A Calculus for Relaxed Memory

Karl Crary Michael J. Sullivan
Carnegie Mellon University
crary@cs.cmu.edu mjsulliv@cs.cmu.edu

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.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

Keywords Relaxed memory concurrency

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 [17]. 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 [15] and ARM [13]) it is more useful to start from a view of memory as simply a pool of available writes [5, 22] (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, 22], 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 [3–5, 22, 23], but for portable, imperative programming languages the memory models are either insufficiently expressive or quite complex:

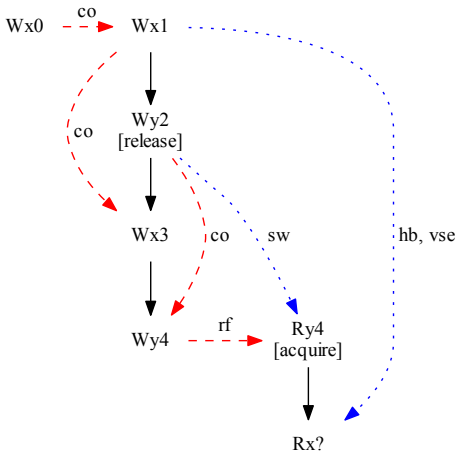
In Java [18] and C/C++ [7, 8, 11] the memory model is divided into two parts: a simple mechanism for data-race-free programming, and a less simple one for lock-free data structures and for low-level implementation of synchronization utilities. Like much of the work on architecture models, our primary concern in this paper is the second aspect; our approach to data-race-free programs—discussed in Section 9—is conventional.

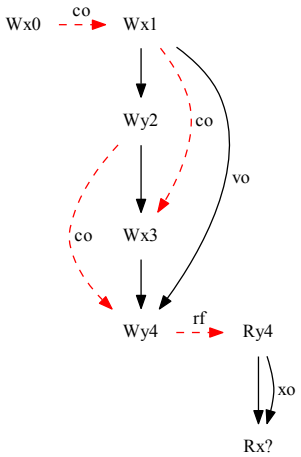
Java's `volatile` variables and C/C++'s `atomic` variables need not be used within a mutex, and thus are suitable for lock-free data structures and synchronization implementation. Java's volatiles are guaranteed to be sequentially consistent; this provides a convenient programming model, but it also 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:

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676984





The unlabelled edges are the program order. The dashed red edges are witness relations[1] derived directly from the trace: Ry4 reads from ($\xrightarrow{rf}$) Wy4. The coherence order ($\xrightarrow{co}$, the global ordering among writes to the same location) on x is Wx0, Wx1, Wx3, and on y is Wy2, Wy4. From these edges[2] 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.

[1] in the terminology of Batty, *et al.* [7]

[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.

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* ($\xrightarrow{vo}$) and *execution-order* ($\xrightarrow{xo}$) 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. (4) There exists a push mechanism (this corresponds to Power's `sync`, ARM's `dmb`, and x86's `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 those 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, *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 [5, 22, 23]. 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, *et al.*'s model for Power [22], and Sewell, *et al.*'s model for x86 [23], 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 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, *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: (1) A type safety result (including a novel adaptation of progress to nondeterministic execution). (2) A sequential-consistency theorem showing that sequential consistency can be obtained by interleaving instructions with memory barriers. This theorem generalizes a theorem of Batty, *et al.* [9] for the more permissive semantics of RMC, and also streamlines the proof. (3) A second sequential-consistency theorem showing that data-race-free programs 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:

`www.cs.cmu.edu/~crary/papers/2014/rmc.tgz`

A version of this paper with a full appendix [12] containing all the inference rules may be found at:

`www.cs.cmu.edu/~crary/papers/2014/rmc.pdf`

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:

```
VEDGE(before, after);
L(before, x = 1);
y = 2;
x = 3;
L(after, y = 4);
```

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

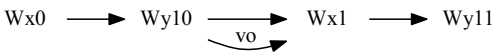
Visibility order implies execution order, because it makes little sense to say that i should be visible to anyone who can see i', if i' can be seen before i even takes place. We can get execution order alone using the `XEDGE` keyword.

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:

```
VEDGE(before, after);
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:

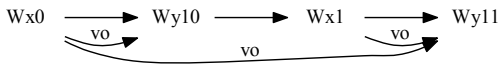


We do not generate nonsensical edges upstream from Wy10 to Wx0, or from Wy11 to Wx1.

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:

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

generates:



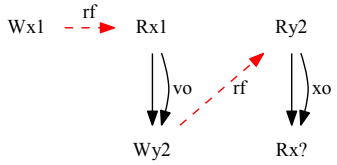
Note the long edge from Wx0 to Wy11. This long edge is desired, as we will see in an example in Section 5.

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`.

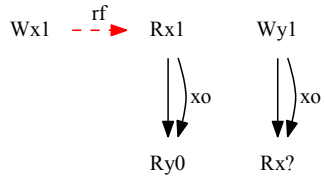
3. Visibility

The model recognizes five kinds of actions: reads, writes, read-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 $i_1 \overset{\text{vo}}{\to} i_2 \overset{\text{vo}}{\to} i_3$, the transitive edge $i_1 \overset{\text{vo}}{\to}{}^{+} i_3$ might be important even if $i_1 \overset{\text{vo}}{\to} i_2$ were unimportant in its own right. (This is the only use for no-ops.)

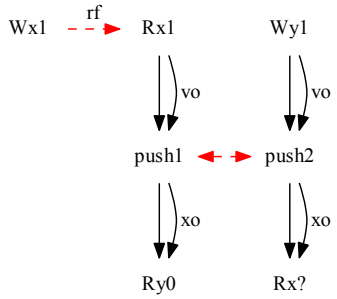
More substantially, consider the following trace. (In this and subsequent traces, all locations are initialized to zero. These writes are omitted from the diagrams to reduce clutter.) The $\mathrm{Wx1} \xrightarrow{\mathrm{rf}} \mathrm{Rx1}$ edge induces visibility order between Wx1 and Rx1. Thus, Wx1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



Pushes perform a global synchronization, like `sync` on Power, `dmb` on ARM, or `mfence` on x86. Once a push is executed, it is visible to all threads. Consider the following trace, adapted from Boehm and Adve's [11] read-to-write causality example.

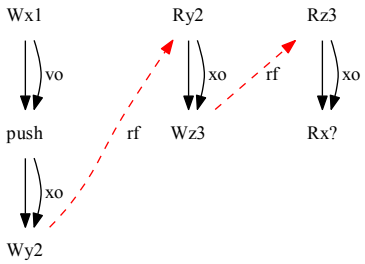


Rx? can read 0, because nothing forces Wx1 to be visible to Rx?. Changing the xo edges to vo has no effect, because visibility-order edges to reads have no more effect than execution order (except possibly to create other paths by transitivity, which would not happen here). However, introducing pushes does change matters:



Now Rx? must read 1. Pushes are totally ordered (a consequence of being globally visible), so either push1 is visible to push2 or vice versa. If the former, then Wx1 is visible to Rx?. If the latter, then the trace is impossible because Wy1 is visible to Ry0. This reasoning is generalized in the Sequential Consistency theorem (Section 9).

For another example of reasoning using pushes, consider:



Observe that Wx1 is visibility-ordered before the push, but has no specified relationship to Wz3, so we do not know immediately that Wx1 is visible to Rx?. However, Wy2 takes place after the push, and Ry2, Wz3, 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:

$$
\begin{array}{lll}
 & x \leftarrow \mathsf{R}[a] \text{ in } \_ \leftarrow \mathsf{W}[b, 12] \text{ in ret } x & \\
\mapsto & x \leftarrow i_1 \text{ in } \_ \leftarrow \mathsf{W}[b, 12] \text{ in ret } x & \\
\mapsto & x \leftarrow i_1 \text{ in } \_ \leftarrow i_2 \text{ in ret } x & \\
\mapsto & x \leftarrow i_1 \text{ in } \_ \leftarrow \text{ret } () \text{ in ret } x & \text{(with write effect)} \\
\mapsto & x \leftarrow i_1 \text{ in ret } x & \\
\mapsto & x \leftarrow \text{ret } 10 \text{ in ret } x & \text{(with read effect)} \\
\mapsto & \text{ret } 10 &
\end{array}
$$

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:

$$
\begin{array}{ll}
 & x \leftarrow \mathsf{R}[a] \text{ in } \_ \leftarrow \mathsf{W}[b, x] \text{ in ret } x \\
\mapsto & x \leftarrow i_1 \text{ in } \_ \leftarrow \mathsf{W}[b, x] \text{ in ret } x \\
\mapsto & ??
\end{array}
$$

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:

$$
\begin{array}{lll}
 & x \leftarrow i_1 \text{ in } \_ \leftarrow \mathsf{W}[y, x] \text{ in ret } x & \\
\mapsto & 10 \leftarrow i_1 \text{ in } \_ \leftarrow \mathsf{W}[y, 10] \text{ in ret } 10 & \text{(NB!)} \\
\mapsto & 10 \leftarrow i_1 \text{ in } \_ \leftarrow i_2 \text{ in ret } 10 & \\
\mapsto & \cdots & \\
\mapsto & 10 \leftarrow i_1 \text{ in ret } 10 & \\
\mapsto & 10 \leftarrow \text{ret } 10 \text{ in ret } 10 & \text{(with read effect)} \\
\mapsto & \text{ret } 10 &
\end{array}
$$

Here, the syntax $v \leftarrow e \text{ in } e'$ is a distinct syntactic form[3] from the usual monadic bind ($x \leftarrow e \text{ in } e'$). 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 $x \leftarrow e \text{ in } e'$ to $v \leftarrow e \text{ in } [v/x]e'$. Eventually, once the expression has been executed, we can discharge the speculation, stepping from $v \leftarrow \text{ret } v \text{ in } e$ 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.

[3] In fact, it is actually a derived form, built from a more primitive speculation form in Section 6.

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 $10 \leftarrow \mathsf{ret}\ 11\ \mathsf{in}\ \mathsf{ret}\ 10$ 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 $\mathsf{R}[a]$ 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:[4]

$$
\begin{array}{ll}
 & x \leftarrow \mathsf{R}[a]\ \mathsf{in}\ \mathsf{force}(\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{susp}\, e\ \mathsf{else}\ \mathsf{susp}\, e') \\
\mapsto & x \leftarrow i_1\ \mathsf{in}\ \mathsf{force}(\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{susp}\, e\ \mathsf{else}\ \mathsf{susp}\, e') \\
\mapsto & 10 \leftarrow i_1\ \mathsf{in}\ \mathsf{force}(\mathsf{if}\ 10 = 0\ \mathsf{then}\ \mathsf{susp}\, e\ \mathsf{else}\ \mathsf{susp}\, e') \\
\mapsto & 10 \leftarrow i_1\ \mathsf{in}\ \mathsf{force}(\mathsf{susp}\, e') \\
\mapsto & 10 \leftarrow i_1\ \mathsf{in}\ e' \\
\mapsto & \cdots
\end{array}
$$

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 [14]—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 enqueueing 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 $\texttt{write} \overset{\text{vo}}{\rightarrow} \texttt{eupdate} \overset{\text{rf}}{\rightarrow} \texttt{dcheck} \overset{\text{xo}}{\rightarrow} \texttt{read}$. 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, $\texttt{read}_1$ reads from $\texttt{write}_2$, a "later" write that finds

[4] Here, susp suspends an expression (forming a thunk), and force forces a suspension.

```
bool enqueue(buffer *buf, char ch)
{
    XEDGE(echeck, write);
    VEDGE(write, eupdate);

    unsigned back = buf->back;
    L(echeck, unsigned front = buf->front);

    int enqueued = false;
    if (ring_increment(back) != front) {
        L(write, buf->arr[back] = ch);
        L(eupdate,
          buf->back = ring_increment(back));
        enqueued = true;
    }
    return enqueued;
}

int dequeue(buffer *buf)
{
    XEDGE(dcheck, read);
    XEDGE(read, dupdate);

    unsigned front = buf->front;
    L(dcheck, unsigned back = buf->back);

    int ch = -1;
    if (front != back) {
        L(read, ch = buf->arr[front]);
        L(dupdate,
          buf->front = ring_increment(front));
    }
    return ch;
}
```

Figure 1. A ring-buffer

room in the buffer only because of a dequeue operation subsequent to $\texttt{read}_1$. Hence, a current entry is overwritten.

This problematic trace is impossible, since $\texttt{read}_1 \overset{\text{xo}}{\rightarrow} \texttt{dupdate}_1 \overset{\text{rf}}{\rightarrow} \texttt{echeck}_1 \overset{\text{xo}}{\rightarrow} \texttt{write}_2 \overset{\text{rf}}{\rightarrow} \texttt{read}_1$. (Alternatively, $\texttt{read}_1 \overset{\text{xo}}{\rightarrow} \texttt{dupdate}_2 \overset{\text{rf}}{\rightarrow} \texttt{echeck}_2 \overset{\text{xo}}{\rightarrow} \texttt{write}_2 \overset{\text{rf}}{\rightarrow} \texttt{read}_1$.) In RMC, if $i \overset{\text{rf}}{\rightarrow} i'$ then i must be executed earlier than i' (you cannot read from a write that hasn't yet executed), so it follows that $\texttt{read}_1$ must execute strictly before itself, which is a contradiction.

Note that this argument relies on $\texttt{echeck}_1 \overset{\text{xo}}{\rightarrow} \texttt{write}_2$ (or $\texttt{read}_1 \overset{\text{xo}}{\rightarrow} \texttt{dupdate}_2$). Merely having, say, $\texttt{echeck}_1 \overset{\text{xo}}{\rightarrow} \texttt{write}_1$ would be insufficient, since nothing in the code as written gives us $\texttt{write}_1 \overset{\text{xo}}{\rightarrow} \texttt{write}_2$.

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`.[5]

For property (2), consider the trace in Figure 3. In this trace, $\texttt{dupdate}_1$ synchronizes-with $\texttt{echeck}_1$ (alternatively, $\texttt{dupdate}_2$ synchronizes-with $\texttt{echeck}_2$), so $\texttt{read}_1$ happens-before $\texttt{write}_2$. Thus $\texttt{read}_1$ cannot read from $\texttt{write}_2$.

[5] The latter inference is a bit subtle, since happens-before is not transitive. It includes program order on the left, but includes it on the right only after synchronizes-with.

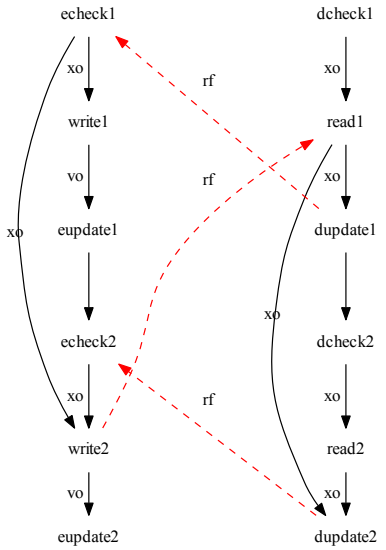


Figure 2. Impossible ring-buffer trace

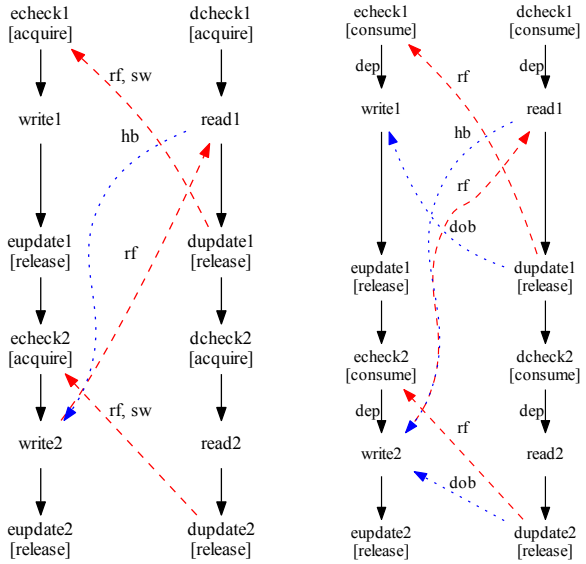


Figure 3. Impossible ring-buffer traces in C/C++

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`.

types	τ	::=	$\mathsf{unit} \mid \mathsf{nat} \mid \tau\ \mathsf{ref} \mid \tau\ \mathsf{susp} \mid \tau \to \tau$
numbers	n	::=	$0 \mid 1 \mid \cdots$
tags	T	::=	$\cdots$
locations	ℓ	::=	$\cdots$
identifiers	i	::=	$\cdots$
threads	p	::=	$\cdots$
terms	m	::=	$x \mid () \mid \ell \mid n \mid \mathsf{ifz}\ m\ \mathsf{then}\ m\ \mathsf{else}\ m$
		$\mid$	$\mathsf{susp}\ e \mid \mathsf{fun}\ x(x{:}\tau){:}\tau.m \mid m\ m$
values	v	::=	$x \mid () \mid \ell \mid n \mid \mathsf{susp}\ e \mid \mathsf{fun}\ x(x{:}\tau){:}\tau.m$
attributes	b	::=	$\mathsf{vis} \mid \mathsf{exe}$
labels	φ	::=	$t \mid T \mid \overset{b}{\triangleleft} \mid \overset{b}{\triangleright}$
expr's	e	::=	$\mathsf{ret}\ m \mid x{:}\tau \leftarrow e\ \mathsf{in}\ e \mid v = v\ \mathsf{in}\ e$
		$\mid$	$\mathsf{R}[m] \mid \mathsf{W}[m,m] \mid \mathsf{RW}[m,m]$
		$\mid$	$\mathsf{RMW}[m, x{:}\tau.m] \mid \mathsf{Push} \mid \mathsf{Nop} \mid i$
		$\mid$	$\mathsf{force}\ m \mid \mathsf{new}\ t \overset{b}{\Rightarrow} t.e \mid \varphi \# e$
execution states	ξ	::=	$\epsilon \mid \xi \parallel p : e$
tag sig	Υ	::=	$\epsilon \mid \Upsilon, T$
loc'n sig	Φ	::=	$\epsilon \mid \Phi, \ell{:}\tau$
ident sig	Ψ	::=	$\epsilon \mid \Psi, i{:}\tau@p$
contexts	Γ	::=	$\epsilon \mid \Gamma, t{:}\mathsf{tag} \mid \Gamma, x{:}\tau$

Figure 4. Syntax

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 $\texttt{dupdate}_1 \xrightarrow{\text{rf}} \texttt{echeck}_1$ or $\texttt{dupdate}_2 \xrightarrow{\text{rf}} \texttt{echeck}_2$. In contrast, with "consume" we can only use the latter. We can observe that $\texttt{dupdate}_1$ is dependency-ordered-before $\texttt{write}_1$, and that $\texttt{dupdate}_2$ is dependency-ordered-before $\texttt{write}_2$. However, since happens-before respects program on the left but not (in general) on the right, only the latter suffices to show $\texttt{read}_1$ happens-before $\texttt{write}_2$.

In RMC, the programmer need not concern him 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 `eupdate` and `dupdate` must be marked "release". This entails the same overhead for each, tantamount to a visibility order specification in RMC. But, as we have seen, visibility order is necessary only in `enqueue`; `dequeue` requires merely the less-expensive execution order.

6. RMC

The syntax of RMC given in Figure 4. Tags (T) and identifiers (i) range over actual dynamically generated tags and identifiers; they do not appear in user programs. We do not discuss the allocation primitives in this paper so memory locations (ℓ) can appear in RMC programs here, but not in real RMC-based programs. There are two sorts of variables: x ranges over values and t ranges over tags.

We make a standard syntactic distinction between pure terms (m) and effectful expressions (e). The terms are standard, and most of the expression forms we have discussed already. The speculation form $v = v'\ \mathsf{in}\ e$ indicates that the values v and v' are speculated to be identical while evaluating e. The monadic speculation form from Section 4 is actually a derived form:

$$v \leftarrow e\ \mathsf{in}\ e' \stackrel{\text{def}}{=} x \leftarrow e\ \mathsf{in}\ (x = v\ \mathsf{in}\ e')$$

$$\frac{\Upsilon;\Phi;\Gamma \vdash m : \tau}{\Upsilon;\Phi;\epsilon;\Gamma \vdash \mathsf{ret}\, m : \tau} \qquad \frac{}{\Upsilon;\Phi;(i{:}\tau@p);\Gamma \vdash i : \tau}$$

$$\frac{\Upsilon;\Phi;\Psi;\Gamma \vdash e_1 : \tau_1 \quad \Upsilon;\Phi;\Psi';(\Gamma, x{:}\tau_1) \vdash e_2 : \tau_2 \quad e_1\ \mathsf{init}}{\Upsilon;\Phi;(\Psi,\Psi');\Gamma \vdash x{:}\tau_1 \leftarrow e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\frac{\Upsilon;\Phi;\Psi;\Gamma \vdash e_1 : \tau_1 \quad \Upsilon;\Phi;\epsilon;(\Gamma, x{:}\tau_1) \vdash e_2 : \tau_2}{\Upsilon;\Phi;\Psi;\Gamma \vdash x{:}\tau_1 \leftarrow e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Figure 5. Static Semantics (abridged)

The tag allocation form $\mathsf{new}\ t \overset{b}{\Rightarrow} t'.e$ generates two new tags bound to variables t and t' in the scope e. Those tags express either a visibility or execution edge, as indicated by the attribute b.[6] The labelling form $\varphi \# e$ attaches a *label* to an expression, which is either a tag or a quasi-tag $\overset{b}{\triangleleft}$ and $\overset{b}{\triangleright}$ (representing `pre` and `post`).

Finally, an execution state is an association list, pairing thread identifiers (p) with the current expression on that thread.

Static Semantics The static semantics of RMC expressions is largely standard. Expressions are typechecked relative to three ambient signatures and a context. The ambient signatures specify the valid tags, locations, and identifiers, giving the types for location and identifiers, and the threads for identifiers. Terms are not passed an identifier signature because identifiers cannot appear within well-formed terms. Execution states are not passed a context, because well-formed execution states must be closed.

The only interesting aspect is in the treatment of the identifier signature, which we use to ensure that each unexecuted identifier appears exactly once, and that the store's view of program order agrees with the code. Thus, on a given thread, the identifier signature is used like a linear, ordered context [19], as shown in Figure 5. There are two rules for bind: Since actions must be initiated in order, identifiers may appear in the second expression of a bind only when the first is fully initiated (*i.e.,* all of its actions have been converted to identifiers). At the top level (not shown, for space reasons), the identifier signature lists unexecuted identifiers in the same order they appear in the store, and the typing rules for execution states filter out the identifiers for other threads.

Dynamic Semantics Threads and the store communicate by synchronously agreeing on a *transaction*. The empty transaction ($\emptyset$)—which most thread transitions use—indicates there is no store effect. The initiation transaction $\varphi_1, \ldots, \varphi_n \# i = \alpha$ indicates that the action α 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 $T \overset{b}{\Rightarrow} T'$ 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 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.

[6] Thus, in RMC, each tag has exactly one partner. From this primitive, the compiler builds up a more flexible mechanism that allows tags to be used in multiple edges, as in the example of Section 5.

actions	α	::=	$\mathsf{R}[\ell] \mid \mathsf{W}[\ell, v] \mid \mathsf{RW}[\ell, v]$
		$\mid$	$\mathsf{Push} \mid \mathsf{Nop}$
transactions	δ	::=	$\emptyset \mid \vec{\varphi} \# i = \alpha \mid i \downarrow v \mid T \overset{b}{\Rightarrow} T$
events	θ	::=	$\mathsf{init}(i,p) \mid \mathsf{is}(i,\alpha) \mid \mathsf{label}(\varphi, i)$
		$\mid$	$\mathsf{edge}(b,T,T) \mid \mathsf{exec}(i) \mid \mathsf{rf}(i,i)$
		$\mid$	$\mathsf{co}(i,i)$
histories	H	::=	$\epsilon \mid H, \theta$

Figure 6. Dynamic Semantics, syntax

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 $[v/x]e$ to $v = v'\ \mathsf{in}\ [v'/x]e$, provided v and v' have the same type. Note that variables are considered values, so an important instance is $e \mapsto (x = v\ \mathsf{in}\ [v/x]e)$.

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 $\epsilon \# i = \alpha$, 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 $\mathsf{RMW}[m_1, x{:}\tau.m_2]$ reads location m_1 (a τ 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 $\mathsf{RMW}[\ell, x{:}\tau.m_2]$. Then the semantics speculates that the read will return some value v, 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 v:

$$\frac{\Upsilon;\Phi;\Gamma \vdash v : \tau}{\Upsilon;\Phi;\Gamma \vdash \mathsf{RMW}[\ell, x{:}\tau.m_2] \overset{\emptyset}{\mapsto} (x{:}\tau \leftarrow \mathsf{RW}[\ell, [v/x]m_2]\ \mathsf{in}\ x = v\ \mathsf{in}\ \mathsf{ret}\, v)}$$

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, *et al.*'s [22] model for Power, with adaptations made for RMC's generality. As in Sarkar, *et al.*, the store is represented, not by a mapping of locations to values, but by a *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: $\mathsf{init}(i,p)$ records that i was initiated on thread p, $\mathsf{is}(i,\alpha)$ records that i rep-

[7] If the location diverges, then there might be no actual location to which values of type τ have been written. Consquently, the type τ might be empty, so no speculation transition would be possible.

$\boxed{\Upsilon; \Phi \vdash \xi \overset{\delta@p}{\longmapsto} \xi'}$

$$\frac{\Upsilon; \Phi; \epsilon \vdash e \overset{\delta}{\mapsto} e'}{\Upsilon; \Phi \vdash (\xi \parallel p : e) \overset{\delta@p}{\longmapsto} (\xi \parallel p : e')} \qquad \frac{\xi \overset{\delta@p'}{\longmapsto} \xi'}{(\xi \parallel p : e) \overset{\delta@p'}{\longmapsto} (\xi' \parallel p : e)}$$

$\boxed{\Upsilon; \Phi; \Gamma \vdash e \overset{\delta}{\mapsto} e' \qquad e\ \mathsf{init} \qquad \varphi \# \delta}$

$$\frac{}{\mathsf{ret}\, m\ \mathsf{init}} \qquad \frac{}{i\ \mathsf{init}} \qquad \frac{e_1\ \mathsf{init} \quad e_2\ \mathsf{init}}{x{:}\tau \leftarrow e_1\ \mathsf{in}\ e_2\ \mathsf{init}} \qquad \frac{e\ \mathsf{init}}{v_1 = v_2\ \mathsf{in}\ e\ \mathsf{init}} \qquad \frac{}{\mathsf{force}(\mathsf{susp}\, e) \overset{\emptyset}{\mapsto} e} \qquad \frac{}{\mathsf{new}\ t \overset{b}{\Rightarrow} t'.e \overset{T \overset{b}{\Rightarrow} T'}{\longmapsto} [T, T'/t, t']e}$$

$$\frac{e_1 \overset{\delta}{\mapsto} e_1'}{(x{:}\tau \leftarrow e_1\ \mathsf{in}\ e_2) \overset{\delta}{\mapsto} (x{:}\tau \leftarrow e_1'\ \mathsf{in}\ e_2)} \qquad \frac{e_1\ \mathsf{init} \quad \Upsilon; \Phi; (\Gamma, x{:}\tau) \vdash e_2 \overset{\delta}{\mapsto} e_2' \quad x \notin \mathsf{FV}(\delta)}{\Upsilon; \Phi; \Gamma \vdash (x{:}\tau \leftarrow e_1\ \mathsf{in}\ e_2) \overset{\delta}{\mapsto} (x{:}\tau \leftarrow e_1\ \mathsf{in}\ e_2')} \qquad \frac{}{(x{:}\tau \leftarrow \mathsf{ret}\, v\ \mathsf{in}\ e) \overset{\emptyset}{\mapsto} [v/x]e}$$

$$\frac{\Upsilon; \Phi; \Gamma \vdash v : \tau \quad \Upsilon; \Phi; \Gamma \vdash v' : \tau}{\Upsilon; \Phi; \Gamma \vdash [v/x]e \overset{\emptyset}{\mapsto} (v = v'\ \mathsf{in}\ [v'/x]e)} \qquad \frac{e \overset{\delta}{\mapsto} e'}{v = v'\ \mathsf{in}\ e \overset{\delta}{\mapsto} v = v'\ \mathsf{in}\ e'} \qquad \frac{}{(v = v\ \mathsf{in}\ e) \overset{\emptyset}{\mapsto} e}$$

$$\frac{}{\alpha \overset{\epsilon \# i = \alpha}{\longmapsto} i} \qquad \frac{e \overset{\delta}{\mapsto} e'}{\varphi \# e \overset{\varphi \# \delta}{\longmapsto} \varphi \# e'} \qquad \frac{e\ \mathsf{init}}{\varphi \# e \overset{\emptyset}{\mapsto} e} \qquad \frac{}{i \overset{i \downarrow v}{\longmapsto} \mathsf{ret}\, v} \qquad \varphi \# \delta = \begin{cases} (\varphi, \vec{\varphi}) \# i = \alpha & \text{if } \delta = (\vec{\varphi} \# i = \alpha) \\ \delta & \text{otherwise} \end{cases}$$

Figure 7. Dynamic Semantics, threads (abridged)

resents action α, and $\mathsf{label}(\varphi, i)$ records that label φ is attached to i. These three events always occur together, but it is technically convenient to treat them as distinct events.

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

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

The final form, $\mathsf{co}(i, i')$, adds a coherence-order edge from i to i'. 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 $H \overset{\delta@p}{\longmapsto} H'$, in which the δ 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 θ appears in H, where θ may contain wildcards ($*$) indicating parts of the event that don't matter. As usual, we write $\rightarrow^+$ and $\rightarrow^*$ for the transitive, and the reflexive, transitive closures of a relation $\rightarrow$. We write the composition of two relations as $\overset{x}{\rightarrow}\overset{y}{\rightarrow}$. We say that $\rightarrow$ is *acyclic* if $\neg\exists x.\, x \rightarrow^+ x$.

We can give three store transitions immediately. An empty transaction generates no new events. An edge transaction simply records the new edge, provided both tags are distinct and fresh. (We define $\mathsf{tagdecl}_H(T)$ to mean $H(\mathsf{edge}(*, T, *)) \vee H(\mathsf{edge}(*, *, T))$.) An initiation transaction records the thread, the action, and any labels that apply, provided the identifier is fresh.

$$\frac{}{H \overset{\emptyset@p}{\longmapsto} H}\ (\textsc{None}) \qquad \frac{T \neq T' \quad \neg\mathsf{tagdecl}_H(T) \quad \neg\mathsf{tagdecl}_H(T')}{H \overset{T \overset{b}{\Rightarrow} T'@p}{\longmapsto} H, \mathsf{edge}(b, T, T')}\ (\textsc{Edge})$$

$$\frac{\neg H(\mathsf{init}(i, *))}{H \overset{\vec{\varphi} \# i = \alpha@p}{\longmapsto} H, \mathsf{init}(i, p), \mathsf{is}(i, \alpha), [\mathsf{label}(\varphi, i) \mid \varphi \in \vec{\varphi}]}\ (\textsc{Init})$$

The remaining rules require several auxiliary definitions:

- Identifiers are in *program order* if they are initiated in order and on the same thread:

  $i \overset{\mathsf{po}}{\rightarrow}_H i' \overset{\text{def}}{=}$
  $\exists H_1, H_2, H_3, p.\ H = H_1, \mathsf{init}(i, p), H_2, \mathsf{init}(i', p), H_3$

  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 identifer is marked as executed by either an exec or an rf event: $\mathsf{exec}_H(i) \overset{\text{def}}{=} H(\mathsf{exec}(i)) \vee H(\mathsf{rf}(*, i))$.
- *Trace order* is the order in which identifiers were actually executed: $i \overset{\mathsf{to}}{\rightarrow}_H i' \overset{\text{def}}{=} \exists H_1, H_2.\ H = H_1, H_2 \wedge \mathsf{exec}_{H_1}(i) \wedge \neg\mathsf{exec}_{H_1}(i')$. Note that this definition makes executed identifiers trace-order-earlier than non-yet-executed identifiers.
- *Specified order,* which realizes the tagging discipline, is defined by the rules:

$$\frac{i \overset{\mathsf{po}}{\rightarrow}_H i' \quad H(\mathsf{label}(T, i)) \quad H(\mathsf{label}(T', i')) \quad H(\mathsf{edge}(b, T, T'))}{i \overset{b}{\Rightarrow}_H i'}$$

$$\frac{i \overset{\mathsf{po}}{\rightarrow}_H i' \quad H(\mathsf{label}(\overset{b}{\triangleleft}, i'))}{i \overset{b}{\Rightarrow}_H i'} \qquad \frac{i \overset{\mathsf{po}}{\rightarrow}_H i' \quad H(\mathsf{label}(\overset{b}{\triangleright}, i))}{i \overset{b}{\Rightarrow}_H i'}$$

- The key notion of *execution order* is defined in terms of specified order: $i \overset{\mathsf{xo}}{\rightarrow}_H i' \overset{\text{def}}{=} \exists b.\, i \overset{b}{\Rightarrow}_H i'$.
- An identifier is *executable* if it has not been executed, and all its execution-order predecessors have been: $\mathsf{executable}_H(i) \overset{\text{def}}{=} \neg\mathsf{exec}_H(i) \wedge \forall i'.\, i' \overset{\mathsf{xo}}{\rightarrow}_H i \supset \mathsf{exec}_H(i')$.
- A read action is either a read or a read-write: $\mathsf{reads}_H(i, \ell) = H(\mathsf{is}(i, \mathsf{R}[\ell])) \vee H(\mathsf{is}(i, \mathsf{RW}[\ell, *]))$.
- Similarly, a write action is either a write or a read-write: $\mathsf{writes}_H(i, \ell, v) = H(\mathsf{is}(i, \mathsf{W}[\ell, v])) \vee H(\mathsf{is}(i, \mathsf{RW}[\ell, v]))$.

- *Reads-from:* $i \xrightarrow{\mathsf{rf}}_H i' \stackrel{\text{def}}{=} H(\mathsf{rf}(i,i'))$.
- Identifiers i and i' are *push-ordered* if i is a push and is trace-order-earlier than i': $i \xrightarrow{\pi\mathsf{o}}_H i' \stackrel{\text{def}}{=} H(\mathsf{is}(i,\mathsf{Push})) \wedge i \xrightarrow{\mathsf{to}}_H i'$. Since pushes are globally visible as soon as they execute, this means that i should be visible to i'.
- The key notion of *visibility order* is defined as the union of specified visibility order, reads-from, and push order: $i \xrightarrow{\mathsf{vo}}_H i' \stackrel{\text{def}}{=} i \xrightarrow{\mathsf{vis}}_H i' \vee i \xrightarrow{\mathsf{rf}}_H i' \vee i \xrightarrow{\pi\mathsf{o}}_H i'$.

Finally, there is the central notion of coherence order (written $i \xrightarrow{\mathsf{co}}_H i'$), 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.* [22]. 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:

$$\frac{\begin{array}{l}H(\mathsf{init}(i,p)) \quad H(\mathsf{is}(i,\alpha)) \quad \alpha = \mathsf{R}[\ell] \vee \alpha = \mathsf{RW}[\ell,v'] \\ \mathsf{executable}_H(i) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H;\mathsf{rf}(i_w,i)}) \\ \mathsf{writes}_H(i_w,\ell,v) \quad \mathsf{exec}_H(i_w)\end{array}}{H \xrightarrow{i\downarrow v@p} H, \mathsf{rf}(i_w,i)} \text{ (READ)}$$

$$\frac{\begin{array}{c}H(\mathsf{init}(i,p)) \quad H(\mathsf{is}(i,\alpha)) \\ \alpha = \mathsf{W}[\ell,v] \vee \alpha = \mathsf{Push} \vee \alpha = \mathsf{Nop} \\ \mathsf{executable}_H(i) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H;\mathsf{exec}(i)})\end{array}}{H \xrightarrow{i\downarrow\circ@p} H, \mathsf{exec}(i)} \text{ (NONREAD)}$$

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 $\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H,\mathsf{rf}(i_w,i)})$. If i_w is an improper write for i to read from—for example, if i_w is coherence-older than some other write visible to i—then adding $\mathsf{rf}(i_w,i)$ to the history will create a cycle in coherence order.

The top level The top-level dynamic semantics appears in Figure 8. RMC's top-level state consists of the three ambient signatures, a history, and an execution state.

$$\begin{array}{lrcl}\text{signature} & \Sigma & ::= & (\Upsilon,\Phi,\Psi) \\ \text{state} & s & ::= & (\Sigma,H,\xi)\end{array}$$

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. 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.* [22], we call this the *coherence order*.

$$\boxed{\Sigma \xrightarrow{\delta@p} \Sigma'} \qquad \frac{}{\Sigma \xrightarrow{\emptyset@p} \Sigma} \qquad \frac{\forall\varphi \in \vec{\varphi}.\ \Upsilon;\epsilon \vdash \varphi : \mathsf{label} \quad \Upsilon,\Phi,\Psi,\epsilon \vdash \alpha : \tau}{(\Upsilon,\Phi,\Psi) \xrightarrow{\vec{\varphi}\#i=\alpha@p} (\Upsilon,\Phi,(\Psi,i{:}\tau@p))}$$

$$\frac{\Psi = \Psi_1, i{:}\tau@p, \Psi_2 \quad \vdash v : \tau}{(\Upsilon,\Phi,\Psi) \xrightarrow{i\downarrow v@p} (\Upsilon,\Phi,(\Psi_1,\Psi_2))} \qquad \frac{\begin{array}{c}\Upsilon' = \Upsilon, T, T' \\ T \neq T' \quad T,T' \notin \Upsilon\end{array}}{(\Upsilon,\Phi,\Psi) \xrightarrow{T \stackrel{b}{\Rightarrow} T'@p} (\Upsilon',\Phi,\Psi)}$$

$$\boxed{s \mapsto s'} \qquad \frac{\begin{array}{c}\Sigma = (\Upsilon,\Phi,\Psi) \\ \Sigma \xrightarrow{\delta@p} \Sigma' \qquad H \xrightarrow{\delta@p} H' \qquad \Upsilon;\Phi \vdash \xi \xrightarrow{\delta@p} \xi'\end{array}}{(\Sigma,H,\xi) \mapsto (\Sigma',H',\xi')}$$

Figure 8. Dynamic Semantics, top-level

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:

$$\frac{i \xrightarrow{\mathsf{wrp}}_H i_r \quad i' \xrightarrow{\mathsf{rf}}_H i_r \quad i \neq i'}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-READ)}$$

Here we write $i \xrightarrow{\mathsf{wrp}} i_r$ (pronounced "i is *write-read-prior* to i_r") to say that the read i_r 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_r reads from i' instead of i, we infer that i' is coherence-later than i.

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

$$\frac{i \xrightarrow{\mathsf{po}}_H i' \quad \mathsf{writes}_H(i,\ell,v) \quad \mathsf{reads}_H(i',\ell)}{i \xrightarrow{\mathsf{wrp}}_H i'} \text{ (WRP-PO)}$$

$$\frac{i \xrightarrow{\mathsf{vo}^+}_H \xrightarrow{\mathsf{xo}^*}_H i' \quad \mathsf{writes}_H(i,\ell,v) \quad \mathsf{reads}_H(i',\ell)}{i \xrightarrow{\mathsf{wrp}}_H i'} \text{ (WRP-VIS)}$$

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

$$\frac{i \xrightarrow{\mathsf{wwp}}_H i'}{i \xrightarrow{\mathsf{co}}_H i'} \text{ (CO-WRITE)}$$

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

$$\frac{i \xrightarrow{\mathsf{po}}_H i' \quad \mathsf{writes}_H(i,\ell,v) \quad \mathsf{writes}_H(i',\ell,v')}{i \xrightarrow{\mathsf{wwp}}_H i'} \text{ (WWP-PO)}$$

$$\frac{i \xrightarrow{\mathsf{vo}^+}_H \xrightarrow{\mathsf{xo}^*}_H i' \quad \mathsf{writes}_H(i,\ell,v) \quad \mathsf{writes}_H(i',\ell,v')}{i \xrightarrow{\mathsf{wwp}}_H i'} \text{ (WWP-VIS)}$$

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

$$\frac{i \overset{\mathsf{wrp}}{\rightarrow}_H i_r \quad i_r \overset{\mathsf{rwp}}{\rightarrow}_H i'}{i \overset{\mathsf{wwp}}{\rightarrow}_H i'} \text{ (WWP-READ)}$$

Read-write-priority ($\overset{\mathsf{rwp}}{\rightarrow}$), used here, has only a program-order rule. Although a visibility rule would also be sound, it is easy to show that it would not add any new coherence edges, so we omit it for simplicity.

$$\frac{i \overset{\mathsf{po}}{\rightarrow}_H i' \quad \mathsf{reads}_H(i, \ell) \quad \mathsf{writes}_H(i', \ell, v)}{i \overset{\mathsf{rwp}}{\rightarrow}_H i'} \text{ (RWP-PO)}$$

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

$$\frac{i_w \overset{\mathsf{rf}}{\rightarrow}_H i \quad H(\mathsf{is}(i, \mathsf{RW}[*,*])) \quad i_w \overset{\mathsf{co}}{\rightarrow}_H i' \quad i \neq i'}{i \overset{\mathsf{co}}{\rightarrow}_H i'} \text{ (CO-RW)}$$

The fourth and final coherence rule says that extra coherence edges can be given in the history. This is a technical device for Corollary 8; the operational semantics never introduces any such events.

$$\frac{H(\mathsf{co}(i, i'))}{i \overset{\mathsf{co}}{\rightarrow}_H i'} \text{ (CO-FIAT)}$$

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 $i_w \overset{\mathsf{co}}{\rightarrow} i'_w$ and i_r has seen i'_w. Then i_r cannot read from i_w. If it did, we would infer that $i'_w \overset{\mathsf{co}}{\rightarrow} i_w$, 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 $i_w \overset{\mathsf{po}}{\rightarrow} i'_w$. 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* [22], 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 $i_r \overset{\mathsf{po}}{\rightarrow} i_w$. 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 Wy2 before Ry1. We call this phenomenon a "leapfrogging write."

[8] Hence, RMC's "executed" is not the same thing as Sarkar, *et al.*'s "committed" [22].

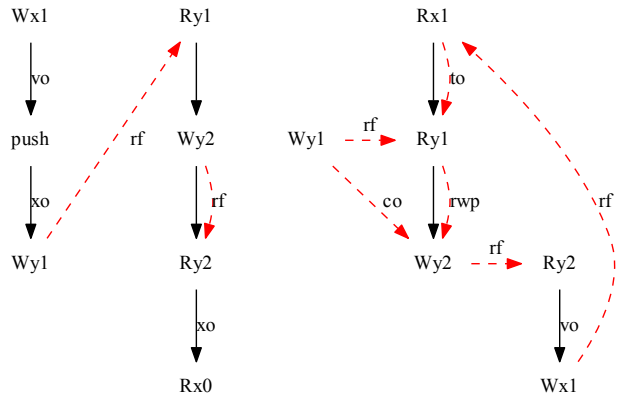


Figure 9. Leapfrogging writes (left observed, right allowed by Sarkar, *et al.* revised model)

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 ($\overset{\mathsf{to}}{\rightarrow}$) were used, RMC would not admit the right-hand trace in Figure 9 because then $\mathrm{Wy2} \overset{\mathsf{vo}}{\rightarrow}{}^{+} \mathrm{Rx1} \overset{\mathsf{to}}{\rightarrow} \mathrm{Ry1}$ would give us $\mathrm{Wy2} \overset{\mathsf{wrp}}{\rightarrow} \mathrm{Ry1}$, so $\mathrm{Wy2} \overset{\mathsf{co}}{\rightarrow} \mathrm{Wy1}$ (by CO-READ), which would contradict $\mathrm{Wy1} \overset{\mathsf{co}}{\rightarrow} \mathrm{Wy2}$ (which we have by CO-WRITE). This trace, although not yet observed on any processor, is admitted by the revision [21] of the Sarkar, *et al.* model [22], 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 $i_w \overset{\mathsf{po}}{\rightarrow} i_r$. Again, it does not follow that i_r must execute after i_w. Instead, it is conceivably possible that the architecture could aggressively complete the read and then ensure that i_w is coherence-earlier that the write that satisfied i_r.

This behavior has not been observed on any architecture, and we certainly do not advocate it. But in light of the existence of leapfrogged writes, this sort of "leapfrogged read" does not seem inconceivable.

Note, however, that this behavior can only occur in the presence of a data race. If i_w is coherence-later than all other writes to the relevant location, as would be the case in properly synchronized code, then i_r cannot read from any other write. In that case, i_r must read from i_w, and thus it cannot execute until i_w does.

Semantic deadlock One consequence of the very loose semantics of RMC's stores is that execution can find its way into a dead end in which no more progress is possible. We call this phenomenon *semantic deadlock.* To be clear, this is a distinct phenomenon from ordinary deadlock, in which a buggy program is unable to make progress, often due to a faulty locking protocol. In *semantic* deadlock a correct program is nevertheless unable to make progress due to inconsistent choices made by the nondeterministic semantics.

Semantic deadlock arises when any progress would create a cycle in coherence order. This can happen because of reads or pushes. An example of the former is shown in the left-hand trace in Figure 10. In this trace, Rx? cannot read from Wx1, because if $\mathrm{Wx1} \overset{\mathsf{rf}}{\rightarrow} \mathrm{Rx?}$ then $\mathrm{Wy2} \overset{\mathsf{vo}}{\rightarrow}{}^{+} \mathrm{Rx?}$, so $\mathrm{Wy2} \overset{\mathsf{wwp}}{\rightarrow} \mathrm{Wy1}$, which would create a cycle in coherence order. However, no other write to x is available, so Rx? cannot execute.

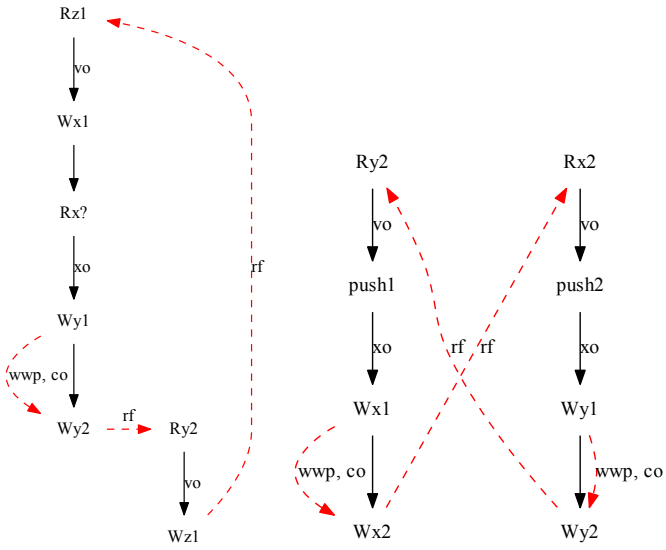


Figure 10. Semantic deadlock

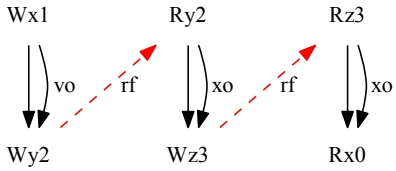


Figure 11. ISA2 litmus test

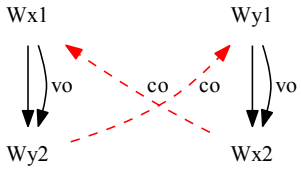


Figure 12. 2+2W litmus test

An example of the latter is shown in the right-hand trace of Figure 10. In this trace, neither push nor Wx1 nor Wy1 can execute. Suppose push1 executes before Wy1. Then push1 $\overset{\text{po}}{\to}$ Wy1, so Wy2 $\overset{\text{vo}}{\to}^{+}$ Wy1, and so Wy2 $\overset{\text{wwp}}{\to}$ Wy1, which creates a cycle in coherence order. Thus Wy1 must execute before push1. Similarly, Wx1 must execute before push2. But push1 $\overset{\text{xo}}{\to}$ Wx1 and push2 $\overset{\text{xo}}{\to}$ Wy1 so no progress can be made.

Obviously, semantic deadlock cannot occur in a real execution. Therefore we restrict our attention to *consistent histories*, 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 [23] 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 [22] 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 `isync` (Power) or `isb` (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.* [9]. 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 $i \overset{\text{vo}}{\to} i'$ 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 [5, 22] in Figure 11. This trace is possible in RMC. The edge `Ry2` $\overset{\text{xo}}{\to}$ `Wz3` breaks up the chain of visibility order between `Wx1` and `Rz3`, so we cannot show that `Wx1` is prior to `Rx0`.

However, when compiled into Power, using an `lwsync` for $\overset{\text{vo}}{\to}$, and a dependency for $\overset{\text{xo}}{\to}$, this trace is forbidden. Power guarantees *B-cumulativity* [5, 15, 22], 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` $\overset{\text{vo}}{\to}$ `Wz3`, and causes `Wx1` to be prior to `Rx0`.

To prevent this, the programmer would need to insert a push between `Wx1` and `Wy2`, or upgrade the `Ry2` $\to$ `Wz3` edge to visibility order. Either way, some additional overhead might be incurred.

As another example, consider the 2+2W litmus test [5, 22] in Figure 12. Nothing in RMC prevents this trace from taking place. However, when compiled into Power, using an `lwsync` for $\overset{\text{vo}}{\to}$, this trace is forbidden. Power guarantees [5, 22] the acyclicity of a relation that in RMC we would write $\overset{\text{co}}{\to} \cup \overset{\text{vo}}{\to}$, 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 $\vdash (\Sigma, H, \xi)\mathsf{ok}$ with the one rule:

$$\frac{\vdash H : \Sigma \quad \Sigma \vdash \xi\ \mathsf{ok}}{\vdash (\Sigma, H, \xi)\ \mathsf{ok}}$$

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 Σ. The details appear in the appendix [12].

Now we can state the preservation theorem:

THEOREM 1 (Preservation). *If $s \mapsto s'$ and $\vdash s$ ok then $\vdash s'$ 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 *moot* if it contains an unresolvable speculation ($v = v'$ in e, where v and v' 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 right. The details can be found in the appendix [12]. Then, instead of "unstuckness", we prove:

THEOREM 2 (Rightness). *If $\vdash s$ ok then s right.*

This is similar to Wright and Felleisen's original technique [24], 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 right, then either (1) for every thread $p : e$ in s, e is final, or (2) there exists s' such that $s \mapsto s'$ 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.[9] A *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 $\overset{\mathsf{ssc}}{\to}$, $\overset{\mathsf{co}}{\to}$, $\overset{\mathsf{rf}}{\to}$, and $\overset{\mathsf{fr}}{\to}$:

$$\begin{array}{lcl} i \overset{\mathsf{ssc}}{\to}_H i' & \overset{\text{def}}{=} & \exists i_p.\, H(\mathsf{is}(i_p, \mathsf{Push})) \wedge i \overset{\mathsf{vo}}{\to}^+_H i_p \overset{\mathsf{xo}}{\to}^+_H i' \\ i \overset{\mathsf{fr}}{\to}_H i' & \overset{\text{def}}{=} & \exists i_w.\, i_w \overset{\mathsf{rf}}{\to}_H i \wedge i_w \overset{\mathsf{co}}{\to}^+_H i' \\ i \overset{\mathsf{sc}}{\to}_H i' & \overset{\text{def}}{=} & i \overset{\mathsf{ssc}}{\to}_H i' \vee i \overset{\mathsf{co}}{\to}_H i' \vee i \overset{\mathsf{rf}}{\to}_H i' \vee i \overset{\mathsf{fr}}{\to}_H i' \end{array}$$

We also define *communication order* [2] as sequentially consistent order without $\overset{\mathsf{ssc}}{\to}$:

$$i \overset{\mathsf{com}}{\to}_H i' \overset{\text{def}}{=} i \overset{\mathsf{co}}{\to}_H i' \vee i \overset{\mathsf{rf}}{\to}_H i' \vee i \overset{\mathsf{fr}}{\to}_H i'$$

An important property of communication order is 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, $\mathsf{acyclic}(\overset{\mathsf{co}}{\to}_H)$). If $i_1 \overset{\mathsf{xo}}{\to}^+_H i_2 \overset{\mathsf{com}}{\to}^*_H i_3 \overset{\mathsf{vo}}{\to}^+_H i_4$, where i_1 and i_4 are pushes, then $i_1 \overset{\mathsf{to}}{\to}_H i_4$.*

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 $\overset{\mathsf{com}}{\to}$ relates only reads and writes) and the result follows immediately. If i_3 is a read, then it reads from some write i'_3, and we can rearrange to obtain $i_1 \overset{\mathsf{xo}}{\to}^+_H i_2 \overset{\mathsf{com}}{\to}^*_H i'_3 \overset{\mathsf{vo}}{\to}^+_H i_4$ (since $i'_3 \overset{\mathsf{vo}}{\to}_H i_3$).

Since H is complete, i_1 and i_4 are executed, so either $i_1 \overset{\mathsf{to}}{\to}_H i_4$ or $i_4 = i_1$ or $i_4 \overset{\mathsf{to}}{\to}_H i_1$. In the first case we are done. In both of the remaining cases we have $i_4 \overset{\mathsf{vo}}{\to}^*_H i_1$, either trivially or using push order. Therefore $i_3 \overset{\mathsf{vo}}{\to}^+_H i_1$. From this we draw a contradiction.

Suppose i_2 is a write. Then $i_2 \overset{\mathsf{co}}{\to}^*_H i_3$ (since $i_2 \overset{\mathsf{com}}{\to}^*_H i_3$). However, we also have $i_3 \overset{\mathsf{wwp}}{\to}_H i_2$ so $i_3 \overset{\mathsf{co}}{\to}_H i_2$, which is a contradiction.

On the other hand, suppose i_2 is a read. Let $i_w \overset{\mathsf{rf}}{\to}_H i_2$. Then $i_w \overset{\mathsf{com}}{\to}^+_H i_3$, so $i_w \overset{\mathsf{co}}{\to}^+_H i_3$. However, $i_3 \overset{\mathsf{wrp}}{\to}_H i_2$ so $i_3 \overset{\mathsf{co}}{\to}_H i_w$, which is a contradiction. □

COROLLARY 5. *Suppose H is trace coherent, complete, and coherence acyclic. If $i_1 \overset{\mathsf{xo}}{\to}^+_H i_2 \overset{\mathsf{sc}}{\to}^*_H i_3 \overset{\mathsf{vo}}{\to}^+_H i_4$, where i_1 and i_4 are pushes, then $i_1 \overset{\mathsf{to}}{\to}_H i_4$.*

Proof

By induction on the number of $\overset{\mathsf{ssc}}{\to}$ steps in $i_2 \overset{\mathsf{sc}}{\to}^*_H i_3$.

THEOREM 6. *Suppose H is trace coherent, complete, and coherence acyclic. Then $\overset{\mathsf{sc}}{\to}_H$ is acyclic.*

[9] In practice the execution-order edge from i_p to i' is ordinarily provided by labelling the push with $\overset{\mathsf{exe}}{\triangleright}$.

Proof

Suppose $i \overset{\mathsf{sc}}{\rightarrow}_H^+ i$. We show there must be at least one $\overset{\mathsf{ssc}}{\rightarrow}$ edge in the cycle. Suppose the contrary. Then $i \overset{\mathsf{com}}{\rightarrow}_H^+ i$. If i is a write, then $i \overset{\mathsf{co}}{\rightarrow}_H^+ i$, which is a contradiction. If i is a read, then $i \overset{\mathsf{com}}{\rightarrow}_H^* i_w \overset{\mathsf{rf}}{\rightarrow}_H i$, so $i_w \overset{\mathsf{co}}{\rightarrow}_H^+ i_w$, which is a contradiction.

Thus suppose that $i_1 \overset{\mathsf{ssc}}{\rightarrow}_H i_2 \overset{\mathsf{sc}}{\rightarrow}_H^* i_1$. Expanding $\overset{\mathsf{ssc}}{\rightarrow}$ we obtain $i_1 \overset{\mathsf{vo}}{\rightarrow}_H^+ i_p \overset{\mathsf{xo}}{\rightarrow}_H^+ i_2 \overset{\mathsf{sc}}{\rightarrow}_H^* i_1$ for a push i_p. By Corollary 5, $i_p \overset{\mathsf{to}}{\rightarrow}_H i_p$, which is a contradiction. □

DEFINITION 7. A history H is *consistent* if there exists H' containing no is events, such that H, H' is trace coherent, complete, and coherence acyclic.

COROLLARY 8 (Sequential consistency). *Suppose H is consistent. Suppose further that S is a set of identifiers such that for all $i, i' \in S$, $i \overset{\mathsf{po}}{\rightarrow}_H i'$ implies $i \overset{\mathsf{ssc}}{\rightarrow}_H i'$. Then there exists a sequentially consistent ordering for S (in the sense of Lamport [17]).*

Proof Sketch

Let H' extend H so that H' is complete and $\overset{\mathsf{co}}{\rightarrow}_{H'}$ totally orders writes to the same location. By Theorem 6, $\overset{\mathsf{sc}}{\rightarrow}_{H'}$ is acyclic, so there exists a total order containing it. By Alglave [2, §4.2.1.3], such an order is a sequentially consistent order for the actions in S.

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 programs. We define *synchronized-before* as follows:

$$i \overset{\mathsf{sb}}{\rightarrow}_H i' \overset{\text{def}}{=} \exists i_s.\, i \overset{\mathsf{vo}}{\rightarrow}_H^+ i_s \overset{\mathsf{po}}{\rightarrow}_H^* i' \wedge H(\mathsf{label}(\overset{\mathsf{exe}}{\triangleright}, i_s))$$

In essence $i \overset{\mathsf{sb}}{\rightarrow} i'$ says that i is visible to some synchronization operation i_s that is execution-order before i'. That i_s must be labelled $\overset{\mathsf{exe}}{\triangleright}$ 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 a program is data-race-free if any two conflicting actions on different threads are ordered by synchronized-before. Boehm and Adve [11] refer to this sort of data race as a "type 2 data race." We may then show that data-race-free programs are sequentially consistent:

THEOREM 9. *Suppose H is trace coherent, coherence acyclic, and data-race-free. Let $\overset{\mathsf{scdrf}}{\rightarrow}_H$ be the union of $\overset{\mathsf{po}}{\rightarrow}_H$, $\overset{\mathsf{co}}{\rightarrow}_H$, $\overset{\mathsf{rf}}{\rightarrow}_H$, and $\overset{\mathsf{fr}}{\rightarrow}_H$. Then $\overset{\mathsf{scdrf}}{\rightarrow}_H$ is acyclic.*

Proof Sketch

Let $\overset{\mathsf{sq}}{\rightarrow}_H$ be the union of $\overset{\mathsf{po}}{\rightarrow}_H$ and $\overset{\mathsf{sb}}{\rightarrow}_H$. We show that $\overset{\mathsf{sq}}{\rightarrow}_H$ is acyclic and contains $\overset{\mathsf{scdrf}}{\rightarrow}_H$.

COROLLARY 10 (Sequential consistency). *Suppose H is consistent and data-race-free. Then there exists a sequentially consistent ordering for all identifiers in H.*

These two sequential consistency results can also be combined into a single theorem for programs that contain some data-race-free code 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.* [16], as follows:

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

$$\frac{e \overset{\delta}{\mapsto} e' \quad \delta \text{ not of the form } i \Downarrow v}{v = v' \text{ in } e \overset{\delta}{\mapsto} v = v' \text{ in } e'}$$

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 that write could have taken place without speculation:

$$\frac{\begin{array}{c}\Sigma = (\Upsilon, \Phi, \Psi) \qquad \Sigma_{alt} = (\Upsilon_{alt}, \Phi_{alt}, \Psi_{alt}) \\ \Sigma \overset{i \Downarrow v@p}{\longmapsto} \Sigma' \qquad H \overset{i \Downarrow v@p}{\longmapsto} H' \qquad \Upsilon; \Phi \vdash \xi \overset{i \downarrow v@p}{\longmapsto} \xi' \\ (\Sigma, H, \xi) \mapsto^* (\Sigma_{alt}, H_{alt}, \xi_{alt}) \\ H_{alt} \overset{i \Downarrow v@p}{\longmapsto} H'_{alt} \qquad \Upsilon_{alt}; \Phi_{alt} \vdash \xi_{alt} \overset{i \Downarrow v@p}{\longmapsto} \xi'_{alt}\end{array}}{(\Sigma, H, \xi) \mapsto (\Sigma', H', \xi')}$$

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 [6] illustrate, it is very difficult to distinguish between thin-air traces and some desirable traces.

We are not troubled by the fact that real architectures do not exhibit thin-air traces, as RMC is designed to be looser than real architectures anyway. Thin-air traces do not implicate type safety; our speculation construct produces only well-typed values. Batty, *et al.* [10] observe that thin-air traces can create difficulties for modular reasoning, but they also observe that the problem can be resolved by type-checking.

However, one negative consequence of thin-air reads that is still relevant to RMC is a failure of abstraction. They allow a program to produce a member of an abstract type by means other than calling the module that defines it, which makes Reynolds-style abstraction [20] fail. We see this as the central problem for thin-air traces.

Yet the problem also suggests a possible path forward. As yet there is no rigorous theory of Reynolds-style abstraction that even comes close to supporting RMC anyway. Such a theory would offer some structure more compelling than mere intuition for identifying problematic thin-air traces, and hopefully for ruling them out. This is an important avenue for future work.

Related work RMC's store is inspired by the storage subsystem of Sarkar, *et al.* [22]. 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 $\overset{\text{co}}{\rightarrow} \cup \overset{\text{vo}}{\rightarrow}$ 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.* [16]. 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.

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 1995.

[2] J. Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris VII, Nov. 2010.

[3] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41:178–210, 2012.

[4] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Twenty-Second International Conference on Computer Aided Verification*, 2010.

[5] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 2014. To appear.

[6] M. Batty and P. Sewell. The thin-air problem. Working note, available at `http://www.cl.cam.ac.uk/~pes20/cpp/notes42.html`, Feb. 2014.

[7] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency: The post-Rapperswil model. Technical Report N3132, ISO IEC JTC1/SC22/WG21, Aug. 2010. Available electronically at `www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf`.

[8] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Thirty-Eighth ACM Symposium on Principles of Programming Languages*, Austin, Texas, Jan. 2011.

[9] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Thirty-Ninth ACM Symposium on Principles of Programming Languages*, Philadelphia, Pennsylvania, Jan. 2012.

[10] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Fortieth ACM Symposium on Principles of Programming Languages*, Rome, Italy, Jan. 2013.

[11] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *2008 SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, Arizona, June 2008.

[12] K. Crary and M. J. Sullivan. A calculus for relaxed memory. Technical Report CMU-CS-14-139, Carnegie Mellon University, School of Computer Science, 2014.

[13] R. Grisenthwaite. ARM barrier litmus tests and cookbook. `http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf`, 2009.

[14] D. Howells and P. E. McKenney. Circular buffers. `https://www.kernel.org/doc/Documentation/circular-buffers.txt`.

[15] IBM. Power ISA™ version 2.06 revision B, 2010.

[16] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Nineteenth European Symposium on Programming*, 2010.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[18] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Thirty-Second ACM Symposium on Principles of Programming Languages*, Long Beach, California, Jan. 2005.

[19] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, Aug. 2001.

[20] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[21] S. Sarkar, P. Sewell, J. Alglave, and L. Maranget. ppcmem executable model (ARM version). Unpublished code, 2011.

[22] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *2011 SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, California, June 2011.

[23] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7), July 2010.

[24] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.