A Calculus for Relaxed Memory

Karl Crary  Michael J. Sullivan
Carnegie Mellon University

Abstract

We propose a new approach to programming multi-core, relaxed-memory architectures in imperative, portable programming languages. Our memory model is based on explicit, programmer-specified requirements for order of execution and the visibility of writes. The compiler then realizes those requirements in the most efficient manner it can. This is in contrast to existing memory models, which—if they allow programmer control over synchronization at all—are based on inferring the execution and visibility consequences of synchronization operations or annotations in the code.

We formalize our memory model in a core calculus called RMC. Outside of the programmer’s specified requirements, RMC is designed to be strictly more relaxed than existing architectures. It employs an aggressively nondeterministic semantics for expressions, in which actions can be executed in nearly any order, and a store semantics that generalizes Sarkar, et al.’s and Alglave, et al.’s models of the Power architecture. We establish several results for RMC, including sequential consistency for two programming disciplines, and an appropriate notion of type safety. All our results are formalized in Coq.

1 Introduction

Modern multi-core computer architectures employ relaxed memory models, interfaces to memory that are considerably weaker than the conventional model of sequential consistency [15]. In a sequentially consistent setting, an execution is consistent with some interleaving of the individual memory operations. Thus, the programmer may assume that (a) all threads have a common view of memory that agrees on the order in which memory operations are performed, and (b) that commonly-agreed order respects program-order in regard to operations issued by the same thread.

Relaxed memory architectures provide no such common view of memory. Although most do enforce a globally agreed order on writes to individual locations, the order does not extend to multiple locations. Indeed, on some architectures (notably Power [13] and ARM [11]) it is more useful to start from a view of memory as a pool of available writes [19, 5] (and then impose structure), than it is to start from memory as a mapping of locations to values and then try to weaken it.

Moreover, modern architectures also execute instructions out of order. While most do out-of-order execution in a fashion that is undetectable to single-threaded programs, on some important relaxed-memory architectures (e.g., Power and ARM again) it is possible for multi-threaded programs to expose out-of-order execution.

As illustrated by Adve and Gharachorloo [1] and others [2, 19], these are two distinct phenomena. Relaxed memory behaviors cannot always be reduced to just one or the other. This point is often not clearly understood by programmers, or even by authors of documentation, which sometimes give rules governing when reads and writes might be reordered, without specifying the “order” in question.

Relaxed memory architectures can be challenging to program. Single-threaded code, and also concurrent code free of data races (in which accesses to shared variables are protected by mutexes), usually require no special effort, but implementing lock-free data structures, or implementing the mutexes themselves can be very delicate indeed.

In the past the problem was made particularly challenging by the lack of clear specifications of the memory models. Recent work has addressed this difficulty for some important architectures [20, 19, 4, 3, 5], but for portable, imperative programming languages the memory models are either insufficiently expressive or quite complex:

In Java [16] and C/C++ [10, 8, 7] the memory model is divided into two parts: a simple mechanism for data-race-free programming, and a less simple one for lock-free data structures and for low-level implementation of synchronization utilities. Like much of the work on architecture models, our primary concern in this paper is the second aspect; our approach to data-race-free programs—discussed in Section 9—is conventional.

Java’s volatile variables and C/C++’s atomic variables need not be used within a mutex, and thus are suitable for lock-free data structures and synchronization implementation. Java’s volatiles are guaranteed to be sequentially consistent; this provides a convenient programming model, but it also imposes a cost which can be undesirable in particularly performance-critical code. C/C++’s atomics are more flexible. The programmer annotates each operation on an atomic with a “memory order” that indirectly dictates the semantics of the operation, which may be as strong as sequentially consistent or may be weaker.

Some of the rules governing C/C++ memory orders are quite complicated. Consider the example below (for illustrative purposes only, the details will not be important in the sequel) which has the standard “message passing” idiom at its core:
The unlabelled edges are the program order. The dashed red edges are witness relations derived directly from the trace: Ry4 reads from \((\text{acquire})\) Wy4. The coherence order \((\text{co})\), the global ordering among writes to the same location) on x is Wx0, Wx1, Wx3, and on y is Wy2, Wy4. From these edges the rules of C/C++ define a plethora of derived edges (shown in dotted blue):

Since Wy2 is marked with “release” order, it is considered a release action and forms the head of a release sequence (not shown, to avoid cluttering the diagram) that also includes Wy4. Since Ry4 is marked with “acquire” order and reads from Wy2’s release sequence, Wy2 synchronizes-with Ry4. (Wy4 does not synchronize with Ry4 because Wy4 has relaxed order and thus is not a release action.) Consequently, Wx1 happens-before Rx?. Thus Wx1 is a visible side effect to Rx? (because there is no other write to x that intervenes by happens-before), and its visible sequence of side effects (not shown) consists of itself and Wx3. An atomic read must come from its visible sequence of side effects, so Rx? must read from either Wx1 or Wx3, not Wx0.

On the other hand, if Wy4 had “release” order, then Wy4 (not Wy2) would synchronize-with Ry4, which would mean that Wx3 happens-before Rx?, so Wx3 (not Wx1) would be a visible side effect to Rx?. The visible sequence of side effects would contain only Wx3, so Rx? could read only from Wx3.

This is just a simple example; the required reasoning can be much more complex. Indeed, the key C/C++ notion of happens-before is intentionally not transitive (!), to enable certain optimizations on architectures like Power and ARM when using the “consume” memory order.

1.1 The RMC Memory Model

We propose a different model for low-level programming on relaxed memory architectures. Rather than have the programmer specify annotations that indirectly determine the relations governing which writes can satisfy which reads, we allow the programmer to specify the key relations directly. It is then the compiler’s job to generate code to realize them.

As before, the solid black edges are given by the program, and the dashed red edges are witness relations derived from the trace. The unlabelled edges are again program order. In addition, the program specifies visibility-order \((\text{vo})\) and execution-order \((\text{xo})\) edges. In this example, to bring about the same outcome as the C/C++ example, the programmer indicates that Wx1 is visibility-ordered before Wy4, and that Ry4 is execution-ordered before Rx?. The visibility specification means that any thread that can see Wy4 must also see Wx1, and the execution specification means that Rx? must occur after Ry4. Thus, Rx? can see Wy4 (since Rx? takes place after its thread reads from Wy4), so it must also see Wx1, and consequently it cannot read from the overwritten Wx0. A programmer who wanted to exclude Wx1 as well would indicate that Wx3 is visibility-ordered before Wy4.

In this paper we present RMC (for Relaxed Memory Calculus), a core calculus for imperative computing with relaxed memory that realizes this memory model. RMC is intended to admit all the strange (to a sequentially consistent eye) executions permitted by relaxed memory architectures. Thus, an RMC program can be implemented on such architectures with no synchronization overhead beyond what is explicitly specified by the programmer.

Importantly, we do not attempt to capture the precise behavior of any architecture. On the contrary, RMC is specifically intended to be strictly more relaxed (in other words, even more perverse) than any existing architecture. This is for two reasons: First, because we find it more elegant to do so. Second, because we hope that by doing so, we can future-proof RMC against further innovations by computer architects.

Thus, RMC makes only five assumptions regarding the architecture:

1. Single-threaded computations (meaning single-threaded programs, and also the critical sections of properly synchronized programs) respect sequential consistency.
2. There exists a strict partial order (called coherence order) on all writes to the same location, and it is respected by reads.
3. The message passing idiom (illustrated above, and discussed in more detail in Section 7) works.

\footnote{\textsuperscript{1}in the terminology of Batty, et al. \cite{batty2001}}
\footnote{\textsuperscript{2}C/C++ refers to program order as “sequenced before” and to the coherence order as “modification order”. We use our terms to maintain consistent terminology throughout the paper.}
4. There exists a push mechanism (this corresponds to Power's \textit{sync}, ARM's \textit{dmb}, and x86's \textit{mfence}).

5. There exists a mechanism for atomic read-write operations (also known as atomic test-and-set).

Some older architectures might not satisfy 3, 4, or 5, but the importance of these assumptions is now well-understood and we expect that future architectures will. An architecture might also provide some atomic read-modify-write operations; RMC accounts for such, but does not require them.

Alglave, \textit{et al.} [5], another architecture-independent formalism, proposes axioms similar in spirit to these assumptions. We compare them to ours in Section 10.

We take as our reference points the (broadly similar) Power and ARM architectures, and the x86 architecture, because they enjoy rigorous, usable specifications [19, 5, 20]. We focus on the former because in all cases relevant to this paper, the complexities of Power and ARM subsume those of x86.

Like Sarkar, \textit{et al.}'s model for Power [19], and Sewell, \textit{et al.}'s model for x86 [20], our calculus is based on an operational semantics, not an axiomatic semantics. Nevertheless, one aspect of our system does give it some axiomatic flavor. Our rules for carrying out storage actions require that one aspect of our system does give it some axiomatic flavor. Our rules for carrying out storage actions require that the coherence order remain acyclic (assumption 2, above), but to afford maximum flexibility, we do not impose any particular protocol to ensure this. Thus, when reasoning about code, this requirement functions much like an axiom. Alglave, \textit{et al.}'s intermediate machine [5] has a somewhat similar flavor.

The paper makes several contributions:

- We show how to program relaxed-memory architectures using programmer-specified visibility- and execution-order edges.
- We give an elegant calculus capturing out-of-order and speculative execution.
- We give an aggressively relaxed model of memory. It accounts for the write-forwarding behavior of Power and ARM, and a "leapfrogging write" behavior on ARM, in addition to other possible phenomena not yet observed or even implemented.
- We prove three main theorems:
  - A type safety result (including a novel adaptation of progress to nondeterministic execution).
  - A sequential-consistency theorem showing that sequential consistency can be obtained by interleaving instructions with memory barriers. This theorem generalizes a theorem of Batty, \textit{et al.} [6] for the more permissive semantics of RMC, and also streamlines the proof.
  - A second sequential-consistency theorem showing that data-race-free executions enjoy sequential consistency. This theorem adapts a standard result to RMC.

All the results of the paper are formalized in Coq. The formalization may be found at:

\texttt{www.cs.cmu.edu/~crary/papers/2014/rmc.tgz}

### 2 Tagging

RMC's main tool for coping with relaxed memory is user-specified edges indicating visibility order (anyone who sees the latter action can see the former) and execution order (the former action must be executed before the latter). It is easy enough to indicate required orderings between operations by drawing edges on an execution trace, but, of course, execution traces are not programs. How are specified orders rendered in source code?

We do this by using tags to identify operations, and including a declaration form to express required edges between tags. This is illustrated in the following piece of code, which implements the middle thread of the example from Section 1.1 in an extension of C using the RMC memory model:

```c
for (i = 0; i < 2; i++) {
    L(before, x = i);
    y = 2;
    x = 3;
    L(after, y = 4);
}
```

Here, the write \( x = 1 \) (i.e., Wx1) is tagged \textit{before}, and the write \( y = 4 \) (i.e., Wy4) is tagged \textit{after}. The declaration "\texttt{VEDGE(before, after)}" indicates that a visibility edge exists between operations tagged \textit{before} and \textit{after}. If we wanted to make Wx3 also visibility ordered before Wy4, we could declare additional tags and edges, or just add the \texttt{before} tag to \( x = 3 \).

Visibility order implies execution order, because it makes little sense to say that \( i \) should be visible to anyone who can see \( i' \), if \( i' \) can be seen before \( i \) even takes place. We can get execution order alone using the \texttt{XEDGE} keyword.

Tagging creates edges only between operations in program order. Thus, we cannot require an operation to execute before an earlier operation.

This is particularly important when tags appear within loops:

```c
for (i = 0; i < 2; i++) {
    L(after, x = i);
    L(before, y = i + 10);
}
```

Here, the visibility edge reaches from each \( y \)-assignment to the \( x \)-assignment in the next iteration of the loop:

\[
\begin{align*}
Wx0 &\rightarrow Wy10 \rightarrow Wx1 \rightarrow Wy11
\end{align*}
\]

We do not generate nonsensical edges upstream from Wy10 to Wx0, or from Wy11 to Wx1.

Also observe that tags generate edges between all operations with appropriate tags, not only the nearest ones. For example, reversing the tags in the previous code to obtain:

```c
for (i = 0; i < 2; i++) {
    L(before, x = i);
    L(after, y = i + 10);
}
```

generates:
Note the long edge from Wx0 to Wy11. This long edge is desired, as we will see in an example in Section 5.

It is also sometimes useful to impose visibility and execution order more broadly than can be done with individual edges. For example, the code to enter a mutex needs to be execution-ordered before everything in the following critical section, and the code to exit a mutex needs to be visibility-ordered after everything in the preceding critical section. Drawing individual edges between the operations of the mutex and the critical section would be tedious, and would break the mutex abstraction. Instead, RMC allows the programmer to draw edges between an operation and all its program-order predecessors or successors using the special quasi-tags pre and post.

3 Visibility

The model recognizes five kinds of actions: reads, writes, pushes (see below), and no-ops. Although the primary interest of the visibility order is writes, it can also be useful to impose visibility among other actions. One reason is that visibility order is transitive, so edges can create new paths even when inconsequential in isolation. For instance, if \( i_1 \xrightarrow{vo} i_2 \xrightarrow{vo} i_3 \), the transitive edge \( i_1 \xrightarrow{vo} i_3 \) might be important even if \( i_1 \xrightarrow{vo} i_2 \) were unimportant in its own right. (This is the only use for no-ops.)

More substantially, consider the following trace. (In this and subsequent traces, all locations are initialized to zero. These writes are omitted from the diagrams to reduce clutter.) The \( Wx1 \xrightarrow{rf} Rx1 \) edge induces visibility order between \( Wx1 \) and \( Rx1 \). Thus, \( Wx1 \) is visibility-ordered before \( Wy2 \) by transitivity. Consequently, \( Rx? \) can see \( Wx1 \), so it cannot read from \( Wx0 \).

\[
\begin{array}{c}
Wx1 \\
\xrightarrow{rf} \\
Rx1 \\
\xrightarrow{vo} \\
Wy2 \\
\xrightarrow{xo} \\
Rx?
\end{array}
\]

Observe that \( Wx1 \) is visibility-ordered before the push, but has no specified relationship to \( Wx3 \), so we do not know immediately that \( Wx1 \) is visible to \( Rx? \). However, \( Wy2 \) takes place after the push, and \( Wy2, Wx3, Rz3 \) and \( Rx? \) happen later still. Therefore, since pushes are globally visible, \( Rx? \) must see the push, and consequently must see \( Wx1 \).

4 Out-of-order and speculative execution

An RMC expression consists of a monadic sequence of actions. In a conventional monadic language, an action’s lifecycle goes through two phases: first, resolve the action’s arguments (purely), then execute the action (generating effects). In RMC we add an intermediate phase. Once an action’s arguments are resolved, we initiate the action, which replaces the action by an action identifier. Later, an initiated action can be executed, at which time the identifier is replaced by the action’s result.

Once an action is initiated, execution may proceed out of order. The semantics may nondeterministically choose to execute the action immediately, or it may delay it and process a later action first.

For example, consider an expression that reads from \( a \) then writes to \( b \). One possible execution executes the write before the read:

\[
\begin{align*}
&x \leftarrow R[a] \in \_ = W[b, 12] \text{ in ret } x \\
&\Rightarrow x \leftarrow i_1\_\_ \leftarrow W[b, 12] \text{ in ret } x \\
&\Rightarrow x \leftarrow i_1\_\_ \leftarrow i_2 \text{ in ret } x \\
&\Rightarrow x \leftarrow i_1\_\_ \leftarrow \text{ret } (\text{ with write effect}) \\
&\Rightarrow x \leftarrow i_1 \text{ in ret } x \\
&\Rightarrow x \leftarrow \text{ret } 10 \text{ in ret } x \\
&\Rightarrow \text{ret } 10 \text{ (with read effect)}
\end{align*}
\]
The purpose of the initiation phase is to require the execution to commit to the arguments of earlier actions before executing later ones. This is important because program-order earlier actions often affect the semantics of later ones, even when the later actions are actually executed first.

Out-of-order execution makes it possible for execution to proceed out of program order, but not out of dependency order. For example, suppose the write to $b$ depends on the value read from $a$:

$$
x \leftarrow R[a] \text{ in } \leftarrow W[b, x] \text{ in } \text{ret } x
$$

$$
x \leftarrow i_1 \text{ in } \leftarrow W[b, x] \text{ in } \text{ret } x
$$

$$
??
$$

The write to $b$ cannot initiate because its arguments are not known. However, RMC permits execution to *speculate* on the value of unknown quantities. For example we may speculate that $x$ will eventually become 10, producing the execution:

$$
x \leftarrow i_1 \text{ in } \leftarrow W[y, x] \text{ in } \text{ret } x
$$

$$
10 \leftarrow i_1 \text{ in } \leftarrow W[y, 10] \text{ in } \text{ret } 10 \quad \text{(NB!)}
$$

$$
10 \leftarrow i_1 \text{ in } \leftarrow i_2 \text{ in } \text{ret } 10
$$

$$
.. .
$$

$$
10 \leftarrow i_1 \text{ in } \text{ret } 10
$$

$$
10 \leftarrow \text{ret } 10 \text{ in } \text{ret } 10 \quad \text{(with read effect)}
$$

$$
\text{ret } 10
$$

Here, the syntax $v \leftarrow e$ in $e'$ is a distinct syntactic form from the usual monadic bind ($x \leftarrow e$ in $e'$). Unlike the usual form, it binds no variables, and instead indicates the value that the bound expression is required to return. Speculation allows us to step from $x \leftarrow e$ in $e'$ to $v \leftarrow e$ in $v/x$|e'. Eventually, once the expression has been executed, we can discharge the speculation, stepping from $v \leftarrow \text{ret } v$ in $e$ to $e$.

The combination of out-of-order and speculative execution allows us to execute the program in nearly any order, subject only to the constraints imposed by the programmer and those inherent in the actions themselves.

RMC uses nondeterminism in a variety of ways, but speculation is the most aggressive. Unlike our other uses (*e.g.*, interleaving of threads, out-of-order execution, the choice of which read satisfies a write), speculation can take execution down a blind alley. In the above example, we speculated $i_1$ would return 10. But suppose it did not? Then we find ourselves in a state like $10 \leftarrow \text{ret } 11$ in ret 10 from which we can make no further progress.

We do not refer to states like this as “stuck”, since stuck usually connotes an unsafe state. Instead we call such states “moot”, since they arise in executions that cannot actually happen. When reasoning about RMC programs, one may ignore any executions that become moot.

Although the use of speculation above might seem fanciful at the architectural level, it could very easily arise in an optimizing compiler, which might be able to determine statically that $R[a]$ would (or at least could) return 10.

A more common use of speculation by the hardware is to execute past a branch when the discriminating value is not yet known. For example:

$$
\begin{align*}
\text{bool enqueue(buffer } & \ast \text{buf, char ch)} \\
\{ & \text{XEDGE(echeck, write)}; \\
& \text{VEDGE(write, eupdate)}; \\
& \text{unsigned back } = \text{buf} \rightarrow \text{back}; \\
& \text{L(echeck, unsigned front } = \text{buf} \rightarrow \text{front}); \\
& \text{int enqueued } = \text{false}; \\
& \text{if } (\text{ring_increment(back) } \neq \text{front}) \{ \\
& \text{L(write, buf} \rightarrow \text{arr[back] } = \text{ch}); \\
& \text{L(eupdate,} \\
& \text{buf} \rightarrow \text{back } = \text{ring_increment(back)}); \\
& \text{enqueued } = \text{true}; \\
& \} \\
& \text{return enqueued}; \\
\}
\end{align*}
$$

$$
\begin{align*}
\text{int dequeue(buffer } & \ast \text{buf)} \\
\{ & \text{XEDGE(dcheck, read)}; \\
& \text{XEDGE(read, dupdate)}; \\
& \text{unsigned front } = \text{buf} \rightarrow \text{front}; \\
& \text{L(dcheck, unsigned back } = \text{buf} \rightarrow \text{back}); \\
& \text{int ch } = -1; \\
& \text{if } (\text{front } \neq \text{back}) \{ \\
& \text{L(read, ch } = \text{buf} \rightarrow \text{arr[front]}) \\
& \text{L(dupdate,} \\
& \text{buf} \rightarrow \text{front } = \text{ring_increment(front)}); \\
& \} \\
& \text{return ch}; \\
\}
\end{align*}
$$

Figure 1: A ring-buffer

$$
\begin{align*}
x \leftarrow R[a] \text{ in force} & (\text{if } x = 0 \text{ then susp } e \text{ else susp } e') \\
\Rightarrow x \leftarrow i_1 \text{ in force} & (\text{if } x = 0 \text{ then susp } e \text{ else susp } e') \\
\Rightarrow 10 \leftarrow i_1 \text{ in force} & (\text{if } 10 = 0 \text{ then susp } e \text{ else susp } e') \\
\Rightarrow 10 \leftarrow i_1 \text{ in force} & (\text{susp } e') \\
\Rightarrow 10 \leftarrow i_1 \text{ in } & e' \\
\Rightarrow .. .
\end{align*}
$$

5 An example

As a realistic example of code using the RMC memory model, consider the code in Figure 1. This code—adapted from the Linux kernel [12]—implements a ring-buffer, a common data structure that implements an imperative queue with a fixed maximum size. The ring-buffer maintains front and back pointers into an array, and the current contents of the queue are those that lie between the back and front pointers (wrapping around if necessary). Elements are inserted by advancing the back pointer, and removed by advancing the front pointer.

The ring-buffer permits at most one writer at a time, and at most one reader at a time, but allows concurrent, unsynchronized access by a writer and a reader.

Note that the buffer is considered empty; not full, when the front and back coincide. Thus, the buffer is full—
and enqueuing fails—when exactly one empty cell remains. This seemingly minor implementation detail complicates the analysis of the code in an interesting way.

We wish the ring-buffer to possess two important properties: (1) the elements dequeued are the same elements that we enqueued (that is, threads do not read from an array cell without the pertinent write having propagated to that thread), and (2) no enqueue overwrites an element that is still current.

The key lines of code are those tagged `echeck`, `write`, and `eupdate` (in `enqueue`), and `dcheck`, `read`, and `dupdate` (in `dequeue`). (It is not necessary to use disjoint tag variables in different functions; we do so only to simplify the exposition.)

For property (1), consider an enqueue-dequeue pair. We have `write` $\rightarrow$ `eupdate` $\rightarrow$ `dcheck` $\rightarrow$ `read`. It follows that `write` is visible to `read`.

Property (2) is a bit more complicated. Since a full ring-buffer has an empty cell, it requires two consecutive enqueues to risk overwriting a cell. The canonical trace we wish to prevent appears in Figure 2. In it, `read1` reads from `write2`, a “later” write that finds room in the buffer only because of a dequeue operation subsequent to `read1`. Hence, a current entry is overwritten.

This problematic trace is impossible, since `read1` $\rightarrow$ `dupdate` $\rightarrow$ `echeck` $\rightarrow$ `write` $\rightarrow$ `read1`. (Alternatively, `read1` $\rightarrow$ `dupdate` $\rightarrow$ `echeck` $\rightarrow$ `write` $\rightarrow$ `read1`.) In RMC, if $i \rightarrow i'$ then $i$ must be executed earlier than $i'$ (you cannot read from a write that hasn’t yet executed), so it follows that `read1` must execute strictly before itself, which is a contradiction.

Note that this argument relies on `echeck` $\rightarrow$ `write` (or `read` $\rightarrow$ `dupdate`). Merely having, say, `echeck` $\rightarrow$ `write` would be insufficient, since nothing in the code as written gives us `write` $\rightarrow$ `write`.

**Contrasted with C/C++** To implement the ring-buffer in C/C++, one can mark the `eupdate` and `dupdate` operations as “release”, and the `echeck` and `dcheck` operations as “acquire”.

For property (1), in an enqueue-dequeue pair we get `eupdate` synchronizes-with `dcheck`, since `eupdate` reads from `dcheck`. This implies that `write` happens-before `read`.

For property (2), consider the trace in Figure 3. In this trace, `eupdate` synchronizes-with `echeck`, (alternatively, `dupdate` synchronizes-with `echeck`), so `read` happens-before `write`. Thus `read` cannot read from `write`.

These correctness arguments for RMC and C/C++ are broadly similar. In part, this is because the C/C++ ring-buffer code is fairly simple, without any nontrivial release sequences or visible sequences of side-effects [7]. And, in part, this is because the C/C++ implementation uses release/acquire synchronization instead of release/consume (which may be cheaper on certain architectures, such as Power and ARM).

To obtain potentially better performance in C/C++, one may instead mark `echeck` and `dcheck` as “consume” and introduce spurious data dependencies from `echeck` to `write`, and `dcheck` to `read`. This makes the correctness argument more delicate. For property (1), in an enqueue-dequeue pair we get (using the former data dependency) that `eupdate` is dependency-ordered-before `read`, which (using the delicate definition of happens-before) once again ensures that `write` happens-before `read`.

The argument for property (2) is a bit more subtle, because happens-before is not transitive. In the argument for RMC, and for C/C++ with “acquire”, we can construct the illegal path using either `dupdate` $\rightarrow$ `echeck`, or `dupdate` $\rightarrow$ `echeck`. In contrast, with “consume” we can only use the latter. We can observe that `dupdate` is dependency-ordered-before `write`, and that `dupdate` is dependency-ordered-before `write`. However, since happens-before respects program order on the left but

\[\text{Figure 2: Impossible ring-buffer trace}\]

\[\text{Figure 3: Impossible ring-buffer traces in C/C++}\]
types $\tau ::= \text{unit} | \text{nat} | \tau \text{ ref} | \tau \text{ susp} | \tau \rightarrow \tau$

numbers $n ::= 0 | 1 | \cdots$

tags $T ::= \cdots$

locations $\ell ::= \cdots$

identifiers $i ::= \cdots$

threads $p ::= \cdots$

terms $m ::= x | 0 | \ell | n | \text{ifz } m \text{ then } m \text{ else } m$

| $\text{ susp } e | \text{ fun } x:(\tau):m | m m$

values $v ::= x | 0 | \ell | n | \text{ susp } e | \text{ fun } x:(\tau):m m$

attributes $b ::= \text{ vis } | \text{ exe}$

labels $\varphi ::= t | T | b | \ell | >$

expr's $e ::= \text{ ret } m | x.(\tau) \leftarrow e \in e | v = v \in e$

| $\text{ R}[m] | \text{ W}[m, m] | \text{ RW}[m, m]$

| $\text{ RMW}[m, x.(\tau).m] | \text{ Push } | \text{ Nop } | i$

| $\text{ force } m | \text{ new } t \text{ in } e | \varphi \neq e$

execution states $\xi ::= e | \xi \parallel p : e$

tag sig $\Upsilon ::= e | \Upsilon, T$

loc'n sig $\Phi ::= e | \Phi, t: \tau$

ident sig $\Psi ::= e | \Psi, i: \tau \triangleright p$

contexts $\Gamma ::= e | \Gamma, t: \text{ tag } | \Gamma, x: \tau$

Figure 4: Syntax

not (in general) on the right, only the latter suffices to show read; happens-before write2.

In RMC, the programmer need not concern himself or herself with acquire vs. consume. He or she simply specifies what is needed—such as execution order between echeck and write—and the compiler synthesizes the most efficient code it can to implement the specification.

Conversely, RMC also allows the programmer to make finer distinctions. In C/C++, both code it can to implement the specification.

Dynamic Semantics

Threads and the store communicate by synchronously agreeing on a transaction. The empty transaction {}, which most thread transitions use—indicates there is no store effect. The initiation transaction $\varphi_1, \ldots, \varphi_n \neq i = \alpha$ indicates that the action $\alpha$ is being initiated with labels $\varphi_1, \ldots, \varphi_n$, and is assigned identifier $i$. The execution transaction $i \downarrow v$ indicates that $i$ has been executed and yielded result $v$. The edge transaction $T \triangleright T'$ indicates that $T$ and $T'$ are fresh tags, expressing a $b$ edge.

The key rules of the dynamic semantics of expressions and execution states appear in Figure 7. The dynamic semantics depends on typing because we want the speculation express either a visibility or execution edge, as indicated by the attribute $b$. The labelling form $\varphi \# e$ attaches a label to an expression, which is either a tag or a quasi-tag $b$ and $\triangleright$ (representing $\text{ pre}$ and $\text{ post}$).

Finally, an execution state is an association list, pairing thread identifiers ($p$) with the current expression on that thread.
actions \( \alpha \) ::= \( R[\ell] \mid W[\ell, v] \mid RW[\ell, v] \mid \text{Push} \mid \text{Nop} \)

transactions \( \delta \) ::= \( \emptyset \mid \varphi \neq i = \alpha \mid i \ni v \mid T \xrightarrow{\delta} T \)

events \( \theta \) ::= init(i, p) \mid \text{is}(i, \alpha) \mid \text{label}(\varphi, i) \mid \text{edge}(b, T, T) \mid \text{exec}(i) \mid \text{rf}(i, i) \mid \text{co}(i, i)

histories \( H \) ::= \( \epsilon \mid H, \theta \)

Figure 6: Dynamic Semantics, syntax

rule to speculate only well-typed values. Consequently the dynamic semantics for expressions and execution states depends on a tag and location signature, and (for expressions) on a context. The evaluation step judgement for expressions indicates the transaction on which that step depends, and the judgement for execution states indicates both the transaction and the thread on which the step took place.

In the interest of brevity, we elide the ambient signatures and context in most of the rules, where they are just ignored or passed to premises. We also leave out the rules that just evaluate subterms (except when they are interesting), and we leave out the dynamic semantics of terms, which is standard.

The rules for out-of-order execution (third row) are straightforward. Speculation (fourth row) allows execution to step from \( \nu[x]e \to v = v'[x'/\nu]e \), provided \( v \) and \( v' \) have the same type. Note that variables are considered so, an important instance is \( e \xrightarrow{\nu} (x = i \in [v/\nu]e) \).

The heart of the system is the bottom row. Once the subterms of an action are evaluated (and thus it matches the grammar of actions given in Figure 6), the action initiates, synchronizing on a transaction \( e \xrightarrow{\nu} \alpha = \nu, \) and is replaced by \( i \). As the transaction bubbles up, it collects any labels that the action lay within. Once all the actions within a label have been initiated, the label can be eliminated. Later, the action executes and is replaced by a return of its value.

Read-modify-writes The semantics deals with read-modify-write expressions by speculating them into read-write expressions. Conceptually, the expression \( \text{RW} [m_1, x : T, m_2] \) reads location \( m_1 \) (a \( \tau \) location), substitutes the read value for \( x \) in \( m_2 \), and writes \( m_2 \), all in one atomic operation. To implement this, the dynamic semantics first evaluates the location (this is necessary for type safety\( ^7 \)) to obtain \( \text{RW} [\ell, x : T, m_2] \). Then the semantics speculates that the read will return some value \( \nu \), which it substitutes for \( x \) to obtain a read-write expression. That read-write expression is then wrapped with speculation scaffolding to ensure that the read-write does in fact return \( \nu \):

\[
\begin{align*}
T; \Phi; \Gamma & \vdash v : \tau \\
T; \Phi; \Gamma & \vdash \text{RW} [\ell, x : T, m_2] \xrightarrow{\theta} (x : T \leftarrow \text{RW} [\ell, [\nu/x]m_2]) \text{ in } x = v \text{ in } \text{ret } v
\end{align*}
\]

Thus, while read-writes are actions seen by the store, read-modify-writes are handled exclusively by the thread semantics. Note that while RMC admits read-modify-write expressions in great generality, we do not assume that languages using the RMC memory model provide such full generality; on the contrary, we expect that they will provide a small set of such operations (e.g. fetch-and-add or compare-and-swap) supported by the architecture.

The Store RMC’s store is modelled in spirit after the storage subsystem of Sarkar, \textit{et al.}’s [19] model for Power, with adaptations made for RMC’s generality. As in Sarkar, \textit{et al.} the store is represented, not by a mapping of locations to values, but by a \textit{history} of all the events that have taken place. The syntax of events appears in Figure 6.

Three events pertain to the initiation of actions: \text{init}(i, p) records that \( i \) was initiated on thread \( p \), \text{is}(i, \alpha) records that \( i \) represents action \( \alpha \), and \text{label}(\varphi, i) records that label \( \varphi \) is attached to \( i \). These three events always occur together, but it is technically convenient to treat them as distinct events.

The event \text{edge}(b, T, T') records the allocation of two tags and an edge between them.

Two events pertain to executed actions: \text{exec}(i) records that \( i \) is executed, and \text{rf}(i, \nu) records both that \( i \nu \) is executed and \( v \) read from \( i \).

The final form, \text{co}(i, \nu), adds a coherence-order edge from \( i \) to \( i \nu \). This is not used in the operational semantics, but it is useful in some proofs to have a way to add extraneous coherence edges.

Store transitions take the form \( H \xrightarrow{\delta} H' \), in which the \( \delta \) is a transaction that is shared synchronously with a transition in thread \( p \).

In the store transition rules, we write \( H(\theta) \) to mean the event \( \theta \) appears in \( H \), where \( \theta \) may contain wildcards (*) indicating parts of the event that don’t matter. As usual, we write \( \rightarrow \) for the transitive, and the reflexive, transitive closures of a relation \( \rightarrow \). We write the composition of two relations as \( \rightarrow \). We say that \( \rightarrow \) is \textit{acyclic} if \( \exists x : x \rightarrow \). We can give three store transitions immediately. An empty transaction generates no new events. An edge transition simply records the new edge, provided both tags are distinct and fresh. (We define \text{tagdecl}_{\text{m}}(T) \text{ to mean } H(\text{edge}(s, T, *)) \lor H(\text{edge}(s, *, T))). An initiation transaction records the thread, the action, and any labels that apply, provided the identifier is fresh.

\[
\begin{align*}
H & \xrightarrow{\text{tagdecl}_{\text{m}}(T)} H' \quad \text{(None)} & T \neq T' \quad \text{tagdecl}_{\text{m}}(T') \quad \text{tagdecl}_{\text{m}}(T) \quad \text{Edge} \\
H & \xrightarrow{\text{exec}(i)} H' \quad \text{H(Edge(b, T, T'))} \\
H & \xrightarrow{\text{co}(i, \nu)} H' \quad \text{H(Init(i, *))} \quad \text{H(Init(i, *))}
\end{align*}
\]

The remaining rules require several auxiliary definitions:

- Identifiers are in \textit{program order} if they are initiated in order and on the same thread:

\[
\begin{align*}
i & \xrightarrow{p} i' \quad \text{def} \\
\exists H_1, H_2, p, H & = H_1(\text{init}(i, p), H_2, \text{init}(i', p), H_3)
\end{align*}
\]

The operational semantics ensures that this notion of program order agrees with the actual program, since (recall) no expression can be executed out-of-order until any preceding actions are initiated.
An identifier is marked as executed by either an `exec` or an `rf` event: `exec_H(i) \equiv H(\text{exec}(i)) \vee H(\text{rf}(\cdot, i))`.

- **Trace order** is the order in which identifiers were actually executed: `i \vdash^d_H i' \equiv \exists H, H_1, H_2, H \equiv H_1, H_2 \land exec_H(i) \land \text{exec}_H(i')`. Note that this definition makes executed identifiers trace-earlier than non-yet-executed identifiers.

- **Specified order**, which realizes the tagging discipline, is defined by the rules:

  \[
  \frac{\vdash^o_H i \quad H(\text{init}(T, i)) \quad H(\text{edge}(b, T, T'))}{i \vdash^o_H i'}
  \]

- The key notion of **visibility order** is defined as the union of specified visibility order, reads-from, and push order: `i \vdash^v H i' \equiv i \vdash^o_H i' \lor i \vdash^{\text{read}}_H i' \lor i \vdash^{\text{push}}_H i'`.

Finally, there is the central notion of coherence order (written `i \vdash^c_H i'`), a strict partial order on actions that write to the same location. The coherence order is inferred `ex post` from the events that transpire in the store, in a manner that we discuss in detail in Section 7. For now, the important property is coherence order must always be acyclic.

Note that our `ex post` view of coherence order is in contrast to the `ex ante` view taken by Sarkar, et al. [19]. In our `ex post` view, coherence order is inferred from events that have already occurred; in the `ex ante` view, coherence edges must already exist (introduced nondeterministically and asynchronously) in order for events to occur.

With these definitions in hand, we give the remaining rules:

- `exec_H(i)`
- `read_H(i, \ell)`
- `write_H(i, \ell, v)`

Observe that read actions (reads and read-writes) are treated the same, and non-read actions (writes, pushes, and no-ops) are treated the same. However, their presence in the history can have very different effects on other actions.

Also observe that `READ` has no explicit premise ensuring that the read returns the “right value,” beyond ensuring that `i_w` writes to the same location `i` reads from. The validity of the read is enforced implicitly by the premise
7 Coherence order

The ultimate question to be resolved is which writes are permitted to satisfy a given read. In a sequentially consistent setting, the only write permitted to satisfy a given read is the unique, most-recent write to the read’s location. In relaxed memory setting, the “most-recent write” is no longer unique. Nevertheless, we assume that there exists an order on writes that is respected by all reads. Following Sarkar, et al. [19], we call this the coherence order.

As in Sarkar, et al., the coherence order is a strict partial order that relates only writes to the same location. In an effort to future-proof our calculus, we place only the minimum constraints on coherence order necessary to accomplish three goals: First, single-threaded computations should exhibit the expected behavior. Second, the message passing idiom—expressed via RMC’s visibility and execution order—should work. Third, read-write operations should be atomic in an appropriate sense. These three aims result in three rules defining coherence order, which we explore below.

The first coherence rule states that a read always reads from the most recent of all writes (to its location) that it has seen:

\[
\frac{i \xrightarrow{\text{wpr}} H \quad i \xrightarrow{\text{wpr}} i'}{i \xrightarrow{\text{co}} H \quad i \xrightarrow{\text{co}} i'} \quad (\text{Co-read})
\]

Here we write \(i \xrightarrow{\text{wpr}} i\) (pronounced “\(i\) is write-read-prior to \(i\)”), to say that the read \(i\) has already seen the write \(i\).

(Other writes may also have been seen; the prior writes are the ones we know for certain have been seen.) Since \(i\) reads from \(i'\) instead of \(i\), we infer that \(i'\) is coherence-later than \(i\).

We know that \(i\) has seen \(i\) in one of two ways. First, \(i\) is program-order-earlier than \(i\) (thus ensuring that single-threaded computations work properly). Second, \(i\) is visibility-order-previous to some action that is execution-order-previous to \(i\) (thus ensuring that the message-passing idiom works properly).

\[
\frac{i \to H \quad i' \xrightarrow{\text{wpr}} H \quad \text{writes}_H(i, \ell, v) \quad \text{reads}_H(i', \ell)}{i \xrightarrow{\text{wpr}} H \quad \text{writes}_H(i, \ell, v) \quad \text{reads}_H(i', \ell)} \quad (\text{Wrp-po})
\]

The second coherence rule says that a write is always more recent than all other writes (to its location) it has seen:

\[
\frac{i \xrightarrow{\text{wpr}} i' \quad \text{writes}_H(i, \ell, v) \quad \text{reads}_H(i, \ell)}{i' \xrightarrow{\text{co}} H \quad i'} \quad (\text{Co-write})
\]

Here we write \(i \xrightarrow{\text{wpr}} i'\) (write-write-prior) to say that the write \(i'\) has already seen the write \(i\). This has two rules similar to write-read priority:

\[
\frac{i \xrightarrow{\text{wpr}} i' \quad \text{writes}_H(i, \ell, v) \quad \text{writes}_H(i', \ell, v')}{i \xrightarrow{\text{wpr}} H \quad i' \xrightarrow{\text{co}} H \quad i'} \quad (\text{Wrp-vis})
\]

Another write-write-priority rule says that \(i'\) has seen \(i\), if \(i'\) has seen some read \(i\), that has seen \(i\):

\[
\frac{i \xrightarrow{\text{wpr}} H \quad i_r \quad \text{reads}_H(i, \ell, v) \quad \text{reads}_H(i', \ell, v)}{i \xrightarrow{\text{wpr}} H \quad i'} \quad (\text{Wrp-read})
\]

The third coherence rule says that when an atomic read-write action \(i\) reads from a write \(i\), no other write \(i'\) can come between them in coherence order:

\[
\frac{i \xrightarrow{\text{wpr}} H \quad i \xrightarrow{\text{co}} H \quad H(\{i, \text{RW}[s, s]\}) \quad i \xrightarrow{\text{co}} H \quad i'd \quad i' \neq i'}{i \xrightarrow{\text{co}} H \quad i'} \quad (\text{Co-rw})
\]

The fourth and final coherence rule says that extra coherence edges can be given in the history. This is a technical device for Corollary 9; the operational semantics never introduces any such events.

\[
\frac{H(\text{co}(i, i'))}{i \xrightarrow{\text{co}} H \quad i'} \quad (\text{Co-fiat})
\]

An auxiliary judgement over ambient signatures updates them according to the transaction. The state can make a transition when all three components agree on a transaction.
7.1 Discussion

The definition of coherence order works in tandem with the requirement for coherence order to be acyclic to prevent illegal resolution of reads. For example, suppose \( i_w \xrightarrow{co} i_w' \) and \( i_r \) has seen \( i_w' \). Then \( i_r \) cannot read from \( i_w \). If it did, we would infer that \( i_w \xrightarrow{co} i_w' \), which would introduce a cycle in coherence order.

Write-write conflicts An alert reader may have observed that RMC enforces execution order only for programmer-specified execution-order edges. Execution order does not respect program order even for conflicting memory accesses. This is an unusual design, and it merits a little discussion. Suppose \( i_w \) and \( i_w' \) are writes to the same location, and suppose \( i_w \xrightarrow{w} i_w' \). It does not follow from any principle identified above that \( i_w \) must execute before \( i_w' \). All that is required is that \( i_w \) be coherence-earlier than \( i_w' \), so that any read that sees both must choose the latter.

In fact, existing architectures do this. Both Power and ARM employ write forwarding [19], wherein a write that is not yet eligible to be sent to the storage system can nevertheless satisfy subsequent reads on the same processor. In RMC’s view, such a write was executed as soon as it became eligible to be read from,8 which might well be sooner than some program-order-previous write.

Read-write conflicts Suppose \( i_r \) reads from the same location \( i_w \) writes to, and suppose \( i_w \xrightarrow{w} i_w' \). Again, it does not follow that \( i_w \) must execute after \( i_r \). All that is required is that \( i_r \) ultimately read from a write coherence-earlier than \( i_w \).

In fact, existing architectures do this also! Some ARM processors will sometimes aggressively carry out a write even when preceding reads are not yet complete, and rely on the storage system to ensure that those preceding reads are satisfied by earlier writes. The left-hand trace in Figure 9 can be observed (rarely) on the APQ8060 processor [5, figure 32]. For this trace to occur, the processor must execute \( Rx0 \) before the push, and hence it must execute \( Wx1 \) before \( RY1 \). We call this phenomenon a “leapfrogging write.”

The leapfrogging write phenomenon also illustrates why priority-via-visibility (e.g., WRP-vis) must be defined using execution order, and not merely trace order, as might otherwise seem attractive. If trace order \((\xrightarrow{in})\) were used, RMC would not admit the right-hand trace in Figure 9 because then \( Wx1 \xrightarrow{co} RY1 \), so \( Wx1 \xrightarrow{co} Wx1 \) by co-read, which would contradict \( Wx1 \xrightarrow{co} Wx1 \) (which we have by co-write). This trace, although not yet observed on any processor, is admitted by the revision [18] of the Sarkar, et al. model [19], to account for leapfrogging writes. Thus, our definition of the message-passing idiom that we seek to respect is based only on deliberate ordering using execution order, and not accidental ordering observed by trace order.

Write-read conflicts Suppose \( i_w \) writes to the same location \( i_r \) reads from, and suppose \( i_w \xrightarrow{w} i_r \). Again, it does not

---

8Hence, RMC’s “executed” is not the same thing as Sarkar, et al.’s “committed” [19].
tories, which are histories that can be extended so that every action is executed, without initiating any new actions. Resolving the former example by adding a new write for Rx? to read from would be cheating. The storage system must function on its own; it cannot rely on a thread to break its deadlock.) This leads to a subtlety in type safety (Theorem 3), similar to our treatment of moot states resulting from inaccurate speculation.

8 Implementation

Implementing the RMC memory model on x86 architectures is easy. The total-store-ordering semantics [20] provides that all instructions are executed in program order (or, more precisely, cannot be observed to be executed out of program order), each write is visible either globally or only to its own thread, and writes become globally visible in program order. Consequently, visibility- and execution-order edges compile away to nothing, and pushes can be implemented by mfences.

Power [19] and ARM [5] are more interesting. Pushes are again implemented by a fence (sync on Power, dmb on ARM). Necessary visibility edges—many can be eliminated statically—are realized by a lightweight fence lying between the two actions. On Power, the lightweight fence is lwsync; ARM does not have one, so a full fence (dmb) is required.

Power and ARM do not provide an analogous execution-order fence, but execution order can be enforced using a number of standard devices, including data or address dependencies (possibly spurious), control dependencies to an lwsync (Power) or lsb (ARM) instruction, and control dependencies to writes.

Note that we do not give a direct mapping of RMC constructs onto the instruction set architecture, such as the mappings for C/C++ considered in Batty, et al. [6]. In C/C++, the synchronization code is determined by the memory-order of an action, so such a mapping is meaningful. In contrast, in RMC the synchronization code is determined by the edges connecting two actions, and the resulting code might appear anywhere between the two. Thus, a direct mapping is not meaningful.

For example, a C/C++ store-release is implemented on Power by lwsync; st. Peephole optimization might improve this, but not much. In RMC, a visibility edge $i \rightarrow i'$ can be realized by a sync or lwsync appearing anywhere between $i$ and $i'$. Indeed, multiple visibility edges might well be realized by the same instruction.

The cost of minimality As always, one can sometimes obtain better performance with hand-crafted assembly language than with compiled code.

Consider the classic ISA2 litmus test [19, 5] in Figure 11. This trace is possible in RMC. The edge $Ry2 \rightarrow Wz3$ breaks the chain of visibility order between $Wx1$ and $Rz3$, so we cannot show that $Wx1$ is prior to $Rz0$.

However, when compiled into Power, using an lwsync for $\rightarrow$, and a dependency for $\rightarrow$, this trace is forbidden. Power guarantees $B$-cumulativity [13, 19, 5], which says that accepting a memory barrier from another thread has the same effect as generating it on your own thread. For $Ry2$ to happen on Power, the second thread must accept the lwsync barrier and the $Wx1$ before it, thus placing a barrier between $Wx1$ and $Wz3$. In our terminology we would say $B$-cumulativity results in $Wx1 \rightarrow Wz3$, and causes $Wx1$ to be prior to $Rz0$.

To prevent this, the programmer would need to insert a push between $Wx1$ and $Wx2$, or upgrade the $Ry2 \rightarrow Wx2$ edge to visibility order. Either way, some additional overhead might be incurred.

As another example, consider the $2+2W$ litmus test [19, 5] in Figure 12. Nothing in RMC prevents this trace from taking place. However, when compiled into Power, using an lwsync for $\rightarrow$, this trace is forbidden. Power guarantees [19, 5] the acyclicity of a relation that in RMC we would write $\rightarrow \cup \rightarrow$, which is plainly violated by the trace. RMC makes no such guarantee.

To prevent this, the programmer would need to insert pushes between both pairs of writes. Again, some additional overhead might be incurred.
Both traces are also possible when translated into C/C++, so C/C++ shares these potential overheads. In RMC we could avoid them by basing our design on stronger assumptions, but we prefer to make the weakest possible assumptions. Thus, some possible additional overhead in cases such as these can be seen as the cost of our minimal assumptions.

9 Theorems

Type safety The first property we establish for RMC is type safety. RMC’s typing judgement for top-level states is \( \vdash (\Sigma, H, \xi) \text{ ok} \) with the one rule:

\[
\begin{align*}
\vdash H : \Sigma & \quad \Sigma \vdash \xi \text{ ok} \\
\vdash (\Sigma, H, \xi) \text{ ok}
\end{align*}
\]

The auxiliary judgement \( \vdash H : \Sigma \) states that \( H \) is trace coherent (which basically means it could be generated by RMC’s dynamic semantics), and \( H \)’s contents match \( \Sigma \). The details appear in Appendix A.

Now we can state the preservation theorem:

Theorem 1 (Preservation) If \( s \rightarrow s' \) and \( \vdash s \text{ ok} \) then \( \vdash s' \text{ ok} \).

The progress theorem is trickier. The standard formulation of progress—well-formed states either are final or take a step—is inappropriate for RMC because of nondeterminism. The goal of type safety is to show that bad states are not reachable from well-formed programs, and the standard formulation identifies bad states with “stuck” states. This is very convenient, provided we can design our operational semantics so that good states have at least one transition and bad ones have no transitions. In a deterministic setting, this is usually feasible.

But for RMC it is both too weak and too strong. Suppose our state has two threads, running \( e \) and \( e' \), and suppose further that \( e \) is in a bad state. Even if \( e \) is stuck, the state as a whole may not be, because we might still be able to take a step on \( e' \). In other words, “stuckness” fails to capture “badness” when we can nondeterministically choose which thread to execute.

Indeed, for RMC the problem is even more pronounced, because no expression can ever be stuck. It is always possible to take a step using the speculation rule.

Conversely, “unstuckness” is also too strong a condition (or would be, if the speculation rule did not make it trivial). We call a state most if it contains an unresolvable speculation \( (v = v' \text{ in } e, \text{ where } v \text{ and } v' \text{ are closed but not equal}) \), or if it is semantically deadlocked. A moot state might well be stuck (except for gratuitous speculation), if it had no work left to do other than undischARGEable speculations or deadlocked actions, but it is not a bad state. It is simply a state that resulted from nondeterministically exploring a blind alley.

Instead, we characterize the good states directly with a judgement \( s \text{ right} \). The details can be found in Appendix D. Then, instead of “unstuckness”, we prove:

Theorem 2 (Rightness) If \( \vdash s \text{ ok} \) then \( s \text{ right} \).

This is similar to Wright and Felleisen’s original technique [21], except they characterized the faulty states instead of the good ones, and for them faultiness was a technical device, not the notion of primary interest.

By itself this is a little unsatisfying because we might have made a mistake in the definition of rightness. To ameliorate this, we anchor rightness back to the operational semantics:

Theorem 3 (Progress) If \( s \) is right, then either (1) for every thread \( p : e \) in \( s \), \( e \) is final, or (2) there exists \( s' \) such that \( s \rightarrow s' \) without using the speculation rule, or (3) \( s \) is moot.

Sequential consistency We also prove two results that establish sequential consistency for different programming disciplines. Our first shows that the programmer can achieve sequential consistency by interleaving actions with pushes.

The proof follows the general lines of Batty, et al.’s [8] sequential consistency proof, but it is generalized to account for the looseness of RMC (e.g., leapfrogging writes) and our main lemma is simpler. We begin with three definitions: Specified sequential consistency indicates that the two actions are separated by a push.

An from-read edge between \( i \) and \( i' \) means that \( i \) reads from a write that is coherence-earlier than \( i' \). Sequentially consistent order [2] is the union of \( \rightarrow, \rightarrow^*, \rightarrow^+ \), and \( \rightarrow \).

It is an important property of communication order that it relates only accesses to the same location, and it agrees with coherence order on writes.

Lemma 4 (Main Lemma) Suppose \( H \) is trace coherent, complete (that is, every identifier in \( H \) is executed), and coherence acyclic (that is, acyclic(\( \Sigma^\rightarrow_H \))). If \( i_1 \rightarrow H i_2 \text{ cop}^+ \)

\( i_3 \rightarrow H i_4 \), where \( i_1 \) and \( i_4 \) are pushes, then \( i_1 \rightarrow H i_4 \).

Proof We may assume, without loss of generality, that \( i_3 \) is a write: If \( i_3 \) is a push or no-op, then \( i_2 = i_3 \) (because \( \rightarrow \) relates only reads and writes) and the result follows immediately. If \( i_3 \) is a read, then it reads from some write \( i' \), and we can rearrange to obtain \( i_1 \rightarrow H i_2 \text{ cop}^+ \rightarrow H i_3 \rightarrow H i_4 \text{ (since } i_3 \rightarrow H i_3 \).

Since \( H \) is complete, \( i_1 \) and \( i_4 \) are executed, so either \( i_1 \rightarrow H i_4 \text{ or } i_4 = i_1 \text{ or } i_4 \rightarrow H i_1 \). In the first case we are done. In both of the remaining cases we have \( i_4 \rightarrow H i_1 \), either trivially or using push order. Therefore \( i_1 \rightarrow H i_1 \).

From this we draw a contradiction.

Suppose \( i_2 \) is a write. Then \( i_2 \rightarrow H i_3 \) (since \( i_2 \rightarrow H i_3 \)). However, we also have \( i_3 \rightarrow H i_2 \), so \( i_3 \rightarrow H i_2 \), which is a contradiction.

On the other hand, suppose \( i_2 \) is a read. Let \( i_w \rightarrow H i_2 \).

Then \( i_w \rightarrow H i_3 \), so \( i_w \rightarrow H i_3 \). However, \( i_3 \rightarrow H i_2 \) so \( i_3 \rightarrow H i_w \), which is a contradiction. □

\(^\text{In practice the execution-order edge from } i_p \text{ to } i' \text{ is ordinarily provided by labelling the push with } i'.\)
Corollary 5 Suppose \( H \) is trace coherent, complete, and coherence acyclic. If \( i_1 \xrightarrow{sq} H i_2 \xrightarrow{sq} H i_3 \xrightarrow{sq} H i_4 \), where \( i_1 \) and \( i_4 \) are pushes, then \( i_1 \rightarrow_H i_4 \).

Proof By induction on the number of \( sq \) steps in \( i_2 \xrightarrow{sq} H i_3 \).

Theorem 6 Suppose \( H \) is trace coherent, complete, and coherence acyclic. Then \( \equiv_H^C \) is acyclic.

Proof Suppose \( i \xrightarrow{sq} H i' \). We show there must be at least one \( sq \) edge in the cycle. Suppose the contrary. Then \( i \rightarrow_H i' \).

If \( i \) is a read, then \( i \xrightarrow{comp} H i', \) which is a contradiction. If \( i \) is a write, then \( i \xrightarrow{comp} H i', \) so \( i \xrightarrow{sq} H i' \), which is a contradiction.

Thus suppose that \( i \xrightarrow{sq} H i_2 \xrightarrow{sq} H i_3 \xrightarrow{sq} H i_4 \) are conflicting \( sq \) edges in the cycle. Suppose the contrary. Then \( i \xrightarrow{comp} H i' \), expanding \( sq \) we obtain \( i \xrightarrow{sq} H i_1 \xrightarrow{sq} H i_2 \xrightarrow{sq} H i_3 \xrightarrow{sq} H i_4 \) for a push \( i_p \). By Corollary 5, \( i_p \rightarrow_H i_p \), which is a contradiction. □

Definition 7 A history \( H \) is consistent if there exists \( H' \) containing no \( sq \) events, such that \( H, H' \) is trace coherent, complete, and coherence acyclic.

Definition 8 Suppose \( R \) is a binary relation over identifiers and \( S \) is a set of identifiers. Then \( R \) is a sequentially consistent ordering for \( S \) if (1) \( R \) is a total order, (2) \( R \) respects program order for elements of \( S \), and (3) every read in the domain of \( R \) is satisfied by the most \( R \)-recent write to its location.

Corollary 9 (Sequential consistency) Suppose \( H \) is consistent. Suppose further that \( S \) is a set of identifiers such that for all \( i, i' \in S \), \( i \xrightarrow{sq} H i' \) implies \( i \xrightarrow{sq} H i' \). Then there exists a sequentially consistent ordering for \( S \).

Proof Let \( H, H' \) be trace coherent, complete, and coherence acyclic. Let \( H'' \) extend \( H, H'' \) by making enough coherence commitments to totally order all writes to the same location. (The Co-tow rule makes this tricky, since we must join read-write chains only at the ends, but it’s not hard to show it can be done.) Then \( H'' \) too is trace coherent, complete, and coherence acyclic.

By Theorem 6, \( \equiv_{H''} \) is acyclic, so there exists a total order containing it. Let \( R \) be such an order. Following Algavle [2], we show that \( R \) is a sequentially consistent order for \( S \):

- Suppose \( i, i' \in S \) and \( i \xrightarrow{po} H i' \). By assumption, \( i \xrightarrow{sq} H i' \). Since \( H'' \) extends \( H \), we have \( i \xrightarrow{sq} H'' i' \), so \( i \xrightarrow{sq} H'' i' \), so \( i \xrightarrow{sq} H'' i' \).

- Suppose \( i_w \xrightarrow{df} H i_w \). Then \( i \xrightarrow{sq} H i'' \), so \( i \xrightarrow{sq} H'' i'' \), and \( i \xrightarrow{sq} H'' i'' \). Let \( \ell \) be the location that \( i_w \) writes and \( i_w \) reads. It remains to show there is no write \( i'' \) to \( \ell \) that such \( i_w \) reads. By \( i \), \( R \) reads \( i_w \).

- Suppose \( i_w \xrightarrow{sq} H i_w \). Then \( i \xrightarrow{sq} H i_w \), so \( i \xrightarrow{sq} H'' i'' \), and \( i \xrightarrow{sq} H'' i'' \). If \( \ell \) is a write to \( \ell \) then \( \ell \xrightarrow{sq} H'' i'' \). If \( \ell \xrightarrow{sq} H'' i'' \), then \( i \xrightarrow{sq} H'' i'' \), and \( i \xrightarrow{sq} H'' i'' \). □

Observe that the sequentially consistent order includes all of \( H \)'s writes, not only those belonging to \( S \). Thus, writes not belonging to \( S \) are adopted into the sequentially consistent order. This means that the members of \( S \) have a consistent view of the order in which all writes took place, not just the writes belonging to \( S \). However, for non-members that order might not agree with program order. (The order contains non-\( S \) reads too, but for purposes of reasoning about \( S \) we can ignore them. Only the writes might satisfy a read in \( S \).)

Our second is a standard result establishing sequential consistency for data-race-free executions. We define synchronized-before as follows:

\[
i \xrightarrow{sb} H i' \quad \text{def} \quad \exists i_w, i \xrightarrow{po} H i_w \xrightarrow{w} H (\text{label}(\text{write}, i_w))
\]

In essence \( i \xrightarrow{sb} i' \) says that \( i \) is visible to some synchronization operation \( i_w \), that is execution-order before \( i' \). That \( i_w \) must be labelled \( sb \) so that \( i \) will also be synchronized before any program-order successors of \( i' \). Note that this definition is insensitive to the mechanism of synchronization: it might use an atomic read-write, the Bakery algorithm, or any other.

We say that an execution is data-race-free if any two conflicting actions on different threads are ordered by synchronized-before. Boehm and Adve [10] refer to this sort of data race as a “type 2 data race.” We may then show that data-race-free executions are sequentially consistent:

Theorem 10 Suppose \( H \) is trace coherent, coherence acyclic, and data-race-free. Let \( \equiv_{\text{scdef}} \) be the union of \( \equiv_{\text{po}} \), \( \equiv_{\text{df}} \), \( \equiv_{\text{sb}} \), \( \equiv_H \), \( \equiv_r \), and \( \equiv_H \). Then \( \equiv_{\text{scdef}} \) is acyclic.

Proof Let \( \equiv_{\text{scdef}} \) be the union of \( \equiv_{\text{po}} \) and \( \equiv_{\text{df}} \). We claim that \( \equiv_{\text{scdef}} \) is acyclic and contains \( \equiv_H \).

Suppose \( i \xrightarrow{sq} H i \). Then there must be an \( sq \) edge in the cycle, otherwise \( i \rightarrow_H i \), which is impossible. Let \( i_w \) be the synchronization operation in one of the \( sq \) edges. We may prove that \( i_w \rightarrow_H i \) by induction on the number of \( sq \) edges in the cycle, which is a contradiction.

It remains to show that \( \equiv_{\text{scdef}} \) is contained in \( \equiv_{\text{scdef}} \). Clearly \( \equiv_{\text{scdef}} \) edges are in \( \equiv_{\text{scdef}} \). Suppose \( i_w \rightarrow_{\text{scdef}} i_w \). Then \( i_w \) and \( i_w \) are conflicting actions. If they are on the same thread, they are related by program order, and if not they are related by synchronized-before. If they are related in the correct direction we are done, so suppose \( i_w \rightarrow_{\text{scdef}} i_w \rightarrow_{\text{scdef}} i_w \). Either way, \( i_w \rightarrow_{\text{scdef}} i_w \), so \( i_w \rightarrow_{\text{scdef}} i_w \). Thus there is a cycle in coherence order.

Suppose \( i_w \rightarrow_{\text{scdef}} i_w \). Again, \( i_w \) and \( i_w \) are conflicting actions. As before, suppose \( i \rightarrow_{\text{scdef}} i_w \) or \( i \rightarrow_{\text{scdef}} i_w \). In the former case, \( i_w \rightarrow_{\text{scdef}} i_w \rightarrow_{\text{scdef}} i_w \), so \( i \rightarrow_{\text{scdef}} i_w \), so \( i_w \rightarrow_{\text{scdef}} i_w \), giving a cycle in coherence order. In the latter case, synchronized-before implies trace order, so \( i_w \rightarrow_{\text{scdef}} i_w \rightarrow_{\text{scdef}} i_w \), so \( i_w \) executes before itself.

Suppose \( i \rightarrow_{\text{scdef}} i_w \). Again, \( i \) and \( i_w \) are conflicting actions. As before, suppose \( i_w \rightarrow_{\text{scdef}} i \), or \( i_w \rightarrow_{\text{scdef}} i \).
Either way, \( i_w \xrightarrow{wp} H i_r \). Expanding \( \delta \), we obtain \( i_w \xrightarrow{\delta} \) \( H i_r \) and \( i_r \xrightarrow{\text{co}} \xrightarrow{\text{co}} H i_w \). By Co-Read we have \( i_w \xrightarrow{co} H i_r \), so there is a cycle in coherence order. □

**Corollary 11 (Sequential consistency)** Suppose \( H \) is consistent and data-race-free. Then there exists a sequentially consistent ordering for all identifiers in \( H \).

**Proof**

As above, noting that a data-race-free history extended without any new is events is still data-race-free.

These two sequential consistency results can also be combined into a single theorem for executions that are partially data-race-free and also use specified sequential consistency. Details appear in the Coq formalization.

10 Conclusion

The "thin-air" problem As given above, RMC makes no effort to rule out "thin-air reads", in which a write of a speculated value justifies itself, thereby allowing a value to appear which has no cause to exist (out of thin air, so to speak). This could be done by adapting the parallel trace mechanism in Jagadeesan, et al. [14], as follows:

First we break the execute transaction \( (i \downarrow v) \) into two forms: write execution \( (i \downarrow v) \) for writes, and benign execution \( (i \downarrow v) \) for everything else. The important difference is that write execution is not passed up by speculation expressions:

\[
\begin{align*}
\delta & \xrightarrow{\delta} \delta' \\
\delta & \notin \text{not of the form } i \downarrow v \\
\end{align*}
\]

By itself, this prevents speculated values from being written to the store, since speculated values appear only in actions that arise within speculation expressions. We do not want to go so far as to rule out speculated writes entirely. Instead, we allow a benign execution in the thread (which might be speculated) to rendezvous with a write execution, provided there exists an alternative trace in which the write could have taken place without speculation:

\[
\begin{align*}
\Sigma & = (\Upsilon, \Phi, \Psi) \\
\Sigma_{\text{alt}} & = (\Upsilon_{\text{alt}}, \Phi_{\text{alt}}, \Psi_{\text{alt}}) \\
\Sigma' & = (\Upsilon, \Phi, \Psi) \\
\Sigma_{\text{alt}}' & = (\Upsilon_{\text{alt}}, \Phi_{\text{alt}}, \Psi_{\text{alt}}) \\
H & \xrightarrow{\text{co}} H' \\
H_{\text{alt}} & \xrightarrow{\text{co}} H'_{\text{alt}} \\
\end{align*}
\]

However, this is not a complete solution to the problem. It prevents thin-air reads, but at the cost of preventing some desirable compiler optimizations. As Batty and Sewell [9] illustrate, it is very difficult to distinguish between thin-air traces and some desirable traces.

Related work RMC’s store is inspired by the storage subsystem of Sarkar, et al. [19]. However, the RMC store is more general, as it is not intended to capture the behavior of a specific architecture, and the RMC’s thread semantics is entirely different from Sarkar, et al.’s thread subsystem.

Another similar formalism is the axiomatic framework of Alglave, et al. [5] (hereafter “Cats”), which builds on Alglave [2]. Like RMC, Cats is a generic system, not specific to any architecture. RMC is an operational framework and Cats is axiomatic, but this difference is less than it might seem since (as noted in Section 1.1), RMC’s acyclic-coherence requirement gives it an axiomatic flavor. Moreover, Cats also defines a operational system in a similar fashion (transitions may take place provided they violate no axiom), and show it equivalent to the axiomatic system.

The greatest differences between RMC and Cats stem from their differing aims: First, Cats is not intended to serve as a programming language. Second, while RMC aims to be weaker than all real architectures, Cats aims to model them. Thus, Cats includes a variety of parameterized machinery that can be instantiated to obtain the behavior of a specific architecture, machinery that RMC omits.

Cats includes four axioms that relate to RMC’s five assumptions. Their “sc per location” axiom corresponds to our first two assumptions (sequential consistency for single-threaded computations, and an acyclic coherence order), except that unlike RMC, Cats chooses to forbid read-read hazards. (Nevertheless Cats could easily be altered to permit them, and RMC could forbid them with an additional coherence rule.) Their “observation” axiom includes our third assumption (message-passing works), but also includes B-cumulativity (recall Section 8) which RMC does not. Their “propagation” axiom includes our fourth assumption (pushes exist), and defines a propagation order similar to our visibility order. However, the propagation axiom goes further, requiring (in RMC terminology) that \( \psi \cup \psi^c \) be acyclic (recall Section 8), which we do not. We omit Cats’s “no thin air” axiom, while Cats omits our fifth assumption (atomic read-writes exist), although Cats includes much of the machinery necessary to support them.

RMC’s speculation mechanism is inspired by the one suggested in Jagadeesan, et al. [14]. Like RMC, they allow a speculated value to be invented out of whole cloth, provided that speculation is subsequently discharged. However, the details are quite different. In their system, the speculated value is written immediately into the store. This makes speculation an effectful operation, while in RMC it is pure.

Moreover, the means of discharge is quite different. In RMC, speculation is tied to a value, and is discharged when it becomes equal to the speculated value. In Jagadeesan et al. speculation is tied to a location in the store. Discharge occurs when the speculated value is written to that location, but that condition would be trivial if employed naively, since such a write already happened at the moment of speculation. Instead, they maintain a parallel trace without the speculated write, and the speculation is discharged when the write occurs in the parallel trace.

Putting speculation in the thread semantics, as we do, rather than the store semantics, makes for a cleaner formalism, since it separates orthogonal concerns, and since it does not require a parallel-trace mechanism. On the other hand, their approach automatically rules out thin-air reads, which ours does not. As discussed above, doing so in our setting requires the addition of a parallel-trace mechanism reminiscent of theirs.

**Formalization** All the results of this paper are formalized in Coq. The formalization consists of 29 thousand lines of code (including comments and whitespace), and takes 93 seconds to check on a 3.4 GHz Intel Core i7.
Acknowledgements

We gratefully acknowledge Luke Zarko, Susmit Sarkar, and Jade Alglave for their many helpful comments and suggestions.

A Static semantics

\[\begin{array}{c}
\Gamma(x) = \tau \\
\Phi(\ell) = \tau \\
\Gamma(x) = \tau \\
\Phi(\ell) = \tau
\end{array}\]

\[\begin{array}{c}
\Phi = \sigma \\
\Phi = \tau \\
\Phi = \sigma \\
\Phi = \tau
\end{array}\]

\[\begin{array}{c}
\Phi(e) = \tau \\
\Phi(e) = \tau \\
\Phi(e) = \tau \\
\Phi(e) = \tau
\end{array}\]

\[\begin{array}{c}
\Phi(e) = \tau \\
\Phi(e) = \tau \\
\Phi(e) = \tau \\
\Phi(e) = \tau
\end{array}\]

\[\begin{array}{c}
\Phi(e) = \tau \\
\Phi(e) = \tau \\
\Phi(e) = \tau \\
\Phi(e) = \tau
\end{array}\]
The auxiliary definition initialized_H(ℓ) says that there exists a write to ℓ that is visible to all reads from ℓ:

\[
\text{initialized}_H(\ell) \overset{\text{def}}{=} \exists i_w, i_p, v. \text{writes}_H(i_w, i_p, v) \land H(is(i_p, \text{Push})) \\
\land i_w \overset{\text{trco}}{\to}_H i_p \land \text{exec}_H(i_p) \\
\land \forall i_r. \text{reads}_H(i_r, \ell) \supset i_p \overset{\text{trco}}{\to}_H i_r
\]

\[\vdash \Psi \text{ and } \exists \alpha. H(\text{init}(i, p)) \land H(is(i, \alpha)) \\
\land \forall \iota. \exists \alpha : \tau \land \neg \text{exec}_H(I)
\]

\[\vdash \epsilon \text{ and } \exists \alpha. H(\text{init}(i, p)) \land H(is(i, \alpha)) \\
\land \forall \iota. \exists \alpha : \tau \land \neg \text{exec}_H(I)
\]

\[\vdash \tau \text{ and } \exists \alpha. H(\text{init}(i, p)) \land H(is(i, \alpha)) \\
\land \forall \iota. \exists \alpha : \tau \land \neg \text{exec}_H(I)
\]

C Dynamic semantics

\[
m \mapsto m'
\]

\[\text{ifz } m_1 \text{ then } m_2 \text{ else } m_3 \mapsto \text{ifz } m'_1 \text{ then } m_2 \text{ else } m_3
\]

\[\vdash \text{exec}(\Sigma_H) \\
\vdash H : \Sigma \\
\vdash \Sigma, H, \xi \text{ ok}
\]

B Trace coherence

\[\vdash e \text{ trco}
\]

\[\vdash \text{writes}_H(i_w, \ell, v) \land \text{exec}_H(i_w) \\
\land \forall i_r. \exists \alpha : \tau \land \neg \text{exec}_H(i_r)
\]

\[\vdash \text{reads}_H(i_r, \ell) \supset \text{exec}_H(i_r)
\]

\[\vdash H \neq T' \text{ and tagdecl}_H(T) \text{ and } \neg \text{tagdecl}_H(T')
\]

17
Γ ⊢ φ right_H

\[
\begin{align*}
\Gamma \vdash t \rightarrow_H & \quad \text{tagdec}_H(T) \quad \text{Γ} \vdash T \rightarrow_H \\
\Gamma \vdash b \rightarrow_H & \quad \Gamma \vdash B \rightarrow_H
\end{align*}
\]

A read i from ℓ is right if there is at least one write to ℓ that is prior to i. This is a subtle but important safety condition, to rule out reading an ill-formed stale value from the cache. When the processor goes to read ℓ it will certainly find something. If there exists at least one visible write, then some value known to the operational semantics has gone to the thread. However, if there exists no visible write at all, the processor might find an old junk value in its cache.

\begin{align*}
\Sigma = (T, \Phi, \Psi) & \quad H \text{trco} \quad \Gamma \vdash \xi \rightarrow_H \\
\forall i. H(\text{init}(i, p)) & \supset \neg \text{exec}_H(i) \supset \xi \rightarrow_H \forall i. H(\text{init}(i, p)) \supset \neg \text{exec}_H(i) \supset \xi \rightarrow_H
\end{align*}

The notation ids(e) represents the set of identifiers of e:

\[
\begin{align*}
\text{ids}(i) & \overset{\text{def}}{=} \{i\} \\
\text{ids}(x : τ \leftarrow e_1 \text{ in } e_2) & \overset{\text{def}}{=} \text{ids}(e_1) \cup \text{ids}(e_2) \\
\text{ids}(v = v' \text{ in } e) & \overset{\text{def}}{=} \text{ids}(e) \\
\text{ids}(\phi \neq e) & \overset{\text{def}}{=} \text{ids}(e) \\
\text{ids}(e) & \overset{\text{def}}{=} \emptyset \quad (\text{for } e \text{ not as above})
\end{align*}
\]

The notation FV(v) represents the set of free variables:

\[
\begin{align*}
\Gamma \vdash m \rightarrow \quad \gamma \vdash \text{FV}(m) \subset \text{Dom}(\Gamma) \\
\Gamma \vdash v_1 = v_2 \in e \rightarrow \quad \gamma \vdash v_1 = v_2 \in e \rightarrow
\end{align*}
\]

References


