

Bounding Data Races in Space and Time

(Extended version, with appendices)

Stephen Dolan
University of Cambridge, UK
stephen.dolan@cl.cam.ac.uk

KC Sivaramakrishnan
University of Cambridge, UK
sk826@cl.cam.ac.uk

Anil Madhavapeddy
University of Cambridge, UK
anil.madhavapeddy@cl.cam.ac.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.

Keywords weak memory models, operational semantics

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 (§5).
- show that our model supports many common compiler optimisations and provide sound compilation schemes to both the x86 and ARMv8 architectures (§6), and demonstrate their efficiency in practice in the hybrid functional-imperative language OCaml (§7).

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*

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.

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. Consider the following fragment of C++, with nonatomic global variables a , b , c :

Example 1.

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

||

```
b = 1;
```

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;
b = t;
... // some computation
c = t;
```

Register pressure in the elided computation can cause the temporary `t` to be spilled. Since its value is already stored in location `b`, a clever register allocator may choose to re-materialise `t` from `b` instead of allocating a new stack slot¹, giving:

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

However, in the transformed program, the data race between `b = a + 10` and `b = 1` will 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, in the original program, the only data race is the two concurrent writes to `b`, 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 [3] 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 [9] 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. Consider this example. Here, `a`, `b` and `c` are nonatomic variables (initially 0), and `flag` is a `volatile` (atomic) variable (initially `false`):

Example 2.

```
a = 1;
flag = true;
|||
a = 2;
if (flag) {
    b = a;
    c = a;
}
```

¹This 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, one of the final stages in the compiler pipeline. However, Lattner[8] describes this as a desirable improvement to LLVM’s register allocator.

Here, there is a race between the two writes to `a`, although not between reads and writes: if the reads of `a` occur, then both writes must happen before both reads, since `volatile` variables induce synchronisation in Java. During the body of the `if` there are no concurrent writes to `a`, so we might imagine that after this program finishes, `b = c`: either the reads do not occur, and both `b` and `c` remain 0, or the reads do occur and both read the same value.

However, Java permits the outcome `b = 1, c = 2`. 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.²

Surprisingly, non-sequential behaviour can also occur because of data races in the future, as in the following example:

```
class C { int x; }
int a;

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

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.

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

Example 3.

```
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 §7, and revisit the topic in §8.1.

2.3 Local DRF

So, 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:

²Appendix D shows a `jcstress` test case which exhibits this kind of behaviour on Java 8

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 and prove local DRF for our model in §4, after introducing the operational semantics in §3. Here, we give an informal example of reasoning with the local DRF property. The following program is the standard message-passing idiom, using a message stored in a nonatomic variable m and a flag stored in an atomic variable F . Unlike the standard idiom, this example also includes an irrelevant data-race on a third location a .

$$\begin{array}{l} m = 42; \\ a = 1; \\ F = 1; \end{array} \parallel \begin{array}{l} a = 2; \\ \text{if } (F) \{ \\ \quad x = m; \\ \} \end{array}$$

Our operational semantics is a small-step transition relation between machine states, and some of the transitions that it may perform do not correspond to steps a sequentially consistent machine may take. We term these *weak transitions*.

Local DRF lets us choose an arbitrary collection of locations L to analyse, which in this example will be $\{m\}$. We must start in a state in which there are no ongoing data races on any location in L , either because there were never any such races or they have already ended (as in example 2). We make this idea precise in §4.

The theorem tells us that from this state, there will not be a weak transition involving a location in L , until that location encounters a data race. In our example, since there are no data races on m (due to synchronisation via F), we know that m will have sequential behaviour, so the read of m must return 42, despite the data race on a .

3 A simple operational model

Memory consists of *locations* $\ell \in \mathcal{L}$, divided into *atomic locations* A, B, \dots and *nonatomic locations* a, b, \dots , 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. [6], 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

$$C; F \xrightarrow{\ell:\phi} C'; F'$$

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: it is not required that the timestamp be later than everything else in the history, merely that it be later than any other write the writing thread knows about.

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:

$$e \xrightarrow{\epsilon} e' \quad e \xrightarrow{\ell:\phi} e'$$

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 $e \xrightarrow{\ell:\text{read } x} e'$, then for every $y, e \xrightarrow{\ell:\text{read } y} e_y$ for some e_y .*

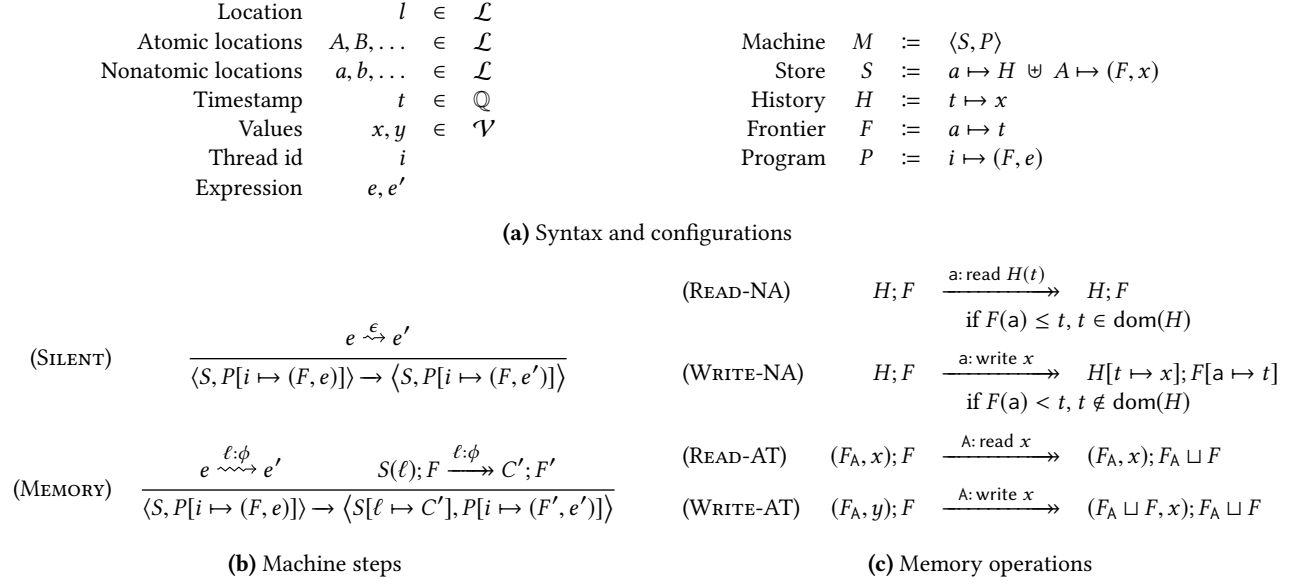


Figure 1. Operational semantics

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 $e \xrightarrow{\epsilon} e'$, then its new state is (F, e') (rule SILENT). Otherwise, if it can step by $e \xrightarrow{\ell: \phi} e'$, 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:

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

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*

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

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

This model is quite close to a sequential model of memory. The only situations in which it differs from sequential consistency is during transitions READ-AT and WRITE-AT. 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:

- $H; F \xrightarrow{a: \text{read } x} H; F$ 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.
- $H; F \xrightarrow{a: \text{write } x} H[t \mapsto x]; F'$ when t is not greater than the largest timestamp $t' \in H$. Informally, this write is not the latest write in 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.*

4.1 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 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 .

4.2 The local DRF theorem

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.

$$\begin{aligned} \text{(SILENT-G)} \quad & \frac{e \xrightarrow{\varepsilon} e'}{\langle G, P[i \mapsto (n, e)] \rangle \rightarrow \langle G, P[i \mapsto (n, e')] \rangle} \\ \text{(MEMORY-G)} \quad & \frac{e \xrightarrow{\ell, \phi} e'}{\langle G, P[i \mapsto (n, e)] \rangle \rightarrow \langle G \cup \{(i, n, \ell, \phi)\}, P[i \mapsto (n+1, e')] \rangle} \end{aligned}$$

Figure 2. Generating events from programs

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 L -sequential transition $M_n \xrightarrow{T'} M'$ with a data race between some T_i and T'

The usual DRF-SC theorem follows as a simple application of local DRF: take L to be the entire set of nonatomic locations, take M to be the initial state M_0 (which is vacuously L -stable). Then the local DRF theorem states that either all possible traces are sequential, or there is some sequential trace leading to a data race.

5 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 6.2 and 6.3), characterise allowed instruction reorderings by compiler optimisations (§6.1), and for comparison with other memory models (§8.2).

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 $\phi = \text{read } x$, and similarly for *write events*.

The event identifiers k are of one of two forms: either a pair (i, n) , indicating the n th 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.

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 :

$$G_0 = \{(IW_\ell, \ell, v_0) \mid \ell \in \mathcal{L}\}$$

The event graphs G generated by the rules of fig. 2 include all possible executions of the program, as well as many non-sensical 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 $E_1 co E_2$. Intuitively, $E_1 fr E_2$ if E_1 reads a value which is later overwritten by E_2 . The relation fr_{at} is fr restricted to atomic locations.

A *consistent execution* is a candidate execution satisfying:

- Causality** There are no cycles in $hb \cup rf \cup fr_{at}$
- 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 $E_1 hb E_2, E_2 fr E_1$

5.1 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 $(|\Sigma|, po_{\Sigma}, rf_{\Sigma}, co_{\Sigma})$ as follows:

- po_{Σ} is the largest subset of $<_{\Sigma}$ relating only events on the same thread.
- $rf_{\Sigma, A}$ 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.
- $rf_{\Sigma, a}$ 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.
- $rf_{\Sigma} = \bigcup_{\ell} rf_{\ell}$
- $co_{\Sigma, A}$ is the largest subset of $<_{\Sigma}$ relating only events writing to A .
- $co_{\Sigma, a}$ orders the write events to a by timestamp. Note that this might disagree with $<_{\Sigma}$.
- $co_{\Sigma} = \bigcup_{\ell} co_{\ell}$

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

Theorem 14 (Soundness of axiomatic semantics). *For all Σ , $(|\Sigma|, po_{\Sigma}, rf_{\Sigma}, co_{\Sigma})$ is a consistent execution.*

Theorem 15 (Completeness of axiomatic semantics). *Every consistent (G, po, rf, co) is $(|\Sigma|, po_{\Sigma}, rf_{\Sigma}, co_{\Sigma})$ for some Σ .*

6 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:

$$rf_i = rf \cap po \quad rfe = rf \setminus po$$

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 hb_{init} 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 hb_{com} to describe the communication via atomic operations, as:

$$po_{at-}; ((coe_{at} \cup rfe_{at}); po_{at-at})^*; (coe_{at} \cup rfe_{at}); po_{at-}$$

These relations characterise the hb relation precisely:

Theorem 16. *For any candidate execution, $hb = hb_{init} \cup hb_{com} \cup po$.*

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

Theorem 17. *A candidate execution is consistent iff it satisfies the following conditions:*

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

$$hb_{com} \cup po_{at-} \cup po_{-at} \cup po_{RW} \cup rfe \cup fre_{at}$$

Coherence *The following relation is irreflexive:*

$$(hb_{init} \cup hb_{com} \cup po_{con}); (fr \cup co)$$

6.1 Compiler optimisations

In this section, we reason about the correctness of compiler optimisations in terms of valid reorderings and peephole optimisations. Theorem 17 characterises consistent executions, but refers only to certain subrelations of po , and never to the entire program order relation. Therefore, any reordering which preserves 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 are nonatomic locations, $r1, r2$ are registers and x, y are values.

Redundant load (RL): $[r1 = a; r2 = a] \Rightarrow [r1 = a; r2 = r1]$. 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): $[a = x; r1 = a] \Rightarrow [a = x; r1 = x]$. 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): $[a = x; a = y] \Rightarrow [a = y]$. 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.

Common subexpression elimination: $[r1 = a*2; r2 = b; r3 = a*2] \xrightarrow{\text{reorder}} [r1 = a*2; r3 = a*2; r2 = b] \xrightarrow{RL} [r1 = a*2; r2 = r1; r3 = b]$, where the first step involves relaxing po_{RR} , which is permitted by the memory model.

Loop-invariant code motion: $[\text{while } (\dots) \{ a = b; r1 = c*c; \dots \}] \xrightarrow{\text{reorder}} [r2 = c*c; \text{while } (\dots) \{ a = b; r1 = r2; \dots \}]$, which involves relaxing po_{RR} and po_{WR} , both of which are permitted.

Dead store elimination: $[a = 1; b = c; a = 2] \xrightarrow{\text{reorder}} [b = c; a = 1; a = 2] \xrightarrow{DS} [b = c; a = 2]$, where the first step relaxes po_{WW} and po_{WR} , both of which are permitted.

Constant propagation: $[a = 1; b = c; r = a] \xrightarrow{\text{reorder}} [b = c; a = 1; r = a] \xrightarrow{SF} [b = c; a = 1; r = 1]$, 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.

6.2 Compilation to x86-TSO

The first compilation target is the x86-TSO memory model [15], for which we use the axiomatic presentation of Alglave et al. [1]. The compilation model is shown in table 3.

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

^aThe lock prefix is implicit on the xchg instruction.

Table 3. Compilation to x86-TSO

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:

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

Conditions:

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

Figure 4. Axiomatic model of x86-TSO

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. [16] 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, po, rf, co, rmw) is an *x86-candidate execution* if (G, po, rf, co) is a candidate execution, and rmw is a subset 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. 4. In particular, the axiom $rmw \cap (fre; coe) = \emptyset$ 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, po, rf, co) is compiled to an x86-candidate execution $(G', po', rf', co', rmw')$ 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 3, in particular by mapping atomic writes to read-modify-write instructions.

Theorem 18 (Soundness of compilation to x86). *If (G, po, rf, co) is compiled to $(G', po', rf', co', rmw')$, and the latter is an x86-consistent execution, then the former is a consistent execution.*

6.3 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 [14]. The main issue is that the ARMv8 architecture admits *load-buffering*, allowing cycles in $po \cup rf$. 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 5a and 5b respectively. We benchmark both of these approaches in §7, 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. The ARMv8 `stlr` instruction is designed to implement C++ SC atomics, and inherits some curious behaviours which have no simple operational explanation. Compiling our atomics to `stlr` would not be sound, but using an atomic exchange instead restores soundness. We revisit this point in §8.2.

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, po, rf, co) is compiled to an ARM-candidate execution $(G', po', rf', co', rmw')$ 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

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	

Table 5. Compilation to ARMv8 (AArch64)

- If $E_1 \text{ po}_{RW} E_2$, then $\phi(E_1) (\text{dep} \cup \text{bob}) \phi(E_2)$
- $\phi(E_1) \text{ po}' \phi(E_2)$ iff $E_1 \text{ po} E_2$
- $\phi(E_1) \text{ rf}' \phi(E_2)$ iff $E_1 \text{ rf} E_2$
- $\phi(E_1) \text{ co}' \phi(E_2)$ iff $E_1 \text{ co} E_2$
- $\phi_{WA}(E_W) \text{ rmw}' \phi(E_W)$ 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 $\text{po}_{RW} \subseteq \text{obs}$ (by inserting branches, `dmb ld`, or another means).

An ARM-candidate execution is *ARM-consistent* if it satisfies the rules of fig. 6. The rules in fig. 6 are an abridged version of the multi-copy atomic ARMv8 specification. Ignoring the [...] markers, these rules define a somewhat 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. [13].

The soundness theorem parallels that for x86:

Theorem 19 (Soundness of compilation to ARMv8).

If $(G, \text{po}, \text{rf}, \text{co})$ is compiled to $(G', \text{po}', \text{rf}', \text{co}', \text{rmw}')$, and the latter is an ARM-consistent execution, then the former is a consistent execution.

7 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

Definitions of sets of events:

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

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`
 $\text{obs} = \text{rfe} \cup \text{fre} \cup \text{coe}$
 $\text{dob} = \text{addr} \cup (\text{ctrl} \cap (M \times W)) \cup [\dots]$
 $\text{aob} = \text{rmw} \cup [\dots]$
 $\text{bob} = (\text{po} \cap (\text{Acq} \times M)) \cup (\text{po} \cap (M \times \text{Rel}))$
 $\cup (\text{dmbld} \cap (R \times M)) \cup (\text{dmbst} \cap (W \times W))$
 $\cup (\text{po} \cap (\text{Rel} \times \text{Acq})) \cup [\dots]$
 $\text{ob} = \text{obs} \cup \text{dob} \cup \text{aob} \cup \text{bob}$

Conditions:

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

Figure 6. Axiomatic model of ARMv8 (abridged)

free on x86. We leave the evaluation of the performance of our atomics for future work.

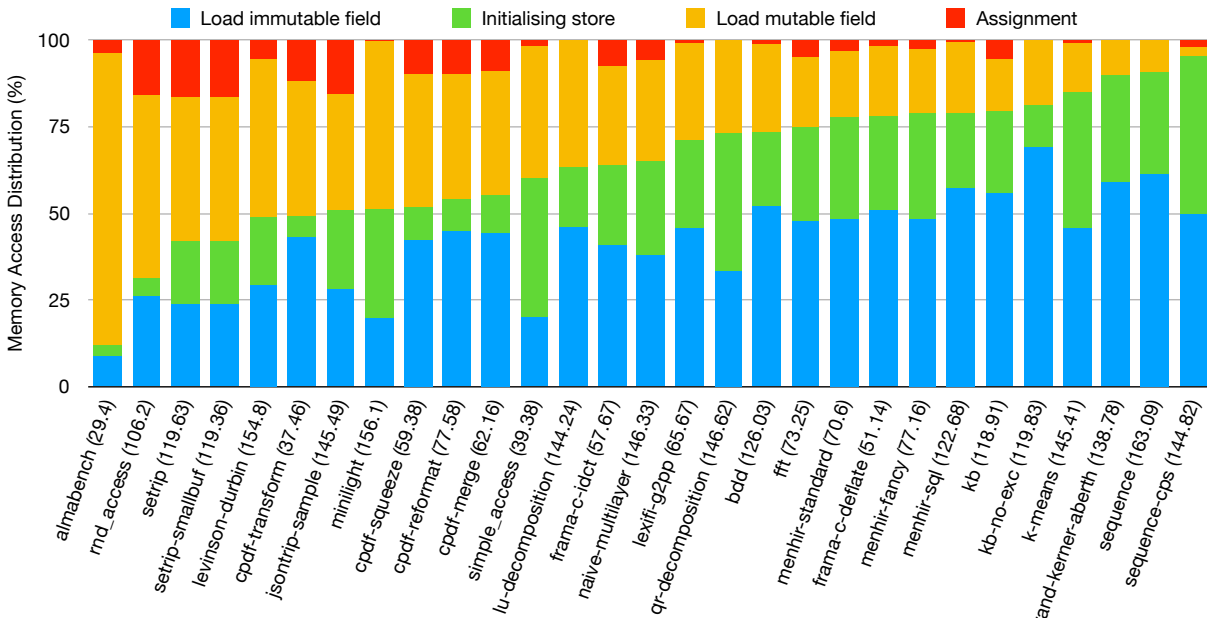
Our aim is to evaluate the suitability of the memory model to be the default one in Multicore OCaml [5], a parallel extension of OCaml with thread-local minor heaps and shared major heap. The compiler variants used in our evaluation are stock OCaml with memory model patches but not the parallel runtime, 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 virtualized server with 64Kb L1 data cache and 32K L1 instruction cache. Benchmarks were run sequentially.

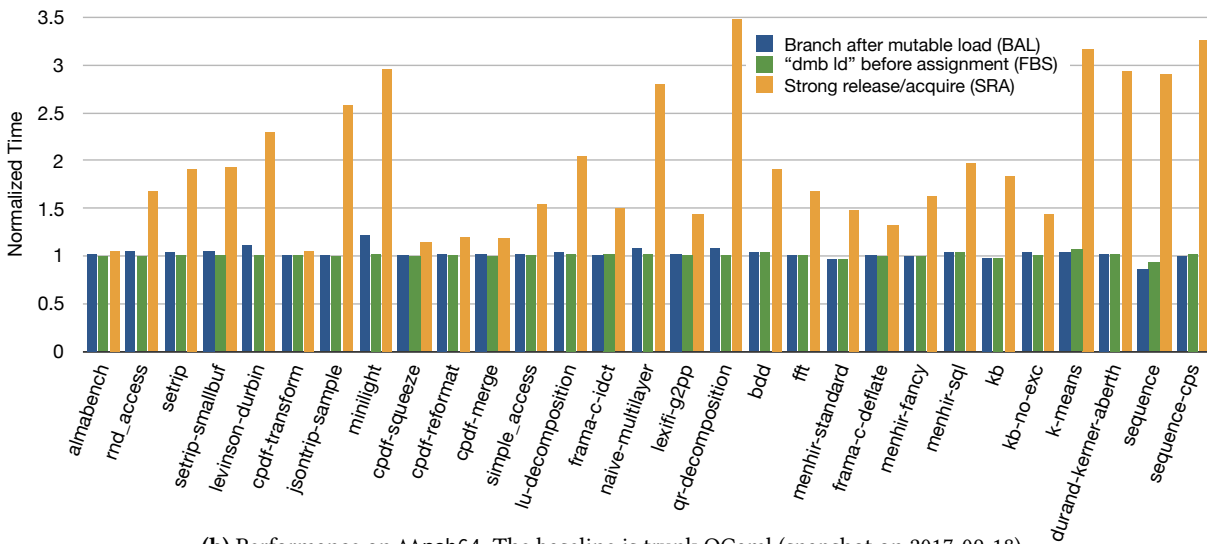
The OCaml benchmarks include a mix of workloads including parsers (`menhir`, `jsontrip`, `setrip`), utilities (`cpdf`), static analysis (`frama-c`) and numerical benchmarks (`lexifi-g2gpp`, `k-means`, `minilight`, `almabench`, etc.). Figure 7a shows the memory access distribution of the benchmarks.

7.1 Initialising stores and immutable loads

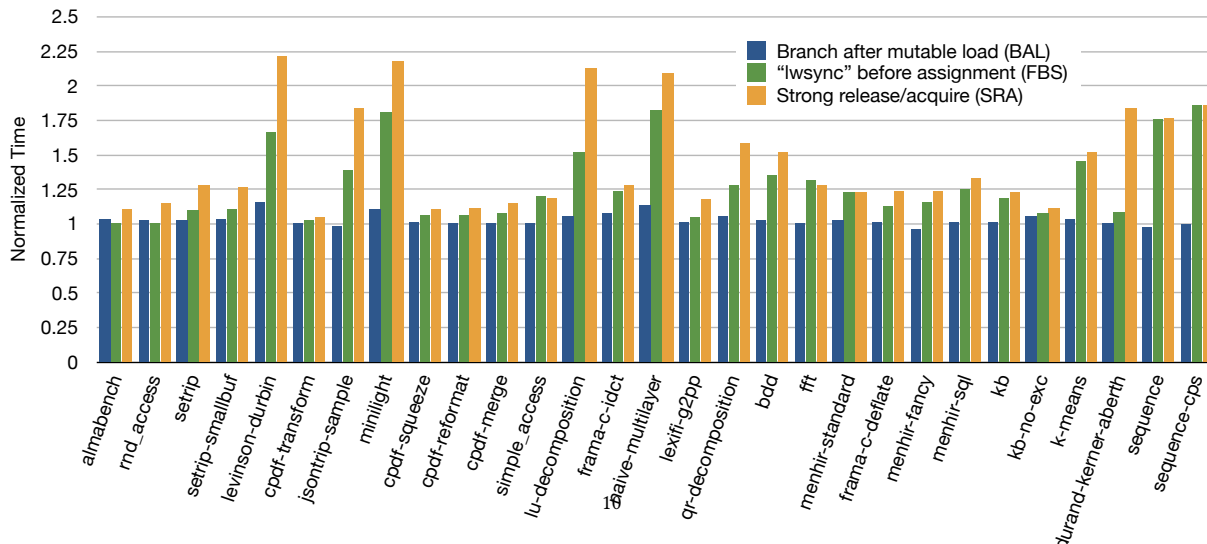
The Multicore OCaml heap layout with thread-local minor heaps and a shared major heap offers the opportunity to optimise initialising stores and loads from immutable fields.



(a) Memory access characteristics: The number in the parentheses indicate the access rate (millions/sec)



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



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

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. In a multi-threaded setting, we need to 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 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. 7a 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.

7.2 Assignments and mutable loads

We are now left only with imperative operations: assignments and loads from mutable fields. As we saw in §6.3, 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 5a), [`r <- ldr; cbz r, L; L:`], or a fence before store (FBS, as per table 5b) 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`]³.

For the sake of comparison, we also include strong release/acquire (SRA) [7], 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).

³`lwsync` is not a precise equivalent of ARM’s `dmb ld`, but both have the effect of preserving load-to-store ordering.

7.3 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. 7b and fig. 7c. 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 §6.3 that our goal is to prevent load-buffering behaviours (i.e 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 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.

8 Related work and discussion

8.1 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):

$$\begin{array}{l} x = a; \\ b = 1; \end{array} \parallel \begin{array}{l} a = b; \\ \end{array}$$

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”:

$$\text{if } (a == 1) \parallel \begin{array}{l} a = b; \\ b = 1; \end{array}$$

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 [2].

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 [4]. 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 minimal (§7).

8.2 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 [12].

The second 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 :

$$\begin{array}{l} x = b; \\ A = 1; \end{array} \parallel \begin{array}{l} A = 2; \\ b = 1; \end{array}$$

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 [9]. 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. [6] 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, Gianarakis and Vafeiadis [7] 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 §7.3). 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 at least as many fences as enforcing 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 [10, 11].

9 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. [6], or by extending the SRA model of Lahav et al. [7] with load-buffering-free nonatomics.

References

- [1] 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, Berlin, Heidelberg, 258–272. https://doi.org/10.1007/978-3-642-14295-6_25
- [2] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. *The Problem of Programming Language Concurrency Semantics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12
- [3] 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, Berkeley, CA, USA, 3–3. <http://dl.acm.org/citation.cfm?id=2001252.2001255>
- [4] 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, New York, NY, USA, Article 7, 6 pages. <https://doi.org/10.1145/2618128.2618134>
- [5] Stephen Dolan, Leo White, and Anil Madhavapeddy. 2014. Multicore OCaml. OCaml Workshop. (2014).
- [6] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 175–189. <https://doi.org/10.1145/3009837.3009850>
- [7] 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>
- [8] Chris Lattner. 2012. Random LLVM Notes. (2012). <http://www.nondot.org/sabre/LLVMNotes/>
- [9] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 378–391. <https://doi.org/10.1145/1040305.1040336>
- [10] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 351–362. <https://doi.org/10.1145/1806596.1806636>
- [11] 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>
- [12] William Pugh. 1999. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande (JAVA '99)*. ACM, New York, NY, USA, 89–98. <https://doi.org/10.1145/304065.304106>
- [13] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018)*. New York, NY, USA.
- [14] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 175–186. <https://doi.org/10.1145/1993498.1993520>
- [15] 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>
- [16] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 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 20. *For all transitions T , $F(T) \leq F'(T)$.*

Proof. Case analysis. $F'(T)$ is always one of $F(T)$, $F[a \mapsto t]$ for some $t > F(a)$ or $F_A \sqcup F(T)$, all of which are $\geq F(T)$. \square

Lemma 21. *Given a trace*

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

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 20, 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 $F'(T_i)$.

\square

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

Lemma 22. *For any nonatomic location a and trace*

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

then if $t \in \text{dom } F(T_j)(a)$ 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 $t \in \text{dom } F'(T_k)(a)$, either

- T_k is a write at timestamp t , in which case we choose $i = k$
- $t \in \text{dom } F(T_k)(a)$, 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 $t \in \text{dom } F(T_m)(a)$ and the inductive hypothesis applies.

\square

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

Lemma 23. *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. \square

Proof of theorem 13

Proof. Consider an arbitrary trace including M :

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

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 $F'(T')(a) < t$. Let T'' be the non-weak and therefore L -sequential transition given by lemma 23 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 21 implies $t = F'(T)(a) \leq F'(T')(a)$, 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 . \square

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 $(|\Sigma|, \text{po}_\Sigma, \text{rf}_\Sigma, \text{co}_\Sigma)$.

Lemma 24. *Excluding initial writes, the relations $\text{rf}_\Sigma, \text{po}_\Sigma$ and $\text{co}_{\Sigma, A}, \text{fr}_{\Sigma, A}$ for atomic locations A are subsets of $<_\Sigma$.*

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

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

Lemma 25. *For all non-initial $E_1, E_2 \in |\Sigma|$, $E_1 \text{ hb}_\Sigma E_2$ 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 hb_Σ uses the relations rf_Σ and co_Σ . For an atomic location A , relations $\text{rf}_{\Sigma, A}$, $\text{co}_{\Sigma, A}$ and $\text{fr}_{\Sigma, A}$ 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 $E_1 \text{ hb}_\Sigma E_2$, which we do by case analysis on the type of E_2 :

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

□

Proof of theorem 14

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

Causality When E_W is an initial write, there are no events $E' \text{ hb } E_W$ nor $E' \text{ rf}_\Sigma E_W$, and there are no $E' \text{ fr}_\Sigma E_W$ 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 $\text{hb}_\Sigma \cup \text{rf}_\Sigma \cup \text{fr}_{\Sigma, \text{at}}$.

For events other than initial writes, $\text{po}_{\Sigma, A}$, $\text{rf}_{\Sigma, A}$ and $\text{fr}_{\Sigma, A}$ (for A atomic location) are all subsets of $<_\Sigma$ (lemma 24), so a cycle in $\text{hb}_\Sigma \cup \text{rf}_\Sigma \cup \text{fr}_{\Sigma, \text{at}}$ implies one in $<_\Sigma$.

CoWW Since $E_2 \text{ co}_\Sigma E_1$, both are writes to some location. We may assume they are writes to a nonatomic location a, since $\text{co}_{\Sigma, A} \subseteq \text{hb}_\Sigma$ 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 $E_1 \text{ hb}_\Sigma E_2$, then by lemmas 21 and 25, $t_1 = F'(T(E_1)) \leq F'(T(E_2)) = t_2 < t_1$, 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 $E' \text{ rf}_\Sigma E_2$, $E' \text{ co}_\Sigma E_1$. 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 $E_1 \text{ hb}_\Sigma E_2$, then by lemmas 21 and 25, $t_1 = F'(T(E_1)) \leq F'(T(E_2)) \leq t' < t_1$, a contradiction.

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

Lemma 26. *If $(G, \text{po}, \text{rf}, \text{co})$ is a consistent execution, and $E \in G$ is an event such that there is no E' where $E \text{ rf } E'$, then the restriction of the execution to $G \setminus \{E\}$ 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 15

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 $\text{hb} \cup \text{rf} \cup \text{fr}_{\text{at}}$ 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 $E \text{ hb } E'$ or $E \text{ rf } E'$ or $E \text{ fr}_{\text{at}} E'$.

Let $G' = G \setminus \{E\}$. By lemma 26, the restriction of $(G, \text{po}, \text{rf}, \text{co})$ to G' is a consistent execution of size n , so by the inductive hypothesis there is a trace Σ' such that $|\Sigma'| = G'$ and $(G', \text{po}_{\Sigma'}, \text{rf}_{\Sigma'}, \text{co}_{\Sigma'})$ is consistent.

E is not an initial write, so let $((i, n), \ell, \phi) = E$. E has no po -successors (since $\text{po} \subseteq \text{hb}$), 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, $\phi = \text{write } x$: Rule WRITE-AT always applies.
- ℓ atomic, $\phi = \text{read } x$: Let $E_W \in G'$ be the unique event such that $E_W \text{ rf } E$. 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 , $E_W \text{ co } E'_W$ or $E'_W \text{ co } E_W$. If $E_W \text{ co } E'_W$, then $E \text{ fr}_{\text{at}} E'_W$, contradicting the fact that E has no fr_{at} -successors. If $E'_W \text{ co } E_W$, then $E'_W \text{ hb } E_W$, so $E'_W <_{\Sigma'} E_W$, a contradiction.
- ℓ nonatomic, $\phi = \text{write } x$: 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 22 there is some write event E_W with timestamp $t' > t$ which happens-before E . But then $E \text{ co}_\Sigma E_W$ and $E_W \text{ hb}_\Sigma E$ (by lemma 25), a violation of CoWW.
- ℓ nonatomic, $\phi = \text{read } x$: Let $E_W \in G'$ be the unique event such that $E_W \text{ rf } E$, 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 22 there is some other write event E'_W with timestamp $t' > t$ which happens-before E . But then $E \text{ fr}_\Sigma E'_W$ and $E'_W \text{ hb}_\Sigma E$ (by lemma 25), a violation of CoWR.

C Proofs about compilation

Proof of theorem 16

Proof. In concise notation, hb is:

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

This includes $\text{hbinit} \cup \text{hbcom} \cup \text{po}$. The difficult case is to show the converse, which we do by showing (i), that:

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

and (ii), that $hbinit \cup hbcom \cup po$ is transitive.

(i) First, note that since $rfi_{at} \subseteq po$, $po \cup rf_{at} = po \cup rfe_{at}$, and likewise for co , so it suffices to prove:

$$hbinit \cup po \cup rfe_{at} \cup coe_{at} \subseteq hbinit \cup hbcom \cup po$$

This holds since $rfe_{at} \cup coe_{at} \subseteq hbcom$.

(ii) We must show

$$(hbinit \cup hbcom \cup po); (hbinit \cup hbcom \cup po) \subseteq hbinit \cup hbcom \cup po$$

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

$$(hbcom \cup po); (hbcom \cup po) \subseteq hbcom \cup po$$

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

$$hbcom; hbcom \subseteq hbcom$$

Expanding the definition of $hbcom$,

$$\begin{aligned} hbcom; hbcom &\subseteq \\ po_{-at}; ((coe_{at} \cup rfe_{at}); po_{at-at})^*; (coe_{at} \cup rfe_{at}); po_{at-}; \\ po_{-at}; ((coe_{at} \cup rfe_{at}); po_{at-at})^*; (coe_{at} \cup rfe_{at}); po_{at-} &\subseteq \\ po_{-at}; ((coe_{at} \cup rfe_{at}); po_{at-at})^*; (coe_{at} \cup rfe_{at}); po_{at-}; \\ ((coe_{at} \cup rfe_{at}); po_{at-at})^*; (coe_{at} \cup rfe_{at}); po_{at-} &\subseteq \\ po_{-at}; ((coe_{at} \cup rfe_{at}); po_{at-at})^*; (coe_{at} \cup rfe_{at}); po_{at-} &= \\ hbcom & \end{aligned}$$

□

Proof of theorem 17

The original Causality axiom is:

$$acyclic(hb \cup rf \cup fr_{at})$$

Expanding hb by theorem 16, this is:

$$acyclic(hbinit \cup hbcom \cup po \cup rf \cup fr_{at})$$

Since $rfi, fri \subseteq po$, this is:

$$acyclic(hbinit \cup hbcom \cup po \cup rfe \cup fre_{at})$$

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:

$$acyclic(hbcom \cup po \cup rfe \cup fre_{at})$$

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:

$$hbcom \cup po_{at-} \cup po_{-at} \cup po_{RW} \cup rfe \cup fre_{at}$$

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

$$irreflexive(hb; (fr \cup co))$$

By theorem 16, this is equivalent to:

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

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

$$irreflexive((hbinit \cup hbcom \cup po_{con}); (fr \cup co))$$

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 18 and 19 ensure they must agree.

Proof of theorem 18 (Compilation to x86) Causality for x86

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

$$acyclic(hbinit \cup po \cup co_{at} \cup fr_{at} \cup rf)$$

or equivalently,

$$acyclic(hbinit \cup po \cup coe_{at} \cup fre_{at} \cup rfe)$$

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

$$acyclic(po \cup coe_{at} \cup fre_{at} \cup rfe)$$

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. 4). $co_{at} \cup fr_{at} \cup rf$ 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 coe_{at} , fre_{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 ($poghb \cup implied$) 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 17. Splitting it into three cases, we must show that these three relations are irreflexive:

$$hbinit; (fr \cup co)$$

$$hbcom; (fr \cup co)$$

$$po_{con}; (fr \cup co)$$

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

Proof of theorem 19 (Compilation to ARMv8) Causality for ARMv8

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

$$acyclic(hbinit \cup po \cup co_{at} \cup fr_{at} \cup rf)$$

or equivalently,

$$acyclic(hbinit \cup po \cup coe_{at} \cup fre_{at} \cup rfe)$$

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

$$acyclic(po \cup coe_{at} \cup fre_{at} \cup rfe)$$

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. 6). $co_{at} \cup fr_{at} \cup rf$ 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 coe_{at} , fre_{at} or rfe , it must be an atomic write (Rel), or a possibly-nonatomic read (R). Since E_2 is followed by coe_{at} , fre_{at} or rfe , it must be an atomic write (Rel), an atomic read (Acq) or a possibly-nonatomic write. This gives six cases:

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

□

Coherence for ARMv8

Proof. We use the formulation of Coherence from theorem 17. Splitting it into three cases, we must show that these three relations are irreflexive:

$$hbinit; (fr \cup co)$$

$$hbcom; (fr \cup co)$$

$$po_{con}; (fr \cup co)$$

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

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

$$po_{-at}; ((coe_{at} \cup rfe_{at}); po_{at-at}^*); (coe_{at} \cup rfe_{at}); po_{at-}$$

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

$$(coe_{at} \cup rfe_{at}); po_{at-}; (fr \cup co) \subseteq ob^+$$

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

$$(coe_{at} \cup rfe_{at}); po_{at-}; (ob \cup po) \subseteq ob^+$$

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

$$(coe_{at} \cup rfe_{at}); po_{at-} \subseteq ob^+$$

If $E_1 po_{at-} E_2$, 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:

$$coe_{at}; (po \cap (Rel \times M)) \subseteq ob^+$$

$$rfe_{at}; (po \cap (Acq \times M)) \subseteq ob^+$$

The second case is easy, since $(po \cap (Acq \times M)) \subseteq ob$. In the first case, consider $E_1 coe_{at} E_2$. 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 $coe \subseteq ob$ if E_1 and E_W are on different threads, or by $po \cap (M \times Rel) \subseteq ob$ 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:

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

so $E_1 ob^+ E_3$. □

D Java Concurrency Stress Tests

`jctest` is a test harness to help uncover concurrency behaviour in the JVM. The code below tests the coherence of non-volatile writes.

```

package ldrf;

import org.openjdk.jcstress.annotations.*;
import org.openjdk.jcstress.infra.results.IntResult3;
import static org.openjdk.jcstress.annotations.Expect.*;

@JCStressTest
@Outcome(id = "0, .*", expect = ACCEPTABLE, desc = "Didn't synchronise")
@Outcome(id = "1, 1, 1", expect = ACCEPTABLE, desc = "Synchronised, final result 1")
@Outcome(id = "1, 2, 2", expect = ACCEPTABLE, desc = "Synchronised, final result 2")
@Outcome(id = "1, 2, 1", expect = FORBIDDEN, desc = "Synchronised and yet incoherent!")
@Outcome(id = "1, 1, 2", expect = FORBIDDEN, desc = "Synchronised and yet incoherent!")
@State
public class ConcurrencyTest {

    private final Holder h1 = new Holder();
    private final Holder h2 = h1;

    private static class Holder {
        int a;
        volatile int flag;
        int trap;
    }

    @Actor
    public void actor1() {
        h1.a = 1;
        h1.flag = 1;
    }

    @Actor
    public void actor2(IntResult3 r) {
        Holder h1 = this.h1;
        Holder h2 = this.h2;

        h1.trap = 0;
        h2.trap = 0;
        h1.a = 2;
        r.r1 = h2.flag;
        r.r3 = h1.a;
        r.r2 = h2.a;
    }
}

```