Bounding Data Races in Space and Time

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:

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 compositionalitity. 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.

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, \( b = a + 10 \). (C++)
- **Possible result** Afterwards, \( b \neq a + 10 \). (C++)

**Explanation** Consider the following multithreaded program, where \( c \) is a nonatomic global variable:

\[
\begin{align*}
  c &= a + 10; \\
  \ldots &\text{// some computation} \\
  c &= 1; \\
  b &= a + 10; \\
\end{align*}
\]

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:

\[
\begin{align*}
  t &= a + 10; \\
  c &= t; \\
  \ldots &\text{// some computation} \\
  b &= t; \\
\end{align*}
\]

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 re-materialise \( t \) from \( c \) instead of allocating a new stack slot\(^1\), giving:

\[
\begin{align*}
  t &= a + 10; \\
  c &= t; \\
  \ldots &\text{// some computation} \\
  b &= t; \\
\end{align*}
\]

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 \( a \) \(^2\). 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.

---

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].

2. This is a point where the results of the reads depend on the order of the writes, which is not a data race.

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, \( b \neq c \). (C++, Java)

**Explanation** Consider this program, where \( f \) is an atomic (volatile in Java) boolean, initially false.

\[
\begin{align*}
  a &= 1; \\
  f &= \text{false}; \\
  c &= a; \\
\end{align*}
\]

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 \( C \) (\( \text{int} \ x; \)).

Example 3. \( C \ c = \text{new} \ C(); \ c.x = 42; \ a = \text{c}.x; \)

- **Assumption** There are no other accesses to \( a \).
- **Expected result** Afterwards, \( a = 42 \).
- **Possible result** Afterwards, \( a \neq 42 \). (C++, Java)
**Explanation** Here, we know that there cannot be any data races in the past on the location : 2 , since : 2 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 subsequent data races. Consider this pair of threads:

\[
\begin{align*}
\text{C} & \text{ c} = \text{ new C()} ; \\
& \text{ c. x} = 42 ; \\
& \text{ a} = \text{ c. x} ; \\
& \text{ g} = \text{ c} ; \\
& \text{ g. x} = 7 ;
\end{align*}
\]

The read of : 2 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 : 2 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 accompanying technical report [11]. 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 \( A, B, \ldots \) and nonatomic locations \( a, b, \ldots \), in which may be stored values \( x, y \in \mathcal{V} \).

The program interacts with memory by performing actions \( \phi \) 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 \( \phi \) applied to the location \( \ell \).

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

\[
C; F \xrightarrow{\ell : \phi} C'; F'
\]

which specifies that when a thread with frontier \( F \) performs an action \( \phi \) on location \( \ell \) 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.
A step may or may not involve performing an action, giving two distinct types of transition:

\[
\begin{align*}
\text{(silent)} & \quad e \xrightarrow{\epsilon} e' & \quad (S, P[i \mapsto (F, e)]) \rightarrow (S, P[i \mapsto (F, e')]) \\
\text{(memory)} & \quad e \xrightarrow{\ell \phi} e' & \quad S(\ell); F \xrightarrow{\ell \phi} C'; F' \\
\end{align*}
\]

where \( \epsilon \) 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 \( e \xrightarrow{\ell \phi} e' \), then for every \( y \), \( e \xrightarrow{\ell \phi} e_y \) 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 \( M = (S, P) \), 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 \( \ell \) (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:

\[
M_0 = \left\{ (a \mapsto (0 \mapsto v_0), A \mapsto (F_0, v_0)) \right. \\
\left. (i \mapsto F_0, e_i \text{ for } i \in \mathcal{I}) \right\}
\]

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 \( T \vdash M \rightarrow M' \).

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

\textbf{Definition 6 (Weak transition).} A weak transition is a ma-
chine step performing a memory operation of one of the fol-
lowing forms:

- \( H; F \xrightarrow{\text{a read } x} H; F \) when \( H(t) \neq x \) for the largest times-
tamp \( t \in \text{dom}(H) \). 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 opera-
tions on atomic values, or operations on nonatomic values
which access the element of history with the largest time-
tamp. 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:

\textbf{Definition 7 (Sequentially consistent traces).} A trace is se-
quentially 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 intro-
duce the happens-before relation.

\textbf{Definition 8 (Happens-before).} Given a trace

\[ M_0 \xrightarrow{T_i} M_1 \xrightarrow{T_j} \ldots \xrightarrow{T_n} M_n \]

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.

\textbf{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.

\textbf{Definition 10 (Data race).} Given a trace

\[ M_0 \xrightarrow{T_i} M_1 \xrightarrow{T_j} \ldots \xrightarrow{T_n} M_n \]

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 \( T_j \).

4.3 \( L \)-stability

\textbf{Definition 11 (L-sequential transitions).} Given a set \( L \) of lo-
cations, 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.

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

\[ M_0 \xrightarrow{T_i} M_1 \xrightarrow{T_j} \ldots \xrightarrow{T_n} M'_n \]

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),
but there are no data races between one operation before \( M \)
and one operation afterwards.

4.4 The local DRF theorem

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

\[ M \xrightarrow{T_i} M_1 \xrightarrow{T_j} \ldots \xrightarrow{T_n} M_n \]

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:

\textbf{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 \( T' \):
\[
M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} M_n \xrightarrow{T'} M'
\]
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. \( c = \text{new} \ C(); c.x = 42; a = c.x \);

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. \( L = \{a, b\} \)
2. \( L = \{a, b, c\} \)
3. \( L = \{a, c, c.x\} \)

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).

**Figure 2.** Generating events from programs

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, \( \ell \) is a location and \( \phi \) 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 \( IW(\ell) \), indicating the initial write of \( v_0 \) to location \( \ell \) 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 \( M_0 \):
\[
G_0 = \{(IW(\ell), \ell, v_0) | \ell \in L \}
\]

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 \( EW \) \( rf \) \( ER \), then \( EW \) is a write event and \( ER \) is a read event, both having the same location and value.
- For every read event \( ER \in G \), there is a unique event \( EW \in G \) such that \( EW \) \( rf \) \( ER \).
- If \( E_1 \) \( co \) \( E_2 \), then \( E_1 \) and \( E_2 \) are write events to the same location.
- For each location \( \ell \), \( co \) 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.
• \(E_1 \text{ hb } E_2\) whenever \(E_1 \text{ po } E_2\).
• \(E_1 \text{ hb } E_2\) 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 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 \(E_1\)

### 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:

\[
|0| = G_0
\]

\[
|M_0 \cdots M_n|_{\Sigma}^{T_{n+1}} M_{n+1} = |\Sigma|
\]

if the transition \(T_{n+1}\) is silent, and

\[
|M_0 \cdots M_n|_{\Sigma}^{T_{n+1}} M_{n+1} \Sigma = |\Sigma| \cup \{(i, m, \ell, \phi)\}
\]

if the transition \(T_{n+1}\) uses memory with \(\ell, \phi\) and \(\Sigma\) contains \(m\) prior memory operation steps on thread \(i\).

If \(\Sigma\) is a trace, then the events of \(|\Sigma|\) correspond to the memory operation steps (fig. 1c) of \(\Sigma\). For an event \(E \in |\Sigma|\) (other than an initial write), we write \(T(E)\) for the corresponding transition in \(\Sigma\). 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 \(\Sigma\). From any \(\Sigma\), we construct a candidate execution \(|\Sigma|, po_\Sigma, rf_\Sigma, co_\Sigma\) as follows:

- \(po_\Sigma\) 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 \(I_WA\) 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 \(I_WA\) 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 15** (Soundness of axiomatic semantics). For all \(\Sigma\), \((|\Sigma|, po_\Sigma, rf_\Sigma, co_\Sigma)\) is a consistent execution.

**Theorem 16** (Completeness of axiomatic semantics). Every consistent \((G, po, rf, co)\) is \((|\Sigma|, po_\Sigma, rf_\Sigma, co_\Sigma)\) for some \(\Sigma\).

### 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 \(E_1 (R_1; R_2) E’\) if there is some \(E''\) such that \(E_1 R_1 E''\) 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 \(R^2\) 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:

\[
rf_i = rf \cap po \quad \text{rfe} = rf \cap po
\]

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:

\[
po_{at,:};((co_{at, \cup rfe_{at}}); po_{at,-at})^.; (co_{at, \cup rfe_{at}}); po_{at,-at}
\]

These relations characterise the hb relation precisely:

**Theorem 17**. For any candidate execution, hb = hbinit \(\cup\) hbcom \(\cup\) 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:
\[ \text{hbcom} \cup \text{po}_{\text{at}} \cup \text{po}_{\text{st}} \cup \text{po}_{\text{RW}} \cup \text{rfe} \cup \text{fre}_{\text{at}} \]

**Coherence** The following relation is irreflexive:
\[ (\text{hbinit} \cup \text{hbcom} \cup \text{po}_{\text{con}}) \cap (\text{fr} \cup \text{co}) \]

### 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:

- \( \text{po}_{\text{at}} \): Operations must not be moved before prior atomic operations.
- \( \text{po}_{\text{st}} \): Operations must not be moved after subsequent atomic writes.
- \( \text{po}_{\text{RW}} \): Prior reads must not be moved after subsequent writes.
- \( \text{po}_{\text{con}} \): 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, \( r_1, r_2 \) are registers and \( x, y \) are values.

**Sequentialisation:** \[ [P || Q] \Rightarrow [P; Q] \] Since replacing a parallel computation 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):** \[ [r_1 = a; r_2 = a] \Rightarrow [r_1 = a; r_2 = r_1] \] 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):** \[ [a = x; r_1 = a] \Rightarrow [a = x; r_1 = x] \] 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 \( \text{po}_{\text{RR}}, \text{po}_{\text{RW}} \) and \( \text{po}_{\text{W}} \) be the program order relations between reads, write to reads and writes, respectively.

**Common subexpression elimination:** \[ [r_1 = a \star 2; r_2 = b; r_3 = a \star 2] \Rightarrow [r_1 = a \star 2; r_2 = r_1; r_3 = b] \] where the first step involves relaxing \( \text{po}_{\text{RR}}, \) which is permitted by the memory model.

**Loop-invariant code motion:** \[ [\text{while} (...) \{ a = b; r_1 = c \star c; \ldots \}] \Rightarrow [r_2 = c \star c; \text{while} (...) \{ a = b; r_1 = r_2; \ldots \}] \] which involves relaxing \( \text{po}_{\text{RR}} \) and \( \text{po}_{\text{W}} \), both of which are permitted.

**Dead store elimination:** \[ [a = 1; b = c; a = 2] \Rightarrow [b = c; a = 1; a = 2] \] where the first step relaxes \( \text{po}_{\text{W}} \) and \( \text{po}_{\text{W}} \), both of which are permitted.

**Constant propagation:** \[ [a = 1; b = c; r = a] \Rightarrow [b = c; a = 1; r = 1] \] where the first step relaxes \( \text{po}_{\text{W}} \) and \( \text{po}_{\text{W}} \), 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: \[ [r_1 = a; b = c; a = r_1] \Rightarrow [r_1 = a; b = c; a = 1; r = 1] \] where the first step relaxes \( \text{po}_{\text{W}} \) and \( \text{po}_{\text{W}} \), both of which are permitted.

### 7.2 Compilation to x86-TSO

The first compilation target is the x86-TSO memory model [22], for which we use the axiomatic presentation of Alglave et al. [6]. The compilation model is shown in table 3.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonatomic read</td>
<td>mov R, [x]</td>
</tr>
<tr>
<td>Nonatomic write</td>
<td>mov [x], R</td>
</tr>
<tr>
<td>Atomic read</td>
<td>mov R, [x]</td>
</tr>
<tr>
<td>Atomic write</td>
<td>(lock) xchg R, [x]</td>
</tr>
</tbody>
</table>

*The lock prefix is implicit on the xchg instruction.*

Table 3. Compilation to x86-TSO
Definitions of sets of events:
\[ M = \text{all events} \quad R = \text{read events} \quad W = \text{write events} \]
\[ WA = \text{atomic write events (those with a rmw-predecessor)} \]

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

Conditions:
\[ \text{acyclic} (\text{poloc} \cup \text{rfe} \cup \text{fr} \cup \text{co}) \]
\[ \text{acyclic} (\text{ghb}) \]
\[ \text{rmw} \cap (\text{fre} ; \text{coe}) = \emptyset \]

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. [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, po, rf, co, rmw) is a 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 \( \text{rmw} \cap (\text{fre}; \text{coe}) \) 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, po, rf, co) is compiled to (G', po', rf', co', rmw') if there are functions \( \phi \) from G to G' and \( \phi_{WA} \) from the atomic writes of G to G' such that:

- \( \phi(E) \) has the same action type (read or write) as E
- \( \phi(E_1) \text{ po'} \phi(E_2) \) 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 \( E_W \)

This definition encodes the scheme of table 3, in particular by mapping atomic writes to read-modify-write instructions.

Theorem 19 (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.

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 \( \cup \text{rf} \). The classic example is as follows:

\[
\begin{align*}
\text{ldr } R0, [x] & \quad \text{ldr } R0, [y] \\
\text{mov } R1, #1 & \quad \text{mov } R1, #1 \\
\text{str } R1, [y] & \quad \text{str } R1, [x]
\end{align*}
\]

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 \( \text{dmb } 1d \) barrier before stores, shown in tables 5a and 5b 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 \( \text{stlr} \) instruction directly. Our atomics make stronger guarantees than those provided by the \( \text{stlr} \) and \( \text{ldar} \) 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 \( \text{dmb } 1d \) on atomic loads), between atomic stores and subsequent nonatomic stores (for which we add \( \text{dmb } 1s \) on atomic stores), and between atomic stores and subsequent nonatomic loads (for which we code atomic stores as exchanges). In each case, the full \( \text{dmb} \) 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 (\( \text{ldar}, \text{stlr}, \text{ldaxr}, \text{stlxr} \)) or not (\( \text{ldr}, \text{str} \)). A candidate execution (G, po, rf, co) is compiled to an ARM-candidate execution (G', po', rf', co', rmw') if there are functions \( \phi \) from G to G' and \( \phi_{WA} \) from the atomic writes of G to G' such that:

- \( \phi(E) \) has the same action type (read or write) as E
- \( \phi(E_1) \text{ po'} \phi(E_2) \) 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 \( E_W \)

This definition encodes the scheme of table 3, in particular by mapping atomic writes to read-modify-write instructions.
We now quantify the performance impact of our memory model. Theorem 20 (Soundness of compilation to ARMv8) The soundness theorem parallels that for x86:

\[ E_1 \triangleq \phi (E_1) \text{ co } \phi (E_2) \iff E_1 \text{ co } E_2 \]

\[ W \triangleq \phi_{AW} (E_W) \text{ rmw' } \phi (E_W) \text{ for atomic writes } E_W \]

The definition is similar to the case for x86, with the additional complication of needing to preserve read-to-write ordering, by ensuring that \( po_{RW} \subseteq \text{obs} \) (by inserting branches, \( \text{dmb} \ 1d \), or another means).

An ARM-candidate execution is \textit{ARM-consistent} if it satisfies the rules of fig. 6, which is an abridged version of the multi-copy atomic ARMv8 specification. Ignoring the \{ \ldots \} 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:

\textbf{Theorem 20} (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.


\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Operation} & \textbf{Implementation} \\
\hline
Nonatomic read & \text{ldr} R, [x]; \\
Nonatomic write & \text{str} R, [x] \\
Atomic read & \text{dmb} 1d; \text{ldar} R, [x] \\
Atomic write & \text{ldaxr}; \text{stlxr}; \text{cbnz} L; \text{dmb} st \\
\hline
\end{tabular}
\caption{Compilation to ARMv8 (AArch64)}
\end{table}

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.

The OCaml benchmarks include a mix of workloads including parsers (menhir, jsonstrip, setrip), utilities (cpdf), static analysis (frama-c) and numerical benchmarks (lexifig2gpp, k-means, minilight, almbench, etc.). The benchmarks were run one after the other, with the other available cores running runtime to simulate a loaded machine. Figure 7a 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.
(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 7. Performance results
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 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 1d 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.

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 5a), \([r \leftarrow 1\mathrm{d}; \mathrm{cbz} \ r, \ L; \ L:]\), or a fence before store (FBS, as per table 5b) that orders prior loads before the store (\([\mathrm{dmb} 1\mathrm{~d}; \ \mathrm{str}]\)). On PowerPC, the equivalent instruction sequences are \([r \leftarrow 1\mathrm{~d}; \ \mathrm{cmpi} \ r, \ 0; \ \mathrm{beq} \ L; \ L:]\) and \([\mathrm{lwsync} \ ; \ \mathrm{st}]\).

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 \([r \leftarrow 1\mathrm{~d}; \ \mathrm{cmpi} \ r, \ 0; \ \mathrm{beq} \ L; \ L: \ \mathrm{isync}]\) on PowerPC) and assignments are store release (\([\mathrm{stlr}]\) on AArch64 and \([\mathrm{lwsync} \ ; \ \mathrm{st}]\) 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. 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 §7.3 that our goal is to prevent load-buffering behaviours (i.e. \(p_{o\mathrm{nw}}\) reorderings). BAL precisely avoids this reordering permitting the processor to reorder other operations, and is the optimal compilation scheme in terms of reorderings allowed. \(\mathrm{dmb} 1\mathrm{~d}\) 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 \(\mathrm{dmb} 1\mathrm{~d}, \ \mathrm{lwsync}\) 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 \(\mathrm{dmb} 1\mathrm{~d}\) after and \(\mathrm{dmb} \ \mathrm{st}\) 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):

\[
x = a; \\
b = 1;
\]

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":

\[
\text{if } (a == 1) \\
a = b; \\
b = 1;
\]
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].

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 Demskey [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 difference is that our atomic writes have stronger semantics, which is why we use atomic exchanges instead of s1r on ARMv8. Consider the following, using an atomic (SC atomic) location A and a nonatomic (relaxed) location b:

\[
\begin{align*}
  x &= b; \\
  A &= 2; \\
  x &= b; \\
  A &= 1; \\
  b &= 1;
\end{align*}
\]

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 \( b = 1 \) 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, Gian-narakis 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, 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 Cambridge Relaxed Memory Concurrency group for help understanding the ARM hardware models, and Suresh Jagannathan, Gowtham Kaki, 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

References