRF allows us to prove the same results, but under more general assumptions that are robust to the presence of data races in other parts of the program.

3 A Simple Operational Model

Here, we introduce the formal memory model for which we prove local DRF in §4. Our model is an small-step operational one, where memory consists of *locations* $\ell \in \mathcal{L}$, divided into *atomic locations* A, B, . . . and *nonatomic locations* a, b, . . . , in which may be stored *values* $x, y \in \mathcal{V}$.

The program interacts with memory by performing *actions* ϕ on locations. There are two types of action: write x, which writes the value x to a location, and read x, which reads a location, resulting in the value x. We write $\ell : \phi$ for the action ϕ applied to the location ℓ.

Memory itself is represented by a *store* S. Under a sequentially consistent semantics, the store simply maps locations to values. Our semantics is not sequentially consistent, and the form of stores is more complex, since there is not necessarily a single value that a read of a location must return.

Instead, our stores map nonatomic locations a to *histories* H, which are finite maps from timestamps t to values x. Following Kang et al. [13], we take timestamps to be rational numbers rather than integers: they are totally ordered but dense, with a timestamp between any two others. Again following Kang et al., we equip every thread with a *frontier* F, which is a map from nonatomic locations to timestamps. Intuitively, each thread's frontier records, for each nonatomic location, the latest write known to the thread. More recent writes may have occurred, but are not guaranteed to be visible.

Atomic locations, on the other hand, are mapped by the store to a pair (F, x), containing a single value x rather than a history. Additionally, atomic locations carry a frontier, which is merged with the frontiers of threads that operate on the location. In this way, nonatomic writes made by one thread can become known to another by communicating via an atomic location.

The core of the semantics is the memory operation relation

$$C; F \overset{\ell:\phi}{\twoheadrightarrow} C'; F'$$

which specifies that when a thread with frontier F performs an action ϕ on location ℓ containing contents C, then the new contents of the location will be C' and the thread's new frontier will be F'.

There are four cases, for read and write, atomic and nonatomic actions, shown in fig. 1c. When reading a nonatomic variable, rule Read-NA specifies that threads may read an arbitrary element of the history, as long as it is not older than the timestamp in the thread's frontier.

Dually, when writing to a nonatomic location, rule Write-NA specifies that the timestamp of the new entry in the location's history must be later than that in the thread's frontier. Note a subtlety here: the timestamp need not be later than everything else in the history, merely later than any other write known to the writing thread.

Atomic operations (rules Read-AT and Write-AT) are standard sequential operations, except that they also involve updating frontiers. During atomic writes, the frontiers of the location and the thread are merged, while during atomic reads the frontier of the location is merged into that of the thread, but the location is unmodified. The join operation $F_1 \sqcup F_2$ combines two frontiers F_1, F_2 by choosing the later timestamp for each location.

The program itself consists of *expressions* e, e'. Our semantics of memory does not specify the exact form of expressions, but we assume they are equipped with a small-step transition relation $\rightsquigarrow$. A step may or may not involve performing an action, giving two distinct types of transition:

$$e \overset{\epsilon}{\rightsquigarrow} e' \qquad e \overset{\ell:\phi}{\rightsquigarrow} e'$$

$$\begin{array}{rrcl}
\text{Location} & l & \in & \mathcal{L} \\
\text{Atomic locations} & A, B, \ldots & \in & \mathcal{L} \\
\text{Nonatomic locations} & a, b, \ldots & \in & \mathcal{L} \\
\text{Timestamp} & t & \in & \mathbb{Q} \\
\text{Values} & x, y & \in & \mathcal{V} \\
\text{Thread id} & i & & \\
\text{Expression} & e, e' & &
\end{array}
\qquad
\begin{array}{rrcl}
\text{Machine} & M & := & \langle S, P \rangle \\
\text{Store} & S & := & a \mapsto H \uplus A \mapsto (F, x) \\
\text{History} & H & := & t \mapsto x \\
\text{Frontier} & F & := & a \mapsto t \\
\text{Program} & P & := & i \mapsto (F, e)
\end{array}$$

(a) Syntax and configurations

$$(\textsc{Silent}) \quad \frac{e \overset{\epsilon}{\rightsquigarrow} e'}{\langle S, P[i \mapsto (F, e)] \rangle \to \big\langle S, P[i \mapsto (F, e')] \big\rangle}$$

$$(\textsc{Memory}) \quad \frac{e \overset{\ell:\phi}{\rightsquigarrow} e' \qquad S(\ell); F \overset{\ell:\phi}{\twoheadrightarrow} C'; F'}{\langle S, P[i \mapsto (F, e)] \rangle \to \big\langle S[\ell \mapsto C'], P[i \mapsto (F', e')] \big\rangle}$$

(b) Machine steps

$$\begin{array}{lrcl}
(\textsc{Read-NA}) & H; F & \xrightarrow{\mathsf{a}:\,\mathsf{read}\ H(t)} & H; F \\
 & & \multicolumn{2}{l}{\text{if } F(\mathsf{a}) \le t,\ t \in \mathsf{dom}(H)} \\
(\textsc{Write-NA}) & H; F & \xrightarrow{\mathsf{a}:\,\mathsf{write}\ x} & H[t \mapsto x]; F[\mathsf{a} \mapsto t] \\
 & & \multicolumn{2}{l}{\text{if } F(\mathsf{a}) < t,\ t \notin \mathsf{dom}(H)} \\
(\textsc{Read-AT}) & (F_{\mathsf{A}}, x); F & \xrightarrow{\mathsf{A}:\,\mathsf{read}\ x} & (F_{\mathsf{A}}, x); F_{\mathsf{A}} \sqcup F \\
(\textsc{Write-AT}) & (F_{\mathsf{A}}, y); F & \xrightarrow{\mathsf{A}:\,\mathsf{write}\ x} & (F_{\mathsf{A}} \sqcup F, x); F_{\mathsf{A}} \sqcup F
\end{array}$$

(c) Memory operations

Figure 1. Operational semantics

where ϵ represents *silent* transitions, those that do not access memory. The only condition that we do assume of these transitions is that read transitions are not picky about the value being read, that is:

Proposition 4. *If $e \overset{\ell:\mathsf{read}\ x}{\rightsquigarrow} e'$, then for every y, $e \overset{\ell:\mathsf{read}\ y}{\rightsquigarrow} e_y$ for some e_y.*

We don't require that e_y not get stuck later, just that the read itself can progress.

The operational semantics is a small-step relation on *machine configurations* $M = \langle S, P \rangle$, consisting of a store S and a *program* P, which consists of a finite set of *threads*, represented as a finite map from thread identifier i to a pair of a frontier F and an expression e.

The two types of transitions in this small-step relation are shown in fig. 1b, and correspond to the two types of transitions for expressions. If a thread i with state (F, e) can take a silent step by $e \overset{\epsilon}{\rightsquigarrow} e'$, then its new state is (F, e') (rule Silent). Otherwise, if it can step by $e \overset{\ell:\phi}{\rightsquigarrow} e'$, then the memory operation relation determines the thread's new frontier and the new contents of ℓ (rule Memory).

3.1 Initial States

For simplicity, we assume that all locations are initially set to some arbitrary value $v_0 \in \mathcal{V}$. The initial state of a program whose threads are the expressions e_i (for i drawn from some finite set of thread indices $\mathcal{I}$) is the machine configuration:

$$M_0 = \left\langle \begin{array}{l} (\mathsf{a} \mapsto (0 \mapsto v_0), \mathsf{A} \mapsto (F_0, v_0) \text{ for } \mathsf{a}, \mathsf{A} \in \mathcal{L}), \\ (i \mapsto F_0, e_i \text{ for } i \in \mathcal{I}) \end{array} \right\rangle$$

The initial frontier F_0 maps all locations a to the timestamp 0. In other words, we assume an initial write of v_0 to every location, with timestamp 0 (for nonatomic locations), and we assume that these initial writes are part of every thread's frontier at startup.

3.2 Traces

We write a machine step T from a machine state M to a machine state M' as $M \xrightarrow{T} M'$.

Definition 5 (Trace)**.** *A trace*

$$\Sigma = M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} M_n$$

is a finite sequence of machine transitions starting from the initial state.

We do not have any requirement that traces lead to final states. Every prefix of a trace is a trace.

4 Formalising Local DRF

Next, we state and prove the local DRF theorem for the model, which states that all data-race-free parts of programs have sequential behaviour. More specifically, we show that if there are no ongoing data races on some set L of locations, then accesses to L will have sequential behaviour, at least until a data race on L occurs.

We need several intermediate definitions before we can prove local DRF. First, to specify what "sequential behaviour" means, we introduce *weak transitions*, and second, to specify what "no ongoing data races" means we introduce *L-stability*.

4.1 Weak Transitions

Our is close to a sequential model of memory, only differing during transitions READ-NA and WRITE-NA. We make this precise by defining *weak transitions*:

Definition 6 (Weak transition). *A weak transition is a machine step performing a memory operation of one of the following forms:*

- $H; F \xrightarrow{\text{a:read } x} H; F$ *when* $H(t) \neq x$ *for the largest timestamp* $t \in \text{dom}(H)$*. Informally, this read does not witness the latest write in that location.*
- $H; F \xrightarrow{\text{a:write } x} H[t \mapsto x]; F'$ *when* t *is not greater than the largest timestamp* $t' \in H$*. Informally, this write is not the latest write to that location.*

Memory operations which are not weak are either operations on atomic values, or operations on nonatomic values which access the element of history with the largest timestamp. So, a sequence of machine steps involving no weak transitions is sequentially consistent: one may ignore all frontiers and discard all elements of histories but the last, and recover a simple sequential semantics. We take this as our *definition* of sequential consistency:

Definition 7 (Sequentially consistent traces). *A trace is sequentially consistent if it includes no weak transitions.*

4.2 Data Races and Happens-Before

Intuitively, a data race occurs whenever a nonatomic location is used by multiple threads without proper synchronisation. To define what "proper synchronisation" means, we introduce the *happens-before* relation.

Definition 8 (Happens-before). *Given a trace*

$$M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \dots \xrightarrow{T_n} M_n$$

the happens-before relation is the smallest transitive relation which relates T_i, T_j, $i < j$ *if*

- T_i *and* T_j *occur on the same thread*
- T_i *is a write and* T_j *is a read or write, to the same atomic location.*

Definition 9 (Conflicting transitions). *In a given trace, two transitions* T_i *and* T_j *are conflicting if they access the same nonatomic location and at least one is a write.*

Definition 10 (Data race). *Given a trace*

$$M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \dots \xrightarrow{T_n} M_n$$

we say that it there is a data race between two conflicting transitions T_i *and* T_j *if* $i < j$ *and* T_i *does not happen-before* T_j.

4.3 L-stability

Definition 11 (L-sequential transitions). *Given a set* L *of locations, a transition is* L*-sequential if it is not a weak transition, or if it is a weak transition on a location not in* L.

If we take L to be the set of all nonatomic locations, then L-sequential transitions are exactly the sequentially consistent transitions.

Definition 12 (L-stable). *A machine* M *is* L*-stable if, for all traces that include* M:

$$M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \dots \xrightarrow{T_n} M \xrightarrow{T_1'} M_1' \xrightarrow{T_2'} \dots \xrightarrow{T_n'} M_n'$$

in which the transitions T_i' *are* L*-sequential, then there is no data race between* T_i *and* T_j'*, for any* i, j.

Intuitively, M is L-stable if there are no data races on locations in L in progress when the program reaches state M. There may be data races before reaching M (as in example 2), there may be data races after reaching M (as in example 3), but there are no data races between one operation before M and one operation afterwards.

4.4 The Local DRF Theorem

Theorem 13 (Local DRF). *Given an* L*-stable machine state* M *(not necessarily the initial state), and a sequence of* L*-sequential machine transitions:*

$$M \xrightarrow{T_1} M_1 \xrightarrow{T_2} \dots \xrightarrow{T_n} M_n$$

then either:

- *all possible transitions* $M_n \xrightarrow{T'} M'$ *are* L*-sequential, or*
- *there is a non-weak transition* $M_n \xrightarrow{T'} M'$*, accessing a location in* L*, with a data race between some* T_i *and* T'

5 Reasoning with Local DRF

Here, we give several examples of reasoning with the local DRF theorem. First, we use it to prove the standard global DRF result, justifying our claim to be more general. Second, we use it to show that the examples of §2 have the expected semantics, and do not exhibit the odd behaviours of the C++ and Java models.

The standard global DRF theorem is that data-race-free programs have sequential semantics, which we formalise as follows:

Theorem 14 (DRF). *Suppose that a given program is data-race-free. That is, suppose that all sequentially consistent traces starting from the initial machine state contain no data races. Then all traces starting from the initial state are sequentially consistent traces.*

Proof. Suppose we have a non-sequentially-consistent trace, whose first weak transition is T':

$$M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \dots \xrightarrow{T_n} M_n \xrightarrow{T'} M'$$

Applying local DRF with L as the set of all locations, either:

- T' is L-sequential, contradicting that T' is weak; or
- There is a sequential transition racing with some T_i, contradicting that the program is data-race-free.

□

Next, we recall the example fragments from §2:

1. `b = a + 10;`
2. `b = a; c = a;`
3. `C c = new C(); c.x = 42; a = c.x;`

Sequentially, we would expect these fragments to result in the following:

1. Afterwards, `b` = `a + 10`
   (assuming no other accesses to `a` or `b`)
2. Afterwards, `b` = `c`
   (assuming no concurrent accesses to `a`, `b` or `c`)
3. Afterwards, `a` = 42
   (assuming no other accesses to `a`)

Despite the global DRF theorems, we saw that these fragments do not always have sequential behaviour in C++ and Java, due to:

1. Data races on a third location `c` (C++ only)
2. Prior data races on `a`
3. Subsequent data races on `a`

However, we can apply local DRF to show that these fragments have the expected results. A good rule of thumb when applying local DRF to a fragment of code is to choose L to contain every location accessed by the fragment, giving:

1. $L = \{\mathsf{a}, \mathsf{b}\}$
2. $L = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$
3. $L = \{\mathsf{a}, \mathsf{c}, \mathsf{c.x}\}$

Next, we must argue that the machine state before executing these fragments is always L-stable. This is true for fragments 1 and 2 since our assumptions imply that there are no concurrent accesses to locations in L before executing these fragments. For fragment 3, there are no concurrent accesses to `a` before executing the fragment due to our assumption. There are no concurrent accesses to `c` or `c.x` since `c` is a newly allocated object.

Local DRF then states that the fragments will have sequential behaviour at least until the next data race on L. Since the fragments themselves contain no data races on L, this implies that they will have the expected behaviour.

6 Axiomatic Semantics

As well as the operational model of §3, we provide an axiomatic model and use it to verify our compilation schemes to hardware (sections 7.2 and 7.3), characterise allowed instruction reorderings by compiler optimisations (§7.1), and for comparison with other memory models (§9.2).

Instead of traces of a machine state, the axiomatic semantics represents program behaviour by a set of *events* $E = (k, \ell, \phi)$, where k is an *event identifier*, ℓ is a location and ϕ is an action. We say that E is a *read event* with value x if $\phi = \mathsf{read}\ x$, and similarly for *write events*.

The event identifiers k are of one of two forms: either a pair (i, n), indicating the nth event performed in program order by thread i, or else IW_ℓ, indicating the *initial write* of v_0 to location ℓ performed before program start.

$$(\textsc{Silent-G})\qquad \frac{e \overset{\epsilon}{\rightsquigarrow} e'}{\langle G, P[i \mapsto (n,e)]\rangle \to \big\langle G, P[i \mapsto (n,e')]\big\rangle}$$

$$(\textsc{Memory-G})\qquad \frac{e \overset{\ell:\phi}{\rightsquigarrow} e'}{\begin{array}{c}\langle G, P[i \mapsto (n,e)]\rangle \to \\ \Big\langle\ G \cup \{((i,n),\ell,\phi)\}, P[i \mapsto (n+1,e')]\ \Big\rangle\end{array}}$$

Figure 2. Generating events from programs

In the axiomatic semantics, events are generated from program execution, building a finite set of events G (called an *event graph*) according to the rules in fig. 2. In these rules, the program P is a finite map of thread identifiers i to pairs (n, e), where e is the current expression of the thread and n is the number of events already produced by that thread.

The initial event graph G_0 contains only the initial writes, corresponding to the initial machine state M_0:

$$G_0 = \{(\mathsf{IW}_\ell, \ell, v_0) \mid \ell \in \mathcal{L}\}$$

The event graphs G generated by the rules of fig. 2 include all possible executions of the program, as well as many nonsensical executions. The axiomatic semantics then restricts the possible event graphs to the *consistent executions*.

The definition of consistent executions is done in two stages. First, we define the intermediate notion of *candidate execution*, which is an event graph G equipped with binary relations po (program order), rf (reads-from) and co (coherence), such that the following conditions hold:

- po relates events with identifiers (i_1, n_1) and (i_2, n_2) if $i_1 = i_2$ and $n_1 < n_2$.
- If E_W rf E_R, then E_W is a write event and E_R is a read event, both having the same location and value.
- For every read event $E_R \in G$, there is a unique event $E_W \in G$ such that E_W rf E_R.
- If E_1 co E_2, then E_1 and E_2 are write events to the same location.
- For each location ℓ, co is a strict (irreflexive) total order on write events to that location.

In any candidate execution, we define the relation hb to be the smallest transitive relation including the following:

- E_1 hb E_2 whenever E_1 is an initial write and E_2 is not.
- E_1 hb E_2 whenever E_1 po E_2.
- E_1 hb E_2 whenever E_1 and E_2 access the same atomic location, and either E_1 co E_2 or E_1 rf E_2.

We also define fr (*from-reads*) so that E_1 fr E_2 if there exists an event E' such that E' rf E_1 and E' co E_2. Intuitively, E_1 fr E_2 if E_1 reads a value which is later overwritten by E_2. The relation $\mathsf{fr}_{\mathsf{at}}$ is fr restricted to atomic locations.

A *consistent execution* is a candidate execution satisfying:

Causality There are no cycles in $\mathsf{hb} \cup \mathsf{rf} \cup \mathsf{fr}_{\mathsf{at}}$

CoWW There are no E_1, E_2 such that E_1 hb E_2, E_2 co E_1
CoWR There are no E_1, E_2 such that E_1 hb E_2, E_2 fr E_1

6.1 Relating the Operational and Axiomatic Semantics

Next, we show that the operational semantics (traces) and axiomatic semantics (consistent executions) do in fact define the same memory model. First, we define the function $|-|$ which maps traces to event graphs:

$$|\emptyset| = G_0$$

$$|\underbrace{M_0 \cdots M_n}_{\Sigma} \xrightarrow{T_{n+1}} M_{n+1}| = |\Sigma|$$

if the transition T_{n+1} is Silent, and

$$|\underbrace{M_0 \cdots M_n}_{\Sigma} \xrightarrow{T_{n+1}} M_{n+1}| = |\Sigma| \cup \{((i, m), \ell, \phi\}$$

if the transition T_{n+1} uses Memory with ℓ, ϕ and Σ contains m prior memory operation steps on thread i.

If Σ is a trace, then the events of $|\Sigma|$ correspond to the memory operation steps (fig. 1c) of Σ. For an event $E \in |\Sigma|$ (other than an initial write), we write $T(E)$ for the corresponding transition in Σ. For two such events $E_1, E_2 \in |\Sigma|$, we write $E_1 <_\Sigma E_2$ if the transition $T(E_1)$ occurs before the transition $T(E_2)$ in the trace Σ. From any Σ, we construct a candidate execution $(|\Sigma|, \mathsf{po}_\Sigma, \mathsf{rf}_\Sigma, \mathsf{co}_\Sigma)$ as follows:

- po_Σ is the largest subset of $<_\Sigma$ relating only events on the same thread.
- $\mathsf{rf}_{\Sigma,\mathsf{A}}$ relates E_W to E_R whenever E_R is a read of A (Read-AT), and E_W is the most recent (by $<_\Sigma$) write to A (Write-AT), or IW_A if no such write exists.
- $\mathsf{rf}_{\Sigma,\mathsf{a}}$ relates E_W to E_R whenever E_R is a read of a (Read-NA), and E_W is the unique write to a (Write-NA) with the same timestamp, or IW_a if no such write exists.
- $\mathsf{rf}_\Sigma = \bigcup_\ell \mathsf{rf}_\ell$
- $\mathsf{co}_{\Sigma,\mathsf{A}}$ is the largest subset of $<_\Sigma$ relating only events writing to A.
- $\mathsf{co}_{\Sigma,\mathsf{a}}$ orders the write events to a by timestamp. Note that this might disagree with $<_\Sigma$.
- $\mathsf{co}_\Sigma = \bigcup_\ell \mathsf{co}_\ell$

The relationship between the operational and axiomatic semantics consists of a pair of theorems:

Theorem 15 (Soundness of axiomatic semantics)**.** *For all* Σ, $(|\Sigma|, \mathsf{po}_\Sigma, \mathsf{rf}_\Sigma, \mathsf{co}_\Sigma)$ *is a consistent execution.*

Theorem 16 (Completeness of axiomatic semantics)**.** *Every consistent* $(G, \mathsf{po}, \mathsf{rf}, \mathsf{co})$ *is* $(|\Sigma|, \mathsf{po}_\Sigma, \mathsf{rf}_\Sigma, \mathsf{co}_\Sigma)$ *for some* Σ.

7 Compilation

We now show that our memory model can be compiled efficiently to the x86 and ARMv8 memory models, and define the optimisations that are valid for a compiler to do.

This section will involve reasoning about relations between events, so we introduce some concise notation. We write $R_1; R_2$ for relational composition, so that $E\ (R_1; R_2)\ E'$ if there is some E'' such that $E\ R_1\ E''$ and $E''\ R_2\ E'$. We write R^{-1} for the transpose of R, so that $E\ R^{-1}\ E'$ if $E'\ R\ E$. We write 1 for the identity relation, and $R?$ for $R \cup 1$, and R^+ for the transitive closure of R. Note that $R_1?; R_2 = (R_1; R_2) \cup R_2$.

In our memory model, not all of the po relation is relevant, which is an important property for both compiler and hardware optimisations. For instance, there is no constraint that two nonatomic reads must be done in program order. To explain this formally, we define several subsets of po for the parts of program order that are relevant, and show that the memory model can be defined using only those parts. We define several subsets of po, relating events E_1 and E_2 when:

- po_{at-}: E_1 is an atomic read or write.
- po_{-at}: E_2 is an atomic write.
- po_{at-at}: E_1 is an atomic read or write and E_2 is an atomic write.
- po_{RW}: E_1 is a read and E_2 is a write (not necessarily to the same location).
- po_{con}: E_1 and E_2 access the same location, and at least one is a write.

Because hardware models treat communication within a single processor and communication between processors differently, the rf relation is split into *internal* and *external* parts:

$$\mathsf{rfi} = \mathsf{rf} \cap \mathsf{po} \qquad \mathsf{rfe} = \mathsf{rf} \backslash \mathsf{po}$$

and likewise for co and fr.

Informally, happens-before arises from initial writes, program order, or program order combined with several steps of communication via atomic operations. We define the relation hbinit to describe happens-before from initial writes, relating E_1 and E_2 when E_1 is an initial write and E_2 is not. We define the relation hbcom to describe the communication via atomic operations, as:

$$\mathsf{po}_{\mathsf{-at}}; ((\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at-at}})^*; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at-}}$$

These relations characterise the hb relation precisely:

Theorem 17. *For any candidate execution,* $\mathsf{hb} = \mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po}$.

To show the correctness of reordering and compilation, we use an alternative characterisation of consistent executions:

Theorem 18. *A candidate execution is consistent iff it satisfies the following conditions:*

Causality *There are no cycles in the following relation:*

$$\mathsf{hbcom} \cup \mathsf{po}_{at-} \cup \mathsf{po}_{-at} \cup \mathsf{po}_{RW} \cup \mathsf{rfe} \cup \mathsf{fre}_{at}$$

Coherence *The following relation is irreflexive:*

$$(\mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po}_{con}); (\mathsf{fr} \cup \mathsf{co})$$

7.1 Compiler Optimisations

In this section, we reason about the correctness of compiler optimisations in terms of valid reorderings and peephole optimisations. Theorem 18 characterises consistent executions, but refers only to certain subrelations of po, and never to the entire program order relation. Since theorem 18 refers only to the nonexistence of certain cycles, optimisations which introduce extra program order edges between events are permissible, since they cannot cause forbidden behaviours to become allowed. Therefore, any optimisation which does not shrink these subrelations of po is permissible, which characterises the constraints on optimisations:

- po_{at-}: Operations must not be moved before prior atomic operations.
- po_{-at}: Operations must not be moved after subsequent atomic writes.
- po_{RW}: Prior reads must not be moved after subsequent writes.
- po_{con}: Conflicting operations must not be reordered.

Furthermore, certain transformations involving adjacent operations on the same location are permissible. We reason about the correctness of peephole optimisations by arguing that the effect of the transformation is explained by our operational semantics (§3). In the following, `a`,`b`,`c` are distinct nonatomic locations, `r1`,`r2` are registers and `x`,`y` are values.

Sequentialisation: `[P || Q]` ⇒ `[P; Q]`. Since replacing a parallel computations with a sequential computation only adds edges to po, no forbidden cycle can become allowed. This seemingly-natural optimisation is in fact invalid under many memory models including C++ and Java [13, 23].

Redundant Load (RL): `[r1 = a; r2 = a]` ⇒ `[r1 = a; r2 = r1]`. By Read-NA, if the read of a yields x, then there is a write of value x at some timestamp t in a's history. The second read is allowed to read the same write.

Store Forwarding (SF): `[a = x; r1 = a]` ⇒ `[a = x; r1 = x]`. By Write-NA, the write of x to a is included in a's history. The subsequent read is allowed to read this write.

Dead Store (DS): `[a = x; a = y]` ⇒ `[a = y]`. By Write-NA, the first write is included in a's history. But the write only affects the current thread's frontier. By Read-NA, every other thread is allowed to see the prior write to this location; such a write always exists due to initial write on every location. Hence, no other thread is obligated to see the first write. Following the second write, write of y to a is included in the history and the current thread's frontier. Any subsequent reads of a in this thread must see the second write (Read-NA). Hence, no threads may witness the first write.

We can combine reordering and peephole optimisations to describe common compiler optimisations. Let po_{RR}, po_{WR} and po_{WW} be the program order relations between reads, write to reads and writes, respectively.

Table 1. Compilation to x86-TSO

Operation	Implementation
Nonatomic read	`mov R, [x]`
Nonatomic write	`mov [x], R`
Atomic read	`mov R, [x]`
Atomic write	`(lock) xchg R, [x]`[a]

[a] The `lock` prefix is implicit on the `xchg` instruction.

Common Subexpression Elimination: `[r1 = a*2; r2 = b; r3 = a*2]` $\xRightarrow{\mathit{reorder}}$ `[r1 = a*2; r3 = a*2; r2 = b]` $\xRightarrow{RL}$ `[r1 = a*2; r2 = r1; r3 = b]`, where the first step involves relaxing po_{RR}, which is permitted by the memory model.

Loop-Invariant Code Motion: `[while (...) { a = b; r1 = c*c; ...}]` $\xRightarrow{\mathit{reorder}}$ `[r2 = c*c; while (...) { a = b; r1 = r2; ...}]`, which involves relaxing po_{RR} and po_{WR}, both of which are permitted.

Dead Store Elimination: `[a = 1; b = c; a = 2]` $\xRightarrow{\mathit{reorder}}$ `[b = c; a = 1; a = 2]` $\xRightarrow{DS}$ `[b = c; a = 2]`, where the first step relaxes po_{WW} and po_{WR}, both of which are permitted.

Constant Propagation: `[a = 1; b = c; r = a]` $\xRightarrow{\mathit{reorder}}$ `[b = c; a = 1; r = a]` $\xRightarrow{SF}$ `[b = c; a = 1; r = 1]`, where the first step relaxes po_{WW} and po_{WR}, both of which are permitted.

Furthermore, if a program fragment satisfies the local DRF property, then the compiler is free to apply any optimisations valid for sequential programs, not just the ones permitted by the memory model, to that program fragment.

On the other hand, any compiler optimisation that breaks the load-to-store ordering is disallowed. For example, consider redundant store elimination optimisation: `[r1 = a; b = c; a = r1]` $\xRightarrow{\mathit{reorder}}$ `[r1 = a; a = r1; b = c]` ⇒ `[r1 = a; b = c]`. The first step relaxes po_{RW} which is disallowed.

7.2 Compilation to x86-TSO

The first compilation target is the x86-TSO model [22], for which we use the axiomatic presentation of Alglave et al. [6]. The compilation model is shown in table 1.

One wrinkle is that the hardware model for x86 has read-modify-write instructions (such as the `xchg` we are using for atomic stores), and our software model does not. Rather than adopting a new event type for RMW instructions, we adopt an encoding used by Wickerson et al. [24] and separate RMWs into a pair of a read and a write, with a marker indicating they are part of the same operation. Formally, we say that $(G, \mathrm{po}, \mathrm{rf}, \mathrm{co}, \mathrm{rmw})$ is an *x86-candidate execution* if $(G, \mathrm{po}, \mathrm{rf}, \mathrm{co})$ is a candidate execution, and rmw is a subset

Definitions of sets of events:

M = all events R = read events W = write events
WA = atomic write events (those with a rmw-predecessor)

Definitions of relations:

$$\mathrm{poloc} = \mathrm{po} \cap \{E_1, E_2 \mid E_1, E_2 \text{ access same location}\}$$
$$\mathrm{poghb} = \mathrm{po} \cap ((W \times W) \cup (R \times M))$$
$$\mathrm{implied} = \mathrm{po} \cap ((W \times WA) \cup (WA \times R))$$
$$\mathrm{ghb} = \mathrm{implied} \cup \mathrm{poghb} \cup \mathrm{rfe} \cup \mathrm{fr} \cup \mathrm{co}$$

Conditions:

$$\mathrm{acyclic}(\mathrm{poloc} \cup \mathrm{rf} \cup \mathrm{fr} \cup \mathrm{co})$$
$$\mathrm{acyclic}(\mathrm{ghb})$$
$$\mathrm{rmw} \cap (\mathrm{fre}; \mathrm{coe}) = \emptyset$$

Figure 3. Axiomatic model of x86-TSO

of po relating reads to writes, with no operations in program order between the read and the write.

An x86-candidate execution is *x86-consistent* if it satisfies the rules of fig. 3. In particular, the axiom rmw ∩ (fre; coe) ensures that RMW instructions such as `xchg` are atomic, by ensuring that there can be no intervening write between the read and the write part of the operation. Of course, there are many other features supported by the hardware not modelled by these rules: fences, non-temporal stores, self-modifying code, and so on. We use a simplified hardware model that does not include these, since they are not used by our compilation scheme.

Soundness of compilation means that every behaviour that the hardware model allows of a compiled program is allowed by the software model of the original program. Formally, we say that a candidate execution $(G, \mathrm{po}, \mathrm{rf}, \mathrm{co})$ is compiled to an x86-candidate execution $(G', \mathrm{po}', \mathrm{rf}', \mathrm{co}', \mathrm{rmw}')$ if there are functions ϕ from G to G' and ϕ_{WA} from the atomic writes of G to G' such that:

- $\phi(E)$ has the same action type (read or write) as E
- $\phi(E_1)$ po′ $\phi(E_2)$ iff E_1 po E_2
- $\phi(E_1)$ rf′ $\phi(E_2)$ iff E_1 rf E_2
- $\phi(E_1)$ co′ $\phi(E_2)$ iff E_1 co E_2
- $\phi_{WA}(E_W)$ rmw′ $\phi(E_W)$ for atomic writes E_W

This definition encodes the scheme of table 1, in particular by mapping atomic writes to read-modify-write instructions.

Theorem 19 (Soundness of compilation to x86)**.** *If* $(G, \mathrm{po}, \mathrm{rf}, \mathrm{co})$ *is compiled to* $(G', \mathrm{po}', \mathrm{rf}', \mathrm{co}', \mathrm{rmw}')$*, and the latter is an x86-consistent execution, then the former is a consistent execution.*

7.3 Compilation to ARMv8 (AArch64)

Compilation to the ARMv8 architecture is more subtle than to x86, due to the complexities introduced by the relaxed memory ordering of ARM processors [21]. The main issue is that the ARMv8 architecture admits *load-buffering*, allowing cycles in po ∪ rf. The classic example is as follows:

```
ldr R0, [x]  ║ ldr R0, [y]
mov R1,#1    ║ mov R1,#1
str R1,[y]   ║ str R1,[x]
```

Even though `x` and `y` are both initially zero, it is possible for both processors to end with `R0 = 1`, having read each other's writes, since the stores may be executed ahead of the loads.

As we saw in example 3, such behaviour is incompatible with local DRF as it causes data races in the future to affect computations now. Our compilation scheme must introduce enough dependencies that the processor is prevented from performing such reorderings.

There are several ways to accomplish this. Two simple ones are to insert a branch after loads, or to insert a `dmb ld` barrier before stores, shown in tables 2a and 2b respectively. We benchmark both of these approaches in §8, and find the overhead to be small.

The second unusual aspect of our compilation scheme is that we compile atomic stores as atomic exchanges, rather than simply using the `stlr` instruction directly. Our atomics make stronger guarantees than those provided by the `stlr` and `ldar` instructions on ARMv8. These instructions were designed to implement the C++ SC atomics, which have some curious behaviours with no simple operational explanation (see §9.2). In particular, ordering is preserved between nonatomic loads and subsequent atomic loads (for which we add `dmb ld` on atomic loads), between atomic stores and subsequent nonatomic stores (for which we add `dmb st` on atomic stores), and between atomic stores and subsequent nonatomic loads (for which we code atomic stores as exchanges). In each case, the full `dmb` barrier would suffice [20], but the versions chosen are cheaper.

Formally, an ARM-candidate execution is the same as an x86-candidate execution, except that events are annotated with whether they are atomic (`ldar`, `stlr`, `ldaxr`, `stlxr`) or not (`ldr`, `str`). A candidate execution $(G, \mathrm{po}, \mathrm{rf}, \mathrm{co})$ is compiled to an ARM-candidate execution $(G', \mathrm{po}', \mathrm{rf}', \mathrm{co}', \mathrm{rmw}')$ if there are functions ϕ from G to G' and ϕ_{WA} from the atomic writes of G to G' such that:

- $\phi(E)$ has the same action type (read or write) as E
- $\phi(E)$ is atomic iff E operates on an atomic location
- If E_1 po_{RW} E_2, then $\phi(E_1)$ (dep ∪ bob) $\phi(E_2)$
- $\phi(E_1)$ po′ $\phi(E_2)$ iff E_1 po E_2
- $\phi(E_1)$ rf′ $\phi(E_2)$ iff E_1 rf E_2
- $\phi(E_1)$ co′ $\phi(E_2)$ iff E_1 co E_2
- $\phi_{WA}(E_W)$ rmw′ $\phi(E_W)$ for atomic writes E_W

The definition is similar to the case for x86, with the additional complication of needing to preserve read-to-write ordering, by ensuring that $\mathrm{po}_{RW} \subseteq \mathrm{obs}$ (by inserting branches, `dmb ld`, or another means).

An ARM-candidate execution is *ARM-consistent* if it satisfies the rules of fig. 4, which is an abridged version of the

Table 2. Compilation to ARMv8 (AArch64)

Operation	Implementation
Nonatomic read	`ldr R, [x]; cbz R, L; L:`
Nonatomic write	`str R, [x]`
Atomic read	`dmb ld; ldar R, [x]`
Atomic write	`L: ldaxr; stlxr; cbnz L; dmb st`

(a) Compilation scheme 1

Operation	Implementation
Nonatomic read	`ldr R, [x]`
Nonatomic write	`dmb ld; str R, [x]`
Atomic read	`dmb ld; ldar R, [x]`
Atomic write	`L: ldaxr; stlxr; cbnz L; dmb st`

(b) Compilation scheme 2

multi-copy atomic ARMv8 specification. Ignoring the [. . .] markers, these rules define a weaker model than the full specification, by omitting several ordering guarantees made by the architecture. This simplified model is enough to establish soundness for our compilation scheme, and readers interested in the full model are referred to Pulte et al. [20].

The soundness theorem parallels that for x86:

Theorem 20 (Soundness of compilation to ARMv8)**.** *If* $(G, \mathrm{po}, \mathrm{rf}, \mathrm{co})$ *is compiled to* $(G', \mathrm{po}', \mathrm{rf}', \mathrm{co}', \mathrm{rmw}')$*, and the latter is an ARM-consistent execution, then the former is a consistent execution.*

8 Performance Evaluation

We now quantify the performance impact of our memory model by evaluating a large suite of *sequential* OCaml benchmarks. Each compiler variant we consider is the stock OCaml compiler (trunk snapshot as of 2017-09-18) with patches to emit the necessary instruction sequences for enforcing our memory model on the target architecture. We focus on quantifying the cost of nonatomic accesses on ARM and POWER architectures, since nonatomics in our memory model are free on x86. We leave the evaluation of the performance of our atomics for future work.

Our aim is to evaluate the suitability of the memory model to be the default one in Multicore OCaml [12], a parallel extension of OCaml with thread-local minor heaps and shared major heap. The compiler variants used in our evaluation are stock OCaml with memory model patches but not the parallel runtime, so that we can quantify the performance impact of the memory model in isolation.

The ARM machine (`AArch64`) is a 2 socket, 96-core 2.5GHz Cavium ThunderX 64-bit ARMv8 server with 32Kb of L1 data cache, 78Kb of L1 instruction cache, and 16Mb shared L2 cache. The POWER machine (`PowerPC`) is a 2-core 3425 MHz IBM pSeries virtualised server with 64Kb L1 data cache and 32K L1 instruction cache.

Definitions of sets of events:

M = all events
R = reads, `ldr`, `ldar` or `ldaxr`
W = writes, `str`, `stlr` or `stlxr`
Acq = load-acquires, `ldar` or `ldaxr`
Rel = store-releases, `stlr` or `stlxr`

Definitions of relations:

ctrl = events in program order,
separated by a branch dependent on the first
dmbld = events in program order, separated by `dmb ld`
dmbst = events in program order, separated by `dmb st`

$$\mathrm{obs} = \mathrm{rfe} \cup \mathrm{fre} \cup \mathrm{coe}$$
$$\mathrm{dob} = \mathrm{addr} \cup (\mathrm{ctrl} \cap (M \times W)) \cup [\ldots]$$
$$\mathrm{aob} = \mathrm{rmw} \cup [\ldots]$$
$$\begin{aligned}\mathrm{bob} = {} & (\mathrm{po} \cap (\mathrm{Acq} \times M)) \cup (\mathrm{po} \cap (M \times \mathrm{Rel})) \\ & \cup (\mathrm{dmbld} \cap (R \times M)) \cup (\mathrm{dmbst} \cap (W \times W)) \\ & \cup (\mathrm{po} \cap (\mathrm{Rel} \times \mathrm{Acq})) \cup [\ldots]\end{aligned}$$
$$\mathrm{ob} = \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{aob} \cup \mathrm{bob}$$

Conditions:

$$\mathrm{acyclic}(\mathrm{poloc} \cup \mathrm{rf} \cup \mathrm{fr} \cup \mathrm{co})$$
$$\mathrm{acyclic}(\mathrm{ob})$$
$$\mathrm{rmw} \cap (\mathrm{fre}; \mathrm{coe}) = \emptyset$$

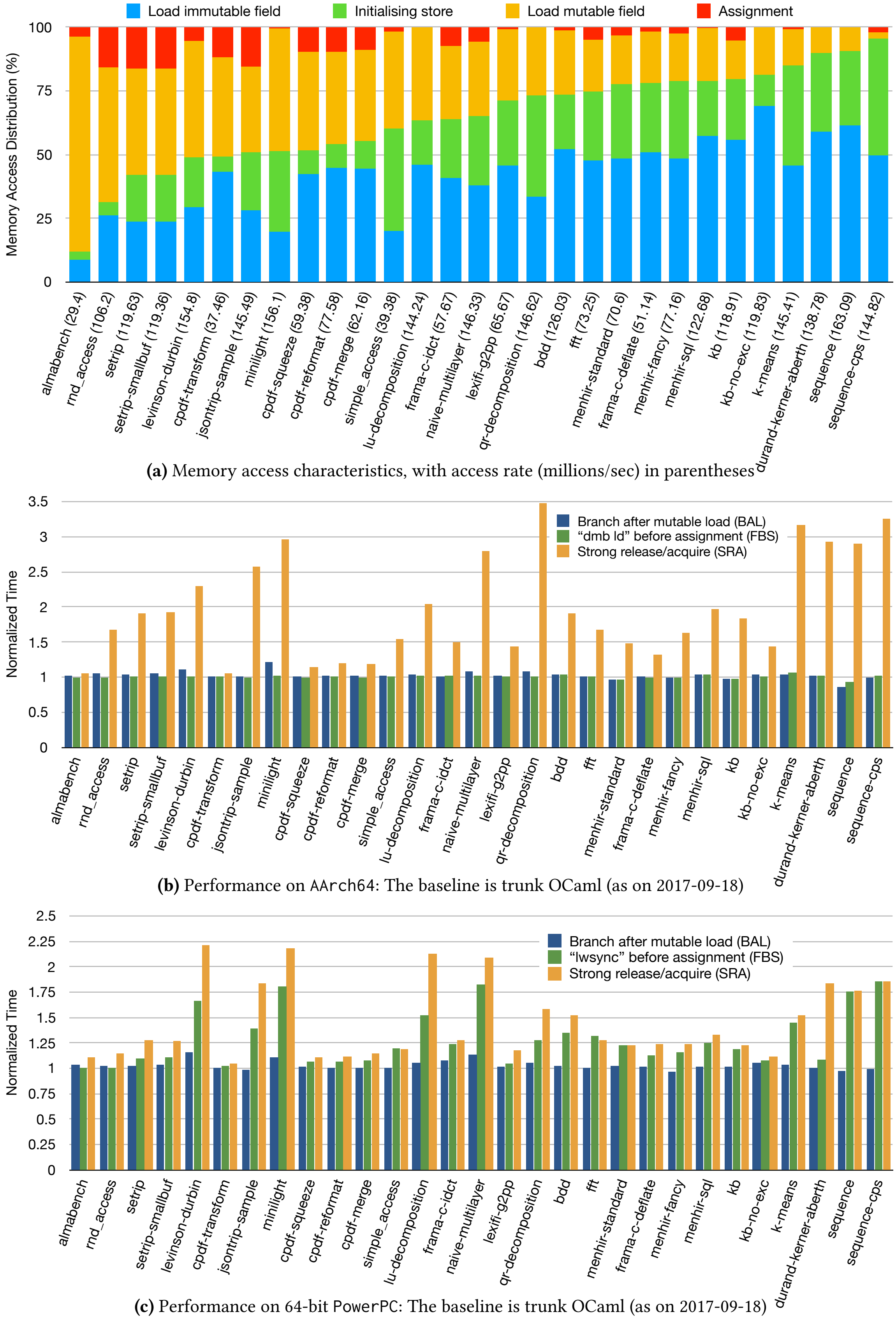
Figure 4. Axiomatic model of ARMv8 (abridged)

The OCaml benchmarks include a mix of workloads including parsers (`menhir`, `jsontrip`, `setrip`), utilities (`cpdf`), static analysis (`frama-c`) and numerical benchmarks (`lexifi-g2gpp`, `k-means`, `minilight`, `almabench`, etc.). The benchmarks were run one after the other, with the other available cores running `rnd_access` to simulate a loaded machine. Figure 5a shows the memory access distribution of the benchmarks.

8.1 Initialising Stores and Immutable Loads

The Multicore OCaml heap layout with thread-local minor heaps and a shared major heap offers the opportunity to optimise initialising stores and loads from immutable fields. New objects in Multicore OCaml are allocated in a thread-local minor heap with large objects allocated directly in the major heap. Such objects are initialised with initialising stores before the program gets a reference to those objects. Hence, a thread will see its own initialising stores to the objects it allocated. However, in a multi-threaded setting, we need to explicitly ensure that a thread does witness the initialising stores from a different thread.

In Multicore OCaml, objects in the thread-local minor heap may become shared when a thread explicitly *promotes* the object to the shared heap in response to a request from a different thread. Thread-local objects may also become



(a) Memory access characteristics, with access rate (millions/sec) in parentheses

(b) Performance on AArch64: The baseline is trunk OCaml (as on 2017-09-18)

(c) Performance on 64-bit PowerPC: The baseline is trunk OCaml (as on 2017-09-18)

Figure 5. Performance results

shared after they are promoted to the major heap at the end of a minor collection followed by assigning the object to a shared variable. We issue a full fence (`dmb ish` on `AArch64`) and (`sync` on `PowerPC`) at the end of a promotion and minor GC, which ensures the visibility of initialising stores by a different thread. We also issue a full fence after initialising large objects in the major heap and after initialising globals at the start of the program. We leave the proof of correctness of initialising stores for future work. As a result, initialising stores in OCaml are practically free in our memory model.

Multicore OCaml statically distinguishes mutable from immutable fields. Immutable fields have initialising stores but have no further assignments. Due to our compilation of initialising stores, loads from immutable fields can be compiled as plain loads. The benchmarks in fig. 5a are arranged in the order of increasing *functionalness*: a program that performs fewer loads of mutable fields and assignments is said to be more functional than a program which does more.

8.2 Assignments and Mutable Loads

We are now left only with imperative operations: assignments to and loads from mutable fields. As we saw in §7.3, we can enforce our memory model on `AArch64` by either compiling loads of mutable fields as a dependent branch after load (BAL, as per table 2a), `[r <- ldr; cbz r, L; L:]`, or a fence before store (FBS, as per table 2b) that orders prior loads before the store (`[dmb ld; str]`). On `PowerPC`, the equivalent instruction sequences are `[r <- ld; cmpi r, 0; beq L; L:]` and `[lwsync; st]`[2].

For the sake of comparison, we also include strong release/acquire (SRA) [14], which is strictly stronger than the compilation models presented above. We enforce SRA by compiling all mutable loads as load acquire (`[ldar]` on `AArch64` and `[r <- ld; cmpi r, 0; beq L; L: isync]` on `PowerPC`) and assignments are store release (`[stlr]` on `AArch64` and `[lwsync; st]` on `PowerPC`).

8.3 Results

We compare the performance of the different compilation schemes against vanilla OCaml (snapshot on 2017-09-18), which compiles loads and stores without any decorations. The results are presented in fig. 5b and fig. 5c. The results show that on average, BAL, FBS and SRA are 2.5%, 0.6% and 85.3% slower than the baseline on `AArch64` and 2.9%, 26.0% and 40.8% slower on `PowerPC`. The low overheads for BAL and FBS on `AArch64` illustrate that the memory model is suitable for compiling Multicore OCaml while permitting modular reasoning in the presence of races.

Recall from §7.3 that our goal is to prevent load-buffering behaviours (i.e po_{RW} reorderings). BAL precisely avoids this reordering permitting the processor to reorder other operations, and is the optimal compilation scheme in terms of reorderings allowed. `dmb ld` orders prior reads before subsequent operations avoiding RR and RW reorderings. However, FBS for `AArch64` only inserts the fence before stores, and allows all RR reorderings. Compared to `dmb ld`, `lwsync` is stronger since it also avoids WW reordering in addition to RR and RW reorderings. Hence, the performance impact of FBS is greater on `PowerPC`.

SRA is slower on `AArch64` due to our compilation model for floating-point loads and stores. `AArch64` does not have the floating-point equivalent of `stlr` and `ldar` instructions. Hence, we compile floating point loads and stores as `dmb ld` after and `dmb st` before the operations, correspondingly. This is the reason for marked slowdown of SRA compiled numerical benchmarks on `AArch64`. On `PowerPC`, enforcing SRA for floating-point memory accesses are no worse than integer accesses modulo the floating-point comparison for loads. Hence, the performance impact is moderated. However, the performance impact is still high compared to the optimal compilation scheme, BAL.

The results show that some of the benchmarks such as BAL and FBS versions of sequence and menhir-standard on `AArch64` run faster than the baseline version. We hypothesised that this was due to instruction cache effects. We tested this hypothesis by padding loads and stores with `nop` instructions in the baseline compiler to match the instruction length in BAL and FBS, which did indeed produce the same performance improvement as the BAL and FBS cases.

9 Related Work and Discussion

9.1 Load-buffering and Out-of-thin-air

Our model prohibits loads from being reordered after later stores, meaning that it is impossible for the following example to yield `a = 1`, `b = 1` (all variables start as 0):

```
x = a; || a = b;
b = 1; ||
```

However, weakly-ordered architectures (e.g. POWER and ARM) allow this outcome, which is why our compilation scheme requires introduction of dependencies. They do not, however, allow `a = 1`, `b = 1` in the following example, as it would involve constructing a value "out of thin air":

```
if (a == 1) || a = b;
  b = 1;    ||
```

If all current compiler and hardware optimisations are to be preserved, then it is necessary to distinguish between these two classes of load-buffering behaviours. This has proven remarkably difficult for software models, due particularly to the difficulty of defining a notion of "dependence" which survives compiler optimisations [7].

[2] `lwsync` is not a precise equivalent of ARM's `dmb ld`, but both have the effect of preserving load-to-store ordering.

We do not attempt to make this distinction, since as example 3 showed, even load-to-store reordering without dependence breaks local DRF. Our model simply bans all load-buffering behaviour instead. This straightforward approach to side-stepping out-of-thin-air has been proposed before, by Boehm and Demsky [10]. It was not adopted for C++ (which currently allows even the out-of-thin-air behaviour above) on performance grounds. At least in our setting of OCaml, the performance impact of this approach is low (§8).

9.2 Comparison to Other Memory Models

C++ Due to the "catch-fire" semantics of data races in C++, a direct comparison with our model is not particularly enlightening. However, it is instructive to compare against C++, if we replace all nonatomic accesses with `relaxed` atomics, and use SC atomics for atomic accesses, which ensures races have well-defined behaviour. Apart from load-buffering (see above), there are two major differences.

The first is that our model has weaker coherence than that provided by C++ relaxed atomics. C++ ensures that if a relaxed atomic write by another thread has been observed by this thread, subsequent reads of the same variable will also observe the write. This stronger coherence property is provided by most hardware models, but invalidates common optimisations such as CSE by requiring compiler to treat reads as side-effecting operations [19].

The second major difference is that our atomic writes have stronger semantics, which is why we use atomic exchanges instead of `stlr` on ARMv8. Consider the following, using an atomic (SC atomic) location `A` and a nonatomic (relaxed) location `b`:

```
x = b; ‖ A = 2;
A = 1; ‖ b = 1;
```

In our model, if `A = 2` afterwards, then `x = 0`. This is clear from the operational semantics: the step `x = b` must precede `A = 1`, which must precede `A = 2` and `b = 1`. However, in C++ the outcome `A = 2` and `x = 1` is possible. In C++, SC atomic events are totally ordered, so `x = b` must happen-before `A = 1`, which must precede in the SC ordering `A = 2`, which must happen-before `b = 1`. However, these two orderings do not compose, and in particular `x = b` does not happen-before `b = 1`, and it is permissible for `x = b` to read-from `b = 1`.

This behaviour cannot be explained operationally without either allowing reads to read from future writes, or allowing atomic locations to contain multiple or incoherent values, so it is not permitted in our simple operational model. However, this means that we must choose an alternative compilation scheme on ARMv8 and similar architectures.

Java Since the primitives memory operations provided by Java (nonatomic fields and `volatile` fields with sequentially consistent semantics) match ours, a direct comparison is possible. The most notable differences are again, the lack of load buffering in our model, as well as the *lack* of coherence properties in Java [16]. It is this lack of coherence which causes the effect of example 2, in which data races occurring in the past do not resolve to a single value.

Promising Semantics The semantics of Kang et al. [13] accounts for a large fragment of the C++ memory model, including release-acquire, relaxed and nonatomic accesses, while introducing a novel "promise" mechanism to give an operational interpretation to load-buffering behaviours. Importantly, the semantics is well-defined even in the presence of data races. In fact, our operational semantics (§3) is a simplified version of this semantics (omitting promises).

We suspect that a weaker version of local DRF holds for this model, which defends the programmer against data races in the past or on other variables. Data races in the future (example 3) still appear, since the purpose of the promising mechanism is explicitly to permit load-buffering.

Strong Release-Acquire The SRA model of Lahav, Giannarakis and Vafeiadis [14] enforces release-acquire semantics on all accesses. This is a strong memory model, and has an operational semantics based on message-passing. We conjecture that the local DRF property holds in their model. Unfortunately, the strength of release-acquire accesses make them efficiently implementable only on strongly-ordered machines like x86 (see §8.3). In an appendix, Lahav et al. sketch an extension of their model with nonatomic locations, although these use catch-fire semantics for races.

Sequential Consistency SC is certainly a well-behaved memory model, and trivially has the local DRF property. Sadly, its implementation on commodity architectures is expensive, requiring even more fences than SRA. Marino et al. argue that SC can be made affordable by cooperation between an SC-preserving optimising compiler and a hardware extension for detecting SC violations [17, 18].

Other Languages Relaxed memory models are currently being proposed for other high-level languages (JavaScript [2], WebAssembly [4], Go [1], Rust [5], Swift [3]), but only have the complex C++ or Java memory models as guidelines. We believe that our memory model could serve as a reasonable template for such high-level languages. In particular, adapting C++ memory model for a safe language proves to be especially difficult as it is not immediately obvious how to exclude out-of-thin-air behaviours without paying the cost of expensive barriers and excluding compiler optimisations [8]. Our model prohibits out-of-thin-air behaviours at a small sequential performance cost.

10 Conclusions and Future Work

Our memory model is simpler than prevalent mainstream models and enjoys strong reasoning principles even in the presence of data races, while remaining efficiently implementable even on relaxed architectures. We intend this model to be the basis of the multicore implementation for the OCaml programming language, and hope it is of interest to the designers of other safe languages.

In future work, we plan to extend our currently spartan model with other types of atomics. In particular, release-acquire atomics would be a useful extension: they are strong enough to describe many parallel programming idioms, yet weak enough to be relatively cheaply implementable. Two routes to this suggest themselves: by extending our operational model with release-acquire primitives in the style of Kang et al. [13], or by extending the SRA model of Lahav et al. [14] with load-buffering-free nonatomics.

Acknowledgments

We thank John Wickerson, Mark Batty, Scott Owens, François Pottier, Luc Maranget, Gabriel Scherer, Aleksey Shipilev and Doug Lea for their useful comments on earlier versions of the memory model, as well as Peter Sewell, Shaked Flur, and the Cambridge Relaxed Memory Concurrency group for help understanding the ARM hardware models, and Suresh Jagannathan, Gowtham Kaki, Jens Palsberg, and the anonymous reviewers for their feedback on earlier drafts of this paper. Portions of this research were funded via a Royal Commission for the Exhibition of 1851 research fellowship, and by Jane Street and VMWare. We also thank IBM and Packet.net for their generous hosting of PowerPC and ARM hardware.

References

[1] 2014. The Go Memory Model. (2014). https://golang.org/ref/mem

[2] 2016. ECMAScript Sharedmem: Formal memory model proposal tracking. (2016). https://github.com/tc39/ecmascript_sharedmem/issues/133

[3] 2017. Concurrency in Swift. (2017). https://github.com/apple/swift/blob/master/docs/proposals/Concurrency.rst

[4] 2017. WebAssembly Threads. (2017). https://github.com/WebAssembly/design/issues/1073

[5] 2018. Rust Atomics. (2018). https://doc.rust-lang.org/beta/nomicon/atomics.html

[6] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV’10)*. Springer-Verlag, 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

[7] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. *The Problem of Programming Language Concurrency Semantics*. Springer Berlin Heidelberg, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

[8] Mark Batty and Peter Sewell. 2014. The Thin-air Problem. (2014). https://www.cl.cam.ac.uk/~pes20/cpp/notes42.html

[9] Hans-J. Boehm. 2011. How to Miscompile Programs with “Benign” Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism (HotPar’11)*. USENIX Association. http://dl.acm.org/citation.cfm?id=2001252.2001255

[10] Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC ’14)*. ACM, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

[11] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. *Bounding Data Races in Space and Time (extended version)*. Technical Report. University of Cambridge, Computer Laboratory.

[12] Stephen Dolan, Leo White, and Anil Madhavapeddy. 2014. Multicore OCaml. OCaml Workshop. (2014).

[13] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proc. POPL ’17*. ACM, 175–189. https://doi.org/10.1145/3009837.3009850

[14] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-acquire Consistency. *SIGPLAN Not.* 51, 1 (Jan. 2016), 649–662. https://doi.org/10.1145/2914770.2837643

[15] Chris Lattner. 2012. Random LLVM Notes. (2012). http://www.nondot.org/sabre/LLVMNotes/MemoryUseMarkers.txt

[16] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proc. POPL ’05*. ACM, 378–391. https://doi.org/10.1145/1040305.1040336

[17] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *Proc. PLDI ’10*. ACM, 351–362. https://doi.org/10.1145/1806596.1806636

[18] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-preserving Compiler. *SIGPLAN Not.* 46, 6 (June 2011), 199–210. https://doi.org/10.1145/1993316.1993522

[19] William Pugh. 1999. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande (JAVA ’99)*. ACM, 89–98. https://doi.org/10.1145/304065.304106

[20] Chirstopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proc. POPL ’18*.

[21] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proc. PLDI ’11*. ACM, 175–186. https://doi.org/10.1145/1993498.1993520

[22] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[23] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Proc. POPL ’15*. ACM, 209–220. https://doi.org/10.1145/2676726.2676995

[24] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proc. POPL ’17*. 190–204. http://dl.acm.org/citation.cfm?id=3009838

A Proof of Local DRF

Given a transition T of some trace, write $F(T)$ for the frontier of the thread performing T before the operation, and $F'(T)$ for the frontier afterwards. We order frontiers pointwise.

Lemma 21. *For all transitions T, $F(T) \leq F'(T)$.*

Proof. Case analysis. $F'(T)$ is always one of $F(T)$, $F[\mathsf{a} \mapsto t]$ for some $t > F(\mathsf{a})$ or $F_{\mathsf{A}} \sqcup F(T)$, all of which are $\geq F(T)$. □

Lemma 22. *Given a trace*

$$M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} M_n$$

then if T_i happens-before T_j, $F'(T_i) \leq F'(T_j)$.

Proof. Induction on the happens-before relation, giving two cases:

- T_i and T_j are on the same thread. By lemma 21, frontiers grow monotonically within a thread.
- T_i is a write and T_j is a read or write to the same atomic location. T_i modifies the location's frontier to $F'(T_i)$, and by inspecting fig. 1c, we see that the frontiers associated with an atomic location grow monotonically, so $F'(T_j)$ contains the location's frontier, which is above $F'(T_i)$.

□

There is a sort of converse of this lemma for nonatomic writes:

Lemma 23. *For any nonatomic location a and trace*

$$M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} M_n$$

then if $t \in \mathrm{dom}\ F(T_j)(\mathsf{a})$ where $t > 0$, then there is some transition T_i which is a write to the same location a at timestamp t, such that T_i happens-before T_j.

Proof. By induction on j. Let T_k $(k < j)$ be the transition on the same thread prior to T_j, so that $F'(T_k) = F(T_j)$ and T_k happens-before T_j. Since $t \in \mathrm{dom}\ F'(T_k)(\mathsf{a})$, either

- T_k is a write at timestamp t, in which case we choose $i = k$
- $t \in \mathrm{dom}\ F(T_k)(\mathsf{a})$, in which case we use the inductive hypothesis.
- T_k is an atomic operation. Then let T_m be the previous atomic write to the same location, so that $t \in \mathrm{dom}\ F(T_m)(\mathsf{a})$ and the inductive hypothesis applies.

□

As a technicality, we must argue that the semantics is never obliged to perform weak transitions:

Lemma 24. *If $M \xrightarrow{T_1} M_1$ for some weak transition T, then $M \xrightarrow{T_2} M_2$ for some non-weak T_2 of the same type (read or write) accessing the same location, and some machine state M_2.*

Proof. If T_1 is a write, pick T_2 by choosing a sufficiently large timestamp. If T_2 is a read, choose a different action that reads the value with the largest timestamp, and choose a different expression transition by proposition 4. □

Proof of theorem 13

Proof. Consider an arbitrary trace including M:

$$M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} M \xrightarrow{T'_1} M'_1 \xrightarrow{T'_2} \ldots \xrightarrow{T'_n} M'_n \xrightarrow{T'} M'$$

in which the transitions T'_i are assumed to be L-sequential (and the others are not). We must show that either T' is L-sequential, or else that there exists some $M'_n \xrightarrow{T''} M''$ with a data race between some T'_i and T''.

Suppose T' is not L-sequential, and is therefore a weak transition acting on a nonatomic location a in L. Let H be the history of a in M'_n. Since T' is weak, H contains a timestamp t such that $F'(T')(\mathsf{a}) < t$. Let T'' be the non-weak transition given by lemma 24 which accesses the same location as T'.

Timestamp t was added to H by some transition T, which must have been a write action to location a. If T happens-before T', then lemma 22 implies $t = F'(T)(\mathsf{a}) \leq F'(T')(\mathsf{a})$, a contradiction.

So, T does not happen-before T' but conflicts with T', so there is a data race between T and T'. L-stability of M implies that T cannot be T_i for any i, so it must be some T'_i, making T'' a non-weak transition with a data race on $\mathsf{a} \in L$. □

B Proof of equivalence between operational and axiomatic models

Given a trace Σ, we write hb_Σ and fr_Σ for the relations defined on the candidate execution $(|\Sigma|, \mathsf{po}_\Sigma, \mathsf{rf}_\Sigma, \mathsf{co}_\Sigma)$.

Lemma 25. *Excluding initial writes, the relations $\mathsf{rf}_\Sigma, \mathsf{po}_\Sigma$ and $\mathsf{co}_{\Sigma,\mathsf{A}}$, $\mathsf{fr}_{\Sigma,\mathsf{A}}$ for atomic locations A are subsets of $<_\Sigma$.*

Proof. Immediate from definition for all but $\mathsf{fr}_{\Sigma,\mathsf{A}}$. Suppose $E_1\ \mathsf{fr}_{\Sigma,\mathsf{A}}\ E_2$, so there is some write E' where $E'\ \mathsf{rf}_{\Sigma,\mathsf{A}}\ E_1$, $E'\ \mathsf{co}_{\Sigma,\mathsf{A}}\ E_2$. If $E_2 <_\Sigma E_1$, then E' would not be the latest write before E_1, contradicting the definition of $\mathsf{rf}_{\Sigma,\mathsf{A}}$, so $E_1 <_\Sigma E_2$. □

The operational semantics defines a happens-before relation (definition 8), which coincides with the axiomatic semantics' version:

Lemma 26. *For all non-initial $E_1, E_2 \in |\Sigma|$, $E_1\ \mathsf{hb}_\Sigma\ E_2$ iff $T(E_1)$ happens-before $T(E_2)$.*

Proof. Both definitions agree on program order, but happens-before refers to the order of atomic operations in the trace while while hb_Σ uses the relations rf_Σ and co_Σ. For an atomic location A, relations $\mathsf{rf}_{\Sigma,\mathsf{A}}$, $\mathsf{co}_{\Sigma,\mathsf{A}}$ and $\mathsf{fr}_{\Sigma,\mathsf{A}}$ relate writes to later operations on the same atomic location, so hb implies happens-before. Conversely, if E_1 is a write and E_2 another operation to the same atomic location, where $E_1 <_\Sigma E_2$, then we must show that $E_1\ \mathsf{hb}_\Sigma\ E_2$, which we do by case analysis on the type of E_2:

- E_2 is a write, so E_1 co_Σ E_2 since $E_1 <_\Sigma E_2$.
- E_2 is a read. Let E_W be the unique write such that E_W rf_Σ E_2, which by definition is the last write before E_2 (ordered by $<_\Sigma$). Therefore, either $E_W = E_1$ so E_1 rf_Σ E_2, or E_1 co_Σ E_W rf E_2.

□

Proof of theorem 15

We must show that the candidate execution $(|\Sigma|, \mathsf{po}_\Sigma, \mathsf{rf}_\Sigma, \mathsf{co}_\Sigma)$ satisfies Causality, CoWW and CoWR.

Causality When E_W is an initial write, there are no events E' hb E_W nor E' rf_Σ E_W, and there are no E' fr_Σ E_W since this would imply some other write E'_W being a co-predecessor of an initial write. Therefore, initial writes cannot appear in cycles of $\mathsf{hb}_\Sigma \cup \mathsf{rf}_\Sigma \cup \mathsf{fr}_{\Sigma,\mathsf{at}}$.

For events other than initial writes, po_Σ, rf_Σ, $\mathsf{co}_{\Sigma,\mathsf{A}}$ and $\mathsf{fr}_{\Sigma,\mathsf{A}}$ (for A atomic location) are all subsets of $<_\Sigma$ (lemma 25), so a cycle in $\mathsf{hb}_\Sigma \cup \mathsf{rf}_\Sigma \cup \mathsf{fr}_{\Sigma,\mathsf{at}}$ implies one in $<_\Sigma$.

CoWW Since E_2 co_Σ E_1, both are writes to some location. We may assume they are writes to a nonatomic location a, since $\mathsf{co}_{\Sigma,\mathsf{A}} \subseteq \mathsf{hb}_\Sigma$ and hb_Σ has no cycles by Causality. Let t_1, t_2 be the timestamps of $T(E_1), T(E_2)$. By definition of WRITE-NA, $F'(T(E_1)) = t_1$ and $F'(T(E_2)) = t_2$. By definition of co_Σ, $t_2 < t_1$. Since E_1 hb_Σ E_2, then by lemmas 22 and 26, $t_1 = F'(T(E_1)) \leq F'(T(E_2)) = t_2 < t_1$, a contradiction.

CoWR As before, we may assume that E_2 is a read and E_1 is a write to a nonatomic location a. Let E' be the write such that E' rf_Σ E_2, E' co_Σ E_1. E_1 is not initial since it has a co_Σ-predecessor, so let t_1 be its timestamp. Let t' be the timestamp of E' (taken to be 0 if E' is initial), and note that by definition of READ-NA, $F'(T(E_2)) \leq t'$, and by definition of co_Σ, $t' < t_1$. Since E_1 hb_Σ E_2, then by lemmas 22 and 26, $t_1 = F'(T(E_1)) \leq F'(T(E_2)) \leq t' < t_1$, a contradiction.

Given a consistent execution $(G, \mathsf{po}, \mathsf{rf}, \mathsf{co})$, define its *restriction* to a subset $G' \subseteq G$ to be $(G', \mathsf{po}_{G'}, \mathsf{rf}_{G'}, \mathsf{co}_{G'})$, where the relation $R_{G'}$ restricts R to relate only elements of G'.

Lemma 27. *If $(G, \mathsf{po}, \mathsf{rf}, \mathsf{co})$ is a consistent execution, and $E \in G$ is an event such that there is no E' where E rf E', then the restriction of the execution to $G\backslash\{E\}$ is a consistent execution.*

Proof. The restriction is a candidate execution because the conditions on po and co are true in subsets, and the conditions on rf cannot be violated if there are no events that read from E. The restriction is consistent because the axioms Causality, CoWW and CoWR specify the nonexistence of certain cycles in relations, so cannot be made false by removing events. □

Proof of theorem 16

We proceed by induction on the number of events (other than initial writes) in G, noting $|\emptyset| = G_0$ for the base case.

Otherwise, G has $n+1$ events other than initial writes. By Causality, the transitive closure of $\mathsf{hb} \cup \mathsf{rf} \cup \mathsf{fr}_{\mathsf{at}}$ has no cycles and is therefore a strict partial order, and by finiteness and nonemptiness of G it has an element E which is maximal, in that there is no E' such that E hb E' or E rf E' or E $\mathsf{fr}_{\mathsf{at}}$ E'.

Let $G' = G\backslash\{E\}$. By lemma 27, the restriction of $(G, \mathsf{po}, \mathsf{rf}, \mathsf{co})$ to G' is a consistent execution of size n, so by the inductive hypothesis there is a trace Σ' such that $|\Sigma'| = G'$ and $(G', \mathsf{po}_{\Sigma'}, \mathsf{rf}_{\Sigma'}, \mathsf{co}_{\Sigma'})$ is consistent.

E is not an initial write, so let $((i, n), \ell, \phi) = E$. E has no po-successors (since $\mathsf{po} \subseteq \mathsf{hb}$), so n is the number of po-prior events on thread i, so we can extend the trace Σ' with a transition corresponding to E (possibly preceded by some silent steps), if we can construct a memory operation for E. We do so by case analysis on whether ℓ is atomic, and on ϕ:

- ℓ atomic, ϕ = write x: Rule WRITE-AT always applies.
- ℓ atomic, ϕ = read x: Let $E_W \in G'$ be the unique event such that E_W rf E. In order to apply READ-AT, we must ensure that there is no intervening write $E'_W \in G'$ such that $E_W <_{\Sigma'} E'_W$. By totality of co, E_W co E'_W or E'_W co E_W. If E_W co E'_W, then E $\mathsf{fr}_{\mathsf{at}}$ E'_W, contradicting the fact that E has no $\mathsf{fr}_{\mathsf{at}}$-successors. If E'_W co E_W, then E'_W hb E_W, so $E'_W <_{\Sigma'} E_W$, a contradiction.
- ℓ nonatomic, ϕ = write x: Choose a timestamp t greater than that of any co_Σ-predecessors of Σ and smaller than that of any co_Σ-successors, which always exists since timestamps are unique and the rationals are dense. In order to apply WRITE-NA, we need that $F(T(E))(\ell) < t$. Suppose instead that $F(T(E))(\ell) > t$ (it cannot be equal, since writes choose distinct timestamps). Then by lemma 23 there is some write event E_W with timestamp $t' > t$ which happens-before E. But then E co_Σ E_W and E_W hb_Σ E (by lemma 26), a violation of CoWW.
- ℓ nonatomic, ϕ = read x: Let $E_W \in G'$ be the unique event such that E_W rf E, and let t be the timestamp of E_W (0 if E_W is initial). In order to apply READ-NA, we must ensure that $F(T(E))(\ell) \leq t$. Suppose instead that $F(T(E))(\ell) > t$. Then by lemma 23 there is some other write event E'_W with timestamp $t' > t$ which happens-before E. But then E fr_Σ E'_W and E'_W hb_Σ E (by lemma 26), a violation of CoWR.

C Proofs about compilation

Proof of theorem 17

Proof. In concise notation, hb is:

$$\mathsf{hb} = (\mathsf{hbinit} \cup \mathsf{po} \cup \mathsf{rf}_{\mathsf{at}} \cup \mathsf{co}_{\mathsf{at}})^+$$

This includes hbinit ∪ hbcom ∪ po. The difficult case is to show the converse, which we do by showing (i), that:

$$\mathsf{hbinit} \cup \mathsf{po} \cup \mathsf{rf}_{\mathsf{at}} \cup \mathsf{co}_{\mathsf{at}} \subseteq \mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po}$$

and (ii), that hbinit ∪ hbcom ∪ po is transitive.

(i) First, note that since $\mathsf{rfi}_{\mathsf{at}} \subseteq \mathsf{po}$, $\mathsf{po} \cup \mathsf{rf}_{\mathsf{at}} = \mathsf{po} \cup \mathsf{rfe}_{\mathsf{at}}$, and likewise for co, so it suffices to prove:

$$\mathsf{hbinit} \cup \mathsf{po} \cup \mathsf{rfe}_{\mathsf{at}} \cup \mathsf{coe}_{\mathsf{at}} \subseteq \mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po}$$

This holds since $\mathsf{rfe}_{\mathsf{at}} \cup \mathsf{coe}_{\mathsf{at}} \subseteq \mathsf{hbcom}$.

(ii) We must show

$$(\mathsf{hbinit}{\cup}\mathsf{hbcom}{\cup}\mathsf{po}); (\mathsf{hbinit}{\cup}\mathsf{hbcom}{\cup}\mathsf{po}) \subseteq \mathsf{hbinit}{\cup}\mathsf{hbcom}{\cup}\mathsf{po}$$

The relations hbinit, hbcom and po do not relate initial events on the right, so $(\mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po}); \mathsf{hbinit} = \emptyset$, and $\mathsf{hbinit}; (\mathsf{hbcom} \cup \mathsf{po}) \subseteq \mathsf{hbinit}$. So, it suffices to show:

$$(\mathsf{hbcom} \cup \mathsf{po}); (\mathsf{hbcom} \cup \mathsf{po}) \subseteq \mathsf{hbcom} \cup \mathsf{po}$$

By definition of hbcom and transitivity of po, we have $\mathsf{po}; \mathsf{hbcom} \subseteq \mathsf{hbcom}$ and $\mathsf{hbcom}; \mathsf{po} \subseteq \mathsf{hbcom}$, so it suffices that:

$$\mathsf{hbcom}; \mathsf{hbcom} \subseteq \mathsf{hbcom}$$

Expanding the definition of hbcom,

$$\begin{aligned} & \mathsf{hbcom}; \mathsf{hbcom} \subseteq \\ & \mathsf{po}_{-\mathsf{at}}; ((\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-\mathsf{at}})^*; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-}; \\ & \mathsf{po}_{-\mathsf{at}}; ((\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-\mathsf{at}})^*; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-} \subseteq \\ & \mathsf{po}_{-\mathsf{at}}; ((\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-\mathsf{at}})^*; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-\mathsf{at}}; \\ & ((\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-\mathsf{at}})^*; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-} \subseteq \\ & \mathsf{po}_{-\mathsf{at}}; ((\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-\mathsf{at}})^*; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-} = \\ & \mathsf{hbcom} \end{aligned}$$

□

Proof of theorem 18

The original Causality axiom is:

$$\mathsf{acyclic}(\mathsf{hb} \cup \mathsf{rf} \cup \mathsf{fr}_{\mathsf{at}})$$

Expanding hb by theorem 17, this is:

$$\mathsf{acyclic}(\mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po} \cup \mathsf{rf} \cup \mathsf{fr}_{\mathsf{at}})$$

Since $\mathsf{rfi}, \mathsf{fri} \subseteq \mathsf{po}$, this is:

$$\mathsf{acyclic}(\mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po} \cup \mathsf{rfe} \cup \mathsf{fre}_{\mathsf{at}})$$

An initial write has no predecessors by hbinit, hbcom, po, rfe, or fre, so edges of hbinit cannot be part of a Causality-breaking cycle, making this equivalent to:

$$\mathsf{acyclic}(\mathsf{hbcom} \cup \mathsf{po} \cup \mathsf{rfe} \cup \mathsf{fre}_{\mathsf{at}})$$

Consider a pair of events E_1 po E_2 in a cycle of these relations. We assume that the next and previous elements in the cycle are on different threads (extending the po-segment by transitivity, if necessary). All of the relations other than rfe use atomic events, so either one of E_1 or E_2 is atomic, or else we have E_W rfe E_1 po E_2 rfe E_R, in which case E_1 is a read and E_2 is a write, making Causality equivalent to:

$$\mathsf{hbcom} \cup \mathsf{po}_{\mathsf{at}-} \cup \mathsf{po}_{-\mathsf{at}} \cup \mathsf{po}_{RW} \cup \mathsf{rfe} \cup \mathsf{fre}_{\mathsf{at}}$$

The original Coherence axioms (CoWW and CoWR), combined, are:

$$\mathsf{irreflexive}(hb; (\mathsf{fr} \cup \mathsf{co}))$$

By theorem 17, this is equivalent to:

$$\mathsf{irreflexive}((\mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po}); (\mathsf{fr} \cup \mathsf{co})$$

But if $E_1 (\mathsf{fr} \cup \mathsf{co}) E_2$, then E_1 and E_2 access the same location and at least one is a write, so this is equivalent to:

$$\mathsf{irreflexive}((\mathsf{hbinit} \cup \mathsf{hbcom} \cup \mathsf{po}_{con}); (\mathsf{fr} \cup \mathsf{co})$$

Both proofs of soundness of the compilation scheme (to x86 and to ARM) are done in the same style, by showing that cycles forbidden in the software model by the Causality and Coherence rules are part of cycles forbidden in the machine model.

To remove some clutter, we leave the map ϕ from the candidate execution to the machine candidate execution implicit in the proofs below, allowing us to conflate po, rf and co with their counterparts in the machine model, since the hypotheses of theorems 19 and 20 ensure they must agree.

Proof of theorem 19 (Compilation to x86)

Causality for x86

Proof. Expanding the definition of hb, the Causality axiom states:

$$\mathsf{acyclic}(\mathsf{hbinit} \cup \mathsf{po} \cup \mathsf{co}_{\mathsf{at}} \cup \mathsf{fr}_{\mathsf{at}} \cup \mathsf{rf})$$

or equivalently,

$$\mathsf{acyclic}(\mathsf{hbinit} \cup \mathsf{po} \cup \mathsf{coe}_{\mathsf{at}} \cup \mathsf{fre}_{\mathsf{at}} \cup \mathsf{rfe})$$

since $\mathsf{coi}, \mathsf{fri}, \mathsf{rfi} \subseteq \mathsf{po}$. As before, hbinit cannot take part in such a cycle, so this is equivalently:

$$\mathsf{acyclic}(\mathsf{po} \cup \mathsf{coe}_{\mathsf{at}} \cup \mathsf{fre}_{\mathsf{at}} \cup \mathsf{rfe})$$

We show that there are no such cycles by showing that any cycle would induce a cycle in ghb, which is forbidden in the x86 model (fig. 3). $\mathsf{co}_{\mathsf{at}} \cup \mathsf{fr}_{\mathsf{at}} \cup \mathsf{rf}$ is included in ghb, so the difficult case is po.

Consider some E_1 po E_2, part of such a cycle. Since po is transitive, we may assume that this edge is preceded and followed by a non-po edge.

Since E_1 is preceded by $\mathsf{coe}_{\mathsf{at}}$, $\mathsf{fre}_{\mathsf{at}}$ or rfe, it must be an atomic write (WA), or a possibly-nonatomic read (R). If it is an atomic write, then E_1 ($\mathsf{poghb} \cup \mathsf{implied}$) E_2, while if it is a read, then E_1 poghb E_2. Either way, E_1 ghb E_2. □

Coherence for x86

We use the formulation of Coherence from theorem 18. Splitting it into three cases, we must show that these three relations are irreflexive:

$$\mathsf{hbinit}; (\mathsf{fr} \cup \mathsf{co})$$
$$\mathsf{hbcom}; (\mathsf{fr} \cup \mathsf{co})$$
$$\mathsf{po}_{\mathsf{con}}; (\mathsf{fr} \cup \mathsf{co})$$

The first is easy since initial writes cannot have co-predecessors. To prove the second, we first note that $\mathsf{po}_{\mathsf{at}-} \subseteq \mathsf{ghb}$ and $\mathsf{po}_{-\mathsf{at}} \subseteq \mathsf{ghb}$, so $\mathsf{hbcom} \subseteq \mathsf{ghb}$ and so if there were E ($\mathsf{hbcom}; (\mathsf{fr} \cup \mathsf{co})$) E, then ghb would have a cycle. The third is easy since cycles in poloc ∪ rf ∪ fr ∪ co are not allowed, and $\mathsf{po}_{\mathsf{con}} \subseteq$ poloc.

Proof of theorem 20 (Compilation to ARMv8) Causality for ARMv8

Proof. Expanding the definition of hb, the Causality axiom states:

$$\mathrm{acyclic}(\mathsf{hbinit} \cup \mathsf{po} \cup \mathsf{co}_{\mathsf{at}} \cup \mathsf{fr}_{\mathsf{at}} \cup \mathsf{rf})$$

or equivalently,

$$\mathrm{acyclic}(\mathsf{hbinit} \cup \mathsf{po} \cup \mathsf{coe}_{\mathsf{at}} \cup \mathsf{fre}_{\mathsf{at}} \cup \mathsf{rfe})$$

since coi, fri, rfi ⊆ po. As before, hbinit cannot take part in such a cycle, so this is equivalently:

$$\mathrm{acyclic}(\mathsf{po} \cup \mathsf{coe}_{\mathsf{at}} \cup \mathsf{fre}_{\mathsf{at}} \cup \mathsf{rfe})$$

We show that there are no such cycles by showing that any cycle would induce a cycle in ob, which is forbidden in the ARMv8 model (fig. 4). $\mathsf{co}_{\mathsf{at}} \cup \mathsf{fr}_{\mathsf{at}} \cup \mathsf{rf}$ is included in ob (via obs), so the difficult case is po.

Consider some E_1 po E_2, part of such a cycle. Since po is transitive, we may assume that this edge is preceded and followed by a non-po edge.

Since E_1 is preceded by $\mathsf{coe}_{\mathsf{at}}$, $\mathsf{fre}_{\mathsf{at}}$ or rfe, it must be an atomic write (Rel), or a possibly-nonatomic read (R). Since E_2 is followed by $\mathsf{coe}_{\mathsf{at}}$, $\mathsf{fre}_{\mathsf{at}}$ or rfe, it must be an atomic write (Rel), an atomic read (Acq) or a possibly-nonatomic write. This gives six cases:

- $E_1 \in \mathsf{Rel}, E_2 \in \mathsf{Rel}$: $(E_1, E_2) \in \mathsf{po} \cap (\mathsf{M} \times \mathsf{Rel}) \subseteq \mathsf{ob}$
- $E_1 \in \mathsf{Rel}, E_2 \in \mathsf{Acq}$: $(E_1, E_2) \in \mathsf{po} \cap (\mathsf{Rel} \times \mathsf{Acq}) \subseteq \mathsf{ob}$
- $E_1 \in \mathsf{Rel}, E_2 \in \mathsf{W}$: $(E_1, E_2) \in \mathsf{dmbst} \cap (W \times W) \subseteq \mathsf{ob}$
- $E_1 \in \mathsf{R}, E_2 \in \mathsf{Rel}$: $(E_1, E_2) \in \mathsf{po} \cap (\mathsf{M} \times \mathsf{Rel}) \subseteq \mathsf{ob}$
- $E_1 \in \mathsf{R}, E_2 \in \mathsf{Acq}$: $(E_1, E_2) \in \mathsf{dmbld} \cap (R \times M) \subseteq \mathsf{ob}$
- $E_1 \in \mathsf{R}, E_2 \in \mathsf{W}$: $(E_1, E_2) \in \mathsf{dep} \cup \mathsf{bob} \subseteq \mathsf{ob}$

□

Coherence for ARMv8

Proof. We use the formulation of Coherence from theorem 18. Splitting it into three cases, we must show that these three relations are irreflexive:

$$\mathsf{hbinit}; (\mathsf{fr} \cup \mathsf{co})$$
$$\mathsf{hbcom}; (\mathsf{fr} \cup \mathsf{co})$$
$$\mathsf{po}_{\mathsf{con}}; (\mathsf{fr} \cup \mathsf{co})$$

The first is easy since initial writes cannot have co-predecessors, and the third is easy since cycles in poloc ∪ rf ∪ fr ∪ co are not allowed, and $\mathsf{po}_{\mathsf{con}} \subseteq$ poloc. The second case is the hard case.

We show that $\mathsf{hbcom}; (\mathsf{fr} \cup \mathsf{co}) \subseteq \mathsf{ob}^+$, which the ARMv8 model states cannot have cycles. By definition, hbcom is

$$\mathsf{po}_{-\mathsf{at}}; ((\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-\mathsf{at}})^*; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-}$$

Since $\mathsf{po}_{-\mathsf{at}} \subseteq \mathsf{po} \cap (M \times \mathsf{Rel}) \subseteq \mathsf{bob} \subseteq \mathsf{ob}$, we have $\mathsf{hbcom} \subseteq \mathsf{ob}^+; (\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}})$, so it suffices to show that:

$$(\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-}; (\mathsf{fr} \cup \mathsf{co}) \subseteq \mathsf{ob}^+$$

Since $\mathsf{fr} \cup \mathsf{co} \subseteq \mathsf{fre} \cup \mathsf{coe} \cup \mathsf{po} \subseteq \mathsf{ob} \cup \mathsf{po}$, it is sufficient to show that:

$$(\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-}; (\mathsf{ob} \cup \mathsf{po}) \subseteq \mathsf{ob}^+$$

Since $\mathsf{po}_{\mathsf{at}-}; \mathsf{po} \subseteq \mathsf{po}_{\mathsf{at}-}$, it also suffices that:

$$(\mathsf{coe}_{\mathsf{at}} \cup \mathsf{rfe}_{\mathsf{at}}); \mathsf{po}_{\mathsf{at}-} \subseteq \mathsf{ob}^+$$

If E_1 $\mathsf{po}_{\mathsf{at}-}$ E_2, then E_1 is either an atomic write or an atomic read. Only writes have coe-predecessors, and only reads have rfe-predecessors, so our goal splits into two cases:

$$\mathsf{coe}_{\mathsf{at}}; (\mathsf{po} \cap (\mathsf{Rel} \times M)) \subseteq \mathsf{ob}^+$$
$$\mathsf{rfe}_{\mathsf{at}}; (\mathsf{po} \cap (\mathsf{Acq} \times M)) \subseteq \mathsf{ob}^+$$

The second case is easy, since $(\mathsf{po} \cap (\mathsf{Acq} \times M)) \subseteq \mathsf{ob}$. In the first case, consider E_1 $\mathsf{coe}_{\mathsf{at}}$ E_2. Let E_W be the latest co-predecessor of E_2 which is not on the same thread as E_2. (In other words, E_W is the write just before the chain of co-edges enters E_2's thread). We have E_1 co? E_W coe E_2. Also, E_1 ob? E_W: trivially, if $E_1 = E_W$, by coe ⊆ ob if E_1 and E_W are on different threads, or by $\mathsf{po} \cap (M \times \mathsf{Rel}) \subseteq \mathsf{ob}$ if they are on the same thread.

Let E'_W be the atomic write immediately after E_W in coherence order, which is on the same thread as E_2 (and might well be the same event), and let E'_R be its associated `ldaxr` event, immediately preceding E'_W, so that E'_R rmw E'_W.

Since E'_R and E'_W are a read-modify-write pair, and E_W is the immediate co-predecessor of E'_W, E_W rfe E'_R. Let E_3 now be some event in program order after E_2, and therefore after E'_R in program order as well. We now have:

$$E_1 \ \mathsf{ob}? \ E_W \ \mathsf{rfe} \ E'_R \ (\mathsf{po} \cap (\mathsf{Acq} \times M)) \ E_3$$

so E_1 ob^+ E_3.

□

D Example 2 in Java

The Java example below exhibits a failure to bound data races in time (Example 2). Occasionally, this program prints `b = 2, c = 1`, although it depends on the precise compiler options in use. We reproduced the example using `javac Example2.java && java -XX:TieredStopAtLevel=1 Example2` on both Java 8 (`1.8.0_162`) and Java 9 (`9.0.1`).

```
import java.util.concurrent.CyclicBarrier;
public class Example2 {
    static class A { int x; }
    A a, a2;
    volatile boolean flag;

    void reset() {
        a = a2 = new A();
        flag = false;
    }

    void p() {
        a.x = 1;
        flag = true;
    }

    void q() {
        A a = this.a, a2 = this.a2;

        a.x = 2;
        boolean f = flag;
        int b = a.x;
        int c = a2.x;

        if (f && b != c)
            System.out.println("b = " + b + ", c = " + c);
    }

    public static void main(String[] args) throws Exception {
        final Example2 ex = new Example2();
        final CyclicBarrier barrier = new CyclicBarrier(2, () -> { ex.reset(); });
        final int ITER = 100000000;
        Thread t = new Thread() {
            public void run() {
                for (int i=0; i<ITER; i++) {
                    try { barrier.await(); }
                    catch (Exception e) { throw new RuntimeException(e); }
                    ex.p();
                }
            }
        };
        t.start();
        for (int i=0; i<ITER; i++) {
            barrier.await();
            ex.q();
        }
        t.join();
    }
}
```