A Promising Semantics for Relaxed-Memory Concurrency

Jeehoon Kang Chung-Kil Hur [*] Ori Lahav Viktor Vafeiadis Derek Dreyer

Seoul National University, Korea
{jeehoon.kang,gil.hur}@sf.snu.ac.kr

MPI-SWS, Germany [†]
{orilahav,viktor,dreyer}@mpi-sws.org

Abstract

Despite many years of research, it has proven very difficult to develop a memory model for concurrent programming languages that adequately balances the conflicting desiderata of programmers, compilers, and hardware. In this paper, we propose the first relaxed memory model that (1) accounts for a broad spectrum of features from the C++11 concurrency model, (2) is implementable, in the sense that it provably validates many standard compiler optimizations and reorderings, as well as standard compilation schemes to x86-TSO and Power, (3) justifies simple invariant-based reasoning, thus demonstrating the absence of bad "out-of-thin-air" behaviors, (4) supports "DRF" guarantees, ensuring that programmers who use sufficient synchronization need not understand the full complexities of relaxed-memory semantics, and (5) defines the semantics of racy programs without relying on undefined behaviors, which is a prerequisite for applicability to type-safe languages like Java.

The key novel idea behind our model is the notion of *promises*: a thread may promise to execute a write in the future, thus enabling other threads to read from that write out of order. Crucially, to prevent out-of-thin-air behaviors, a promise step requires a thread-local certification that it will be possible to execute the promised write even in the absence of the promise. To establish confidence in our model, we have formalized most of our key results in Coq.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords Weak memory models; C++11; operational semantics

1. Introduction

What is the right semantics for concurrent shared-memory programs written in higher-level languages? For programmers, the simplest answer would be a *sequentially consistent (SC)* semantics, in which all threads in a program share a single view of memory and writes to memory take immediate global effect.

However, a naive SC semantics is costly to implement. First of all, commodity architectures (such as x86, Power, and ARM) are not SC: they execute memory operations speculatively or out of order, and they employ hierarchies of buffers to reduce memory latency, with the effect that there is no globally consistent view of memory shared by all threads. To simulate SC semantics on these architectures, one must therefore insert expensive fence instructions to subvert the efforts of the hardware. Secondly, a number of common compiler optimizations—such as constant propagation—are rendered unsound by a naive SC semantics because they effectively reorder memory operations. Moreover, SC semantics is stronger (*i.e.,* more restrictive) than necessary for many concurrent algorithms.

Hence, languages like Java and C++ have opted instead to provide *relaxed* (aka *weak*) memory models [21, 13], which enable programmers to demand SC semantics when they need it, but which also support a range of cheaper memory operations that trade off strongly consistent and/or well-defined behavior for efficiency.

1.1 Criteria for a Programming Language Memory Model

Unfortunately, despite many years of research, it has proven very difficult to develop a memory model for concurrent programming languages that adequately balances the conflicting desiderata of programmers, compilers, and hardware. In particular, we would like to find a memory model that satisfies the following properties:

- The model should be *implementable*, *i.e.,* it should validate common compiler optimizations, as well as standard compilation schemes to the major modern architectures. To be implementable, it must justify many kinds of instruction reordering and merging.
- The model should support *high-level reasoning* principles that programmers and compiler analyses depend on. At a bare minimum, it should validate simple invariant-based verification, and should provide some "DRF" guarantees [4], ensuring that programmers who employ sufficient synchronization need not understand the full complexities of relaxed-memory semantics.
- The model should ideally *avoid relying on undefined behavior* to define the semantics of racy programs. This is a prerequisite for applicability to type-safe languages like Java, in which well-typed programs may contain data races but are nevertheless expected to have safe, well-defined semantics.

Both Java and C++ fail to achieve some of these criteria.

In the case of Java, the memory model fails to validate a number of common program transformations performed by real Java compilers, such as redundant read-after-read elimination and "roach motel" reordering [26]. Although this problem has been known for some time, a satisfactory solution has yet to be developed.

In the case of C++, the memory model relies crucially on undefined behaviors to give semantics to racy programs. Moreover, it permits certain "out-of-thin-air" executions which violate basic invariant-based reasoning (and DRF guarantees) [7].

1.2 The "Out of Thin Air" Problem

To illustrate the problem with C++, consider these two variants of the classic "load buffering" litmus test (with two threads in parallel):

$$\begin{array}{r||l} a := x; & x := y; \\ y := 1; & \end{array} \quad \text{(LB)} \qquad \begin{array}{r||l} a := x; & x := y; \\ y := a; & \end{array} \quad \text{(LBd)}$$

[*] Corresponding author. [†] Saarland Informatics Campus.

Here, we assume that all variables are initially 0, and that all memory accesses are of the weakest consistency level, *i.e.,* they are compiled down to plain loads and stores at the hardware level with no additional synchronization (in C++ this is called "relaxed"). The question is: should it be possible for these programs to assign 1 to a? In the case of LB, the answer is yes: architectures like Power and ARM may reorder the write of y before the read of x in the first thread (since these are accesses to distinct variables), after which a can be assigned 1 by a standard interleaving execution. In the case of LBd, however, the answer *ought* to be no: all the operations simply copy one variable to another and all are initially 0, so if a could receive 1, it would come "out of thin air". No hardware reorderings or reasonable compiler optimizations will produce this behavior. If they did, it would cause major problems: one would not be able to establish even basic invariants (such as $x = y = 0$), and basic sanity results like the aforementioned DRF theorems would cease to hold. It is therefore a serious problem that the formal memory model of C++ allows such out-of-thin-air (OOTA) behavior.

Intuitively, the reason C++ allows OOTA behaviors is that it is not clear how to distinguish them from acceptable behaviors. The C++ model formalizes valid executions as graphs of memory access events (think: partially-ordered traces) subject to a set of coherence axioms, and the same coherent event graph that describes a valid execution of LB in which a receives 1 also describes a valid execution of LBd in which a receives 1.

Hardware memory models (*e.g.,* Power and ARM) handle this problem by taking syntactic dependencies between instructions into account in determining program semantics. Under such models, the out-of-order execution in LB is valid because the write to y is independent of the read from x, whereas in LBd such out-of-order execution is prevented by the syntactic dependency between the two instructions. Although this approach is suitable for modeling hardware, it is too brittle for a language-level semantics because it fails to validate standard compiler optimizations that remove syntactic dependencies (see also [7]). As a very simple example, consider the following variant of LB and LBd:

$$\begin{array}{l} a := x; \\ y := a + 1 - a; \end{array} \;\middle\|\; x := y; \qquad \text{(LBfd)}$$

Under the hardware models, this LBfd program would be treated similarly to LBd due to the syntactic data dependency, so a could not receive 1. But even a basic optimizing compiler could trivially transform LBfd to LB, in which case a could receive 1.

As a result, we still to this day lack a semantics for relaxed-memory concurrency in Java and C++ that corresponds to how these languages are implemented and that provides sufficient reasoning guarantees to programmers and compiler-writers. Several proposals have recently been made for how to fix the Java and C++ memory models (some of which are discussed in §6), but none have been proven to validate the full range of standard optimizations/reorderings performed by Java and C++ compilers and by commodity hardware like Power and ARM. Furthermore, for most of the existing proposals, it is known that indeed they do *not* validate some important reorderings.

1.3 A "Promising" Semantics for Relaxed Memory

In this paper, we present what we believe is a very promising way forward: the first relaxed memory model to support a broad spectrum of features from the C++ concurrency model while also satisfying all three criteria listed in §1.1.

We achieve these ends through a combination of mechanisms (some standard, some not), but the most important and novel idea for the reader to take away from this paper is the notion of *promises*.

Under our model, which is defined by an operational semantics, a thread T may nondeterministically "promise" to write a value v to a memory location x at some point in the future. From the point of view of other threads, a promise is no different from an ordinary write: once T has promised to write v to x, other threads can read from that write. (In contrast, T cannot read from its own promised write until T has fulfilled the promise: this is crucial to preserve basic sanity of the semantics.) Intuitively, promises simulate the effect of read-write reorderings by allowing write events to be visible to other threads before the point at which they occur in the program order.

We must, however, ensure that promises do not introduce bad OOTA behaviors. Toward this end, we only allow T to promise to write v to x if it is possible to *thread-locally certify* that the promise can be fulfilled in a finite number of steps. That is, we must show that T will be able to write v to x after some finite sequence of steps of T's execution (*i.e.,* with no help from other threads). The certification requirement guarantees absence of bad OOTA executions by ensuring that T can only promise to write a value v to x if T could have written v to x anyway.

Returning to the examples from §1.2, it is easy to see how promises give us the desired semantics:

- In LB, the first thread can promise to write 1 to y (since it will indeed write 1 to y no matter what value is assigned to a), after which the second thread can read from that promise and write 1 to x. Subsequently, the first thread can execute normally, reading 1 from x and assigning it to a.
- The execution of LBfd may proceed in exactly the same way. The fact that the write of y depends syntactically on a is irrelevant, because during certification of the promised write of 1 to y, the expression $a + 1 - a$ will always evaluate to 1.
- By contrast, the OOTA behavior will not be allowed for LBd. In order for the first thread to promise to write 1 to y, it would need to certify that it can write 1 to y without promises. But since all variables are initially 0, this is not possible.

Our model supports all features of C++ concurrency except consume reads and SC accesses. Consume reads are widely considered a premature aspect of the C++11 standard and are implemented as acquire reads in mainstream compilers. In contrast, SC accesses are a major feature of C++, and originally our model included an account of SC accesses as well. However, in the course of trying to mechanize correctness of compilation to Power (§5.3), we discovered that our semantics of SC accesses was flawed, and this led us to discover a flaw in the C++11 standard as well! (See [19] for further details.) Thus, a proper handling of SC accesses remains an important problem for future work.

In the rest of the paper, we will flesh out the idea of promises—as well as the other elements of our model—in layers. We begin in §2 by presenting the details of our model restricted to relaxed reads and writes. In §3, we extend this base model further to support atomic updates (*i.e.,* read-modify-write operations, like CAS). Then, in §4, we scale the model up to handle most features of the C++ memory model. In §5, we present our formal results—validating many program transformations, compilation to x86-TSO and Power, DRF theorems, and an invariant-based logic—most of which are fully mechanized in the Coq proof assistant (totalling about 30K lines of Coq). In §6, we compare with related work, and in §7, we conclude with discussion of future work.

2. Basic Model for Handling Relaxed Accesses

In this section, we introduce the key ideas of our memory model, first by example and then more formally. At first we will only consider a semantics for fully "relaxed" atomic read and write accesses (in the sense of C++). This is a natural starting point, since the OOTA problem is fundamentally about how to give a reasonable semantics for these relaxed accesses, and the key elements of our solution are easiest to understand in this simpler setting. We will see in subsequent sections how to extend and generalize this base model to account for a much richer variety of memory operations.

To illustrate our semantics, we will write small programs such as the following:

$$\begin{array}{l\|l} x := 1; & y := 1; \\ a := y; \ /\!/\, 0 & b := x; \ /\!/\, 0 \end{array} \qquad \text{(SB)}$$

As a convention, we write a, b, c for local variables (registers) and x, y, z for (distinct) shared memory locations, and assume that all variables are initialized to 0. We refer to thread i as T_i. Moreover, in order to refer to a specific observation of the program, we annotate the corresponding reads with the values expected to be read (*e.g.,* in the above program, the comment notation indicates the observed result that $a = b = 0$).

2.1 Main Ideas

High-Level Requirements: Reorderings and Coherence Relaxed read and write operations are intended to be compiled down directly to plain loads and stores at the machine level, so one of the main requirements of our semantics is that it be at least as permissive as commodity hardware. Toward this end, our semantics must justify reordering of independent memory operations (*i.e.,* operations that access distinct locations), since the more weakly consistent architectures (like ARM) may potentially perform such reorderings. There are four such classes of reorderings—write-read, write-write, read-read, and read-write—and in §5 we will prove formally that our semantics justifies all of them.

On the other hand, it is also important that our semantics not be unnecessarily weak. In particular, all the existing implementations of C++, even for weaker architectures like Power and ARM, guarantee at a bare minimum a property we call *per-location coherence* (aka *SC-per-location*). Per-location coherence says that, even though threads may observe writes to different locations in different orders, they must observe writes to the *same* location in a single total order (called the "modification order" in C++ lingo). In addition to being supported by hardware, per-location coherence is preserved by common compiler optimizations as well. Hence, we want our semantics of relaxed accesses to guarantee it. (In §4.3 we will present an even weaker mode of accesses that does not provide full per-location coherence.)

Operational Semantics with Timestamps In contrast to the C++ memory model, which relies on declarative semantics over event graphs, ours employs a more standard SC-style operational semantics for concurrency, in which the executions of different threads are nondeterministically interleaved. However, in order to account for weak memory behaviors, we use a more elaborate memory representation than the standard SC semantics does. Instead of being a flat map from addresses to values, our memory records the set of all writes ever performed. It may help to think of writes as messages, and memory as a message pool which grows monotonically. When a thread T reads from a location x, it need not read "the latest" write to x, since there is no shared understanding among threads of what the latest write is. The thread T thus retains flexibility in terms of which message it reads, but we must place some restrictions on this flexibility in order to guarantee per-location coherence.

Specifically, we totally order the writes to each location by attaching a (unique) *timestamp* to each write message. Thus, messages are triples of the form $\langle x : v@t \rangle$ (where x is a location, v a value, and t a timestamp). (The *modification order* for a location x is thus implicitly derivable from the order of timestamps on x's messages.) In addition, for each thread T, we keep track of a map from locations x to the largest timestamp of a write to x that T has observed or executed. We refer to this map as T's *view* of memory, and one can think of it as recording the set of most recent write messages that T has observed. Hence, when T reads from a location x, it must read from a message with a timestamp *at least as large* as the one recorded for x in T's view. And when T writes to x, it must pick a timestamp *strictly larger* than the one recorded for x in its view.

Let us see now how our semantics, as explained thus far, already suffices to justify desirable reorderings while ruling out violations of coherence. First, recall the write-read reordering exhibited by the "store buffering" SB example above, and let us see how the behavior can be justified. Initially, assume the memory contains the initialization messages $\langle x : 0@0 \rangle$ and $\langle y : 0@0 \rangle$, and both threads maintain a view of x and y that maps them to 0. When T_1 performs the assignment $x := 1$, it will choose some timestamp $t > 0$, add the message $\langle x : 1@t \rangle$ to the memory, and update its view of x to t. But this will have no effect on its view of y or T_2's view of x, which remain at 0. Thus, when T_1 subsequently reads y, it is free to read 0. (And analogously for T_2.)

On the flip side, per-location timestamps also explain why the following coherence violation is forbidden.

$$\begin{array}{l\|l} x := 1; & x := 2; \\ a := x; \ /\!/\, 2 & b := x; \ /\!/\, 1 \end{array} \qquad \text{(COH)}$$

Here, the two writes to x must be totally ordered. Suppose, without loss of generality, that the $x := 1$ was written at timestamp 1 and $x := 2$ at timestamp 2. Then, although T_1 may read value 2, T_2 cannot read 1, because 1 was written at a smaller timestamp than the one that T_2 already recorded in its view when it wrote $x := 2$.

One subtle point is that, when writing to a location x, a thread T may select any unused timestamp t larger than the one recorded in its view of x, but t need not be globally maximal. That is, t may be smaller than the timestamp that another thread has already used for a write to x. This freedom is in fact crucial in order to permit write-write reorderings, as exemplified by the following test case:

$$\begin{array}{l\|l} x := 2; & y := 2; \\ y := 1; & x := 1; \\ a := y; \ /\!/\, 2 & b := x; \ /\!/\, 2 \end{array} \qquad \text{(2+2W)}$$

To get the desired weak outcome, the writes of $x := 1$ and $y := 1$ must pick smaller timestamps than the $x := 2$ and $y := 2$ writes, respectively, but at least one of the 1-writes must be executed *after* the 2-write to the same location. Thus, it is essential to be able to write using a timestamp that is not globally maximal.

Promises Unfortunately, our timestamp semantics alone does not suffice to explain *read-write* reorderings, as exemplified by the (LB) and (LBfd) programs from §1.2. It is precisely these reorderings that motivate our introduction of *promises*.

As explained in §1.3, a thread T may at any point *promise* to write $x := v$ at some timestamp t (provided that t is greater than T's current view of x). This promise is treated to a large extent like an actual write operation. In particular, it adds a new message $\langle x : v@t \rangle$ to memory, which may then be read by other threads. However, in order to make such a promise, T must *thread-locally certify* it—that is, T must demonstrate that it will be able to fulfill this promise (writing $x := v$ at timestamp t) in a finite number of thread-local steps. Certification is needed to guarantee plausibility of the promise, but crucially, there is no requirement that the specific steps of execution taken during certification must match the subsequent steps of actual execution. Indeed, we already witnessed this with the (LB) and (LBfd) executions, where T_1 read $x = 0$ during the initial certification of its promised write to y, but read $x = 1$ during the actual execution.

Let us now briefly touch on a few technical points concerning the interaction of promises and timestamps.

First of all, it is important that T cannot directly read its own promises, because this would violate per-location coherence: for example, the single-threaded program $a := x;\ x := 1$ would be able to return $a = 1$! Note that we do not need to explicitly enforce this restriction—it just falls out from our rules concerning timestamps. In particular, if T were to promise $\langle x : v@t \rangle$, and then were to read from its own promise, then T's view of x would be updated to t, and there would be no way for T to subsequently fulfill the promise because it would have to pick a timestamp strictly greater than t when performing the assignment $x := v$.

That said, it is possible for T to read its promised value indirectly via another thread, as in the LB and LBfd programs. It may even read the promised value from the same location where it promised to write it, as in the following example [18].

$$\begin{array}{l} a := x; \ /\!/ 1 \\ x := 1; \end{array} \left\| \ y := x; \ \right\| \ x := y; \qquad \text{(ARM-weak)}$$

This outcome can be explained by T_1 promising $\langle x : 1@2 \rangle$, then T_2 reading $x = 1$ and storing it to y, and T_3 reading $y = 1$ and writing $x := 1$ at timestamp 1, which T_1 can read before fulfilling its promise. Such behavior, strange though it may seem, is actually allowed (though not yet observed) by the ARM memory model [11]

Last but not least, we wish to ensure that promises do not lead to impossible situations later down the road, *i.e.,* that making a promise cannot cause the execution of a program to get stuck. The thread-local certification that accompanies a promise step goes some way toward ensuring this progress condition, but it is not enough. We also amend the semantics in the following two ways:

1. Every step a thread takes, it must *re-certify* all its outstanding promises to make sure they can *still* be fulfilled. To see why, consider a possible execution of the following program:

   $$\begin{array}{l} a := x; \\ x := 1; \end{array} \ \Big\| \ x := 2;$$

   Suppose that T_1 (for no particularly good reason) promises $\langle x : 1@1 \rangle$. At first, this is easy to certify: T_1 can read the initial value of x (the message $\langle x : 0@0 \rangle$), and then perform the assignment $x := 1$ picking timestamp 1. Suppose then that T_2 picks the timestamp 2 when performing $x := 2$. If at this point in the execution T_1 were permitted to read the message $\langle x : 2@2 \rangle$, it would have the effect of bumping up T_1's view of x to timestamp 2, which would prevent it from subsequently fulfilling its promise. It is thus crucial that T_1 *not* be allowed to read $x = 2$ (in this particular execution), and indeed our semantics will not allow it to do so because the re-certification check would fail. As the example illustrates, promises can restrict a thread's future nondeterministic choices concerning the messages it reads.

2. We require the total order on timestamps to be *dense* (*e.g.,* choosing timestamps to be rational numbers), so that there is always a place to put intermediate writes before a promise. Consider, for example, the following program:

   $$\begin{array}{l} x := 1; \\ x := 2; \end{array} \ \Big\| \ x := 3;$$

   Here, T_1 may promise $\langle x : 2@2 \rangle$—in validating this promise, T_1 might write $\langle x : 1@1 \rangle$ before writing $\langle x : 2@2 \rangle$. If, however, T_2 subsequently writes $\langle x : 3@1 \rangle$ before T_1 has actually written $x := 1$, then T_1 can no longer pick 1 as a timestamp for $x := 1$. To make progress here, T_1 needs a timestamp for $x := 1$ strictly between 0 and 2, and 1 is already taken. By requiring the timestamp order to be dense, we ensure that there is always some free timestamp (*e.g.,* 1.5) that T_1 can use.

2.2 Formal Definition

We now define our model for relaxed accesses formally. Let Loc be the set of memory locations, Val be the set of values, and Time be an infinite set of *timestamps*, densely totally ordered by $\leq$, with 0 being the minimum element. A *timemap* is a function $T : \mathsf{Loc} \to \mathsf{Time}$. The order $\leq$ is extended pointwise to timemaps.

Programming Language To keep the presentation abstract, we do not fix a particular programming language; we simply consider each thread i as a transition system with a set of states State_i, initial state $\sigma_i^0 \in \mathsf{State}_i$ and final state $\sigma_i^{\text{final}} \in \mathsf{State}_i$. Intuitively, these states store the values of the local registers and the program counter. Transitions are labeled: the label $\mathtt{R}(x, v)$ correspond to a transition that reads the value v from location x, $\mathtt{W}(x, v)$ denotes a write of the value v in x, while local transitions that do not access the memory are labeled with "Silent". We assume receptiveness of the transition systems (whenever a $\mathtt{R}(x, v)$-transition is possible from a state σ_i, so is a $\mathtt{R}(x, v')$-transition for every value v'), and that they only get stuck in the σ_i^{final}'s.

Messages A *message* m is a tuple $\langle x : v@t \rangle$, where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$ and $t \in \mathsf{Time}$. We denote by $m.\mathtt{loc}$, $m.\mathtt{val}$, and $m.\mathtt{t}$ the components of a message m. Two messages m and m' are called *disjoint*, denoted $m \,\#\, m'$, if $m.\mathtt{loc} \neq m'.\mathtt{loc}$ or $m.\mathtt{t} \neq m'.\mathtt{t}$. Two sets M and M' of messages are called *disjoint*, denoted $M \,\#\, M'$, if $m \,\#\, m'$ for every $m \in M$ and $m' \in M'$.

Memory A *memory* is a pairwise disjoint finite set of messages. A message m may be *(additively) inserted* into memory M if m is disjoint from every message in M. Formally, the additive insertion $M \overset{\text{A}}{\hookleftarrow} m$ is given by $M \cup \{m\}$ and it is only defined if $M \,\#\{m\}$.

Thread States and Configurations A *thread state* is a triple $TS = \langle \sigma, V, P \rangle$, where σ is the thread's local state, V is a timemap representing the thread's *view* of memory, and P is a memory that keeps track of the thread's outstanding promises. We denote by $TS.\mathtt{st}$, $TS.\mathtt{view}$, and $TS.\mathtt{prm}$ the components of a thread state TS. In turn, a *thread configuration* is a pair $\mathbf{TC} = \langle TS, M \rangle$, where TS is a thread state and M is a memory, called the *global memory*. Note that we will always have $TS.\mathtt{prm} \subseteq M$.

Figure 1 shows the five reduction rules for thread configurations. The SILENT rule handles the case when the program performs some local computation that does not affect memory. The READ rule handles the case when the program reads from a location x. The rule nondeterministically selects some message m in the memory, whose timestamp is greater or equal to the recorded one for x in the thread's view, and returns its value; it also updates the thread's view of x to the timestamp of m. The WRITE rule handles the case when the program writes to location x. It extends the memory with a new message for x, whose timestamp t is greater than the one recorded for x in the thread's view, and it updates the thread's view of x to match t. The PROMISE rule extends the memory and the thread's promise set with an arbitrary new message m, whose timestamp is not already present in the memory. (The promise certification is handled separately, as described below.) Finally, the FULFILL rule is similar to the WRITE rule, except that instead of adding a message to the memory, it removes an appropriate message from the thread's promise set P.

We note that the WRITE rule is redundant and have included it to improve readability. Any application of WRITE can be simulated by first promising the appropriate message with the PROMISE rule and then immediately fulfilling the promise with the FULFILL rule.

As we have already mentioned, we have to restrict thread executions so that all promises a thread makes are fulfillable. Thread configurations satisfying this property are called *consistent*. Formally, a thread configuration $\langle TS, M \rangle$ is *consistent* if $\langle TS, M \rangle \to^* \langle TS', M' \rangle$ for some TS' and M' such that $TS'.\mathtt{prm} = \emptyset$. Notice that in the certification of a promise, it is formally possible to make further promises. Since, however, in the end all such promises must be fulfilled, it is useless to make such promises. (A proof of this property is included in our formal development.)

(THREAD: SILENT)
$$\frac{\sigma \xrightarrow{\text{Silent}} \sigma'}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V, P\rangle, M\rangle}$$

(THREAD: READ)
$$\frac{\sigma \xrightarrow{\mathtt{R}(x,v)} \sigma' \qquad \langle x : v@t\rangle \in M \qquad V(x) \le t \qquad V' = V[x \mapsto t]}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V', P\rangle, M\rangle}$$

(THREAD: WRITE)
$$\frac{\sigma \xrightarrow{\mathtt{W}(x,v)} \sigma' \qquad M' = M \overset{\text{A}}{\hookleftarrow} \{\langle x : v@t\rangle\} \qquad V(x) < t \qquad V' = V[x \mapsto t]}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V', P\rangle, M'\rangle}$$

(THREAD: PROMISE)
$$\frac{M' = M \overset{\text{A}}{\hookleftarrow} m \qquad P' = P \overset{\text{A}}{\hookleftarrow} m}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma, V, P'\rangle, M'\rangle}$$

(THREAD: FULFILL)
$$\frac{\sigma \xrightarrow{\mathtt{W}(x,v)} \sigma' \qquad \langle x : v@t\rangle \in P \qquad P' = P \setminus \{\langle x : v@t\rangle\} \qquad V(x) < t \qquad V' = V[x \mapsto t]}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V', P'\rangle, M\rangle}$$

(MACHINE STEP)
$$\frac{\langle \mathcal{TS}(i), M\rangle \to^{+} \langle TS', M'\rangle \qquad \langle TS', M'\rangle \text{ is consistent}}{\langle \mathcal{TS}, M\rangle \to \langle \mathcal{TS}[i \mapsto TS'], M'\rangle}$$

Figure 1. Operational semantics for the simplified model handling only relaxed read and write accesses.

Machine States Finally, a *machine state* $\mathbf{MS} = \langle \mathcal{TS}, M\rangle$ consists of a function $\mathcal{TS}$ assigning a thread state to every thread, and a (global) memory M. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state σ_i^0, a current timestamp of 0 for every location, and an empty set of promises; and the initial memory M^0 that has one initial message $\langle x : 0@0\rangle$ for each location x. A machine takes a step (see the last rule in Figure 1) whenever a thread can take several steps to some *consistent* configuration. Note that we allow multiple thread steps in one machine step. This is convenient in our proofs, and can reduce the amount of certifications during an execution of a program.

We can easily show that a machine can never get stuck except when all threads have terminated (*i.e.,* when each thread has reached $\langle \sigma_i^{\text{final}}, V, \emptyset\rangle$ for some view V). By construction, each thread step preserves consistency of its configuration. It is important to note that the consistency of the configurations of other threads is also preserved, since they can always avoid observing any new messages added to memory. Finally, because of consistency, when a thread has terminated, it must have no promises left.

3. Supporting Atomic Updates

In this section, we extend our basic model for relaxed accesses to also handle (relaxed) *atomic update*—aka *read-modify-write* (RMW)—instructions, such as fetch-and-add and compare-and-swap. Updates are essential as a means to implement synchronization (*e.g.,* mutual exclusion) between threads, but this also makes them tricky to model semantically. In particular, a successful update operation performed by one thread will often have the effect of "winning a race" and hence blocking (previously possible) update operations performed by other "losing" threads. This stands in contrast to the updates-free fragment in §2, in which threads are free to ignore the messages of other threads. Thus, to extend our model to support updates, we must take care to ensure that threads performing updates cannot invalidate the already-certified promises of other threads.

An update is an atomic composition of a read and a write to the same location x. However, unlike under SC, atomicity requires more than just avoiding interference of other threads between the two operations. Consider the following example (taking $\text{FAA}(x, 1)$ to be an atomic *fetch-and-add* of 1 to x, which returns the value of x before the increment):

$$a := \text{FAA}(x, 1); \;\|\; b := \text{FAA}(x, 1); \qquad \text{(Par-Inc)}$$

Atomicity ensures that it is not possible for both threads to increment x from 0 to 1 (we must either get $a = 1$ or $b = 1$). To obtain this, we require that the *read timestamp* of the update (*i.e.,* the timestamp of the write message that the update reads from) immediately precede its *write timestamp* (*i.e.,* the timestamp of the write message that the update generates) in x's modification order, and that future writes to x may not be assigned timestamps in between them. In the example above, if both of the updates were to increment x from 0 to 1, the write timestamp for one of the updates would have to come between the read and write timestamps for the other update.

To enforce this restriction, we extend messages to store a continuous range of timestamps rather than a single timestamp. Thus, messages are now tuples of the form $\langle x : v@(f, t]\rangle$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and $f, t \in \mathsf{Time}$ satisfying $f < t$ or $f = t = 0$. We write $m.\mathtt{from}$ and $m.\mathtt{to}$ to denote the f and t components of a message m. Intuitively, m can be thought of as *reserving* the timestamps in the range $(m.\mathtt{from}, m.\mathtt{to}]$; among these, $m.\mathtt{to}$ is the "real" timestamp of m, but the remaining timestamps in the range are reserved so that other messages cannot use them. Timestamp reservation is reflected in the following revised definition of message disjointness, which enforces that disjoint messages for the same location must have disjoint ranges:

$$\langle x : v@(f, t]\rangle \mathbin{\#} \langle x' : v'@(f', t']\rangle \triangleq$$
$$x \ne x' \;\vee\; t \le f' < t' \;\vee\; t' \le f < t$$

With timestamp reservation, we can easily ensure that the write timestamp of an update is adjacent to its read timestamp in the modification order. Formally, we will say two messages m and m' are *adjacent*, denoted $\mathsf{Adj}(m, m')$, if $m.\mathtt{loc} = m'.\mathtt{loc}$ and $m.\mathtt{to} = m'.\mathtt{from}$. In defining the semantics of updates, we will then insist that the message that the update inserts into memory must appear adjacently after the message that it reads from. This suffices to guarantee the correct outcome in the Par-Inc program above.

Although the introduction of timestamp reservation enables us to easily model updates, it creates a complication for promises, namely that timestamp reservations may invalidate the promise certifications already performed by other threads. Consider, for example, the following program:

$$\begin{array}{l} a := x; \;\color{teal}{/\!/\, 1} \\ b := \text{FAA}(z, 1); \;\color{teal}{/\!/\, 0} \\ y := b + 1; \end{array} \;\left\|\; x := y; \;\right\|\; \text{FAA}(z, 1); \qquad \text{(Upd-Stuck)}$$

This behavior ought to be allowed, since hardware could reorder the read of x after the independent accesses to z and y. To produce this behavior, following our semantics from the previous section, T_1 could promise to write $y := 1$ because it can thread-locally certify that the promise can be fulfilled (the certification will involve updating z from 0 to 1). If, however, T_3 then updates z from 0 to 1, that will mean that T_1 can no longer perform the update it needs to fulfill its promise, and its execution will eventually get stuck.

To avoid such stuck executions, we strengthen the check performed by promise certification, *i.e.,* the consistency requirement on thread configurations. We require that each thread's promises are locally fulfillable not only in the current memory, but also *in any future memory*, *i.e.,* any extension of the memory with additional messages. This quantification over future memories ensures that thread configurations remain consistent whenever another thread performs an execution step, and thus the machine cannot get stuck.

Returning to the above example, T_1 will not be permitted to promise to write $y := 1$ in the initial state, precisely because that

(THREAD: FULFILL UPDATE)

$$\frac{\begin{array}{c}\sigma \xrightarrow{\mathtt{U}(x,v_\mathrm{r},v_\mathrm{w})} \sigma' \qquad \langle x : v_\mathrm{r}@(f_\mathrm{r}, t_\mathrm{r}]\rangle \in M \\ m_\mathrm{w} = \langle x : v_\mathrm{w}@(t_\mathrm{r}, t_\mathrm{w}]\rangle \qquad m_\mathrm{w} \in P \qquad P' = P \setminus \{m_\mathrm{w}\} \\ V(x) \le t_\mathrm{r} \qquad V' = V[x \mapsto t_\mathrm{w}]\end{array}}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V', P'\rangle, M\rangle}$$

Figure 2. Additional rule for updates (all other rules are as before except all messages $\langle x : v@t\rangle$ are replaced by $\langle x : v@(f, t]\rangle$).

promise could not be fulfilled under an arbitrary future memory (*e.g.,* one containing the update of T_3, as we showed). T_1 may, however, first promise $\langle z : 1@(0, 1]\rangle$, reserving the time range from the initialization of z up to its increment. T_1 can fulfill that promise, because no future extension of the memory will be able to add any messages in between. After making that promise, T_1 may then promise, *e.g.,* $\langle y : 1@(3, 4]\rangle$, which it can now fulfill under any extension of the memory. With these promises in place, T_3 will be prevented from updating z from 0 to 1; it will be forced to update z from 1 to 2, which will not block the future execution of T_1.

Our quantification here over *all* future memories may seem rather restrictive in that it completely ignores what can or cannot happen in a particular program. That said, we find it a simple and natural way of ensuring "thread locality". The latter is a guiding principle in our semantics, according to which the set of actions a thread can take is determined only by the current memory and its own state.

Formally, we say that M_future is a *future memory* of M if $M_\text{future} = M \xleftrightarrow{A} m_1 \xleftrightarrow{A} \dots \xleftrightarrow{A} m_n$ for some $n \ge 0$ and messages $m_1, \dots, m_n$. And we now say a thread configuration $\langle TS, M\rangle$ is *consistent* if, for every future memory M_future of M, there exist TS' and M' such that $\langle TS, M_\text{future}\rangle \to^* \langle TS', M'\rangle$ and $TS'.\mathtt{prm} = \emptyset$.

Finally, we extend the operational semantics for thread configurations with one additional rule for update fulfillment shown in Figure 2. This rule forces its write to be adjacent in modification order to its read. As with plain writes, a normal (non-promised) update step can be simulated by a promise step immediately followed by fulfillment. Note that the other rules remain exactly the same; they simply ignore the $m.\mathtt{from}$ component of messages m.

4. Full Model

In this section, we extend the basic model of §2-3 to handle all the features of the C++ concurrency model except SC accesses and consume reads.

4.1 Release/Acquire Synchronization

Release/Acquire Fences A crucial feature of the C++ model is the ability for threads to synchronize using memory fences or stronger kinds of atomic accesses. Consider the message-passing test case:

$$\begin{array}{l||l} x := 1; & a := y; \ /\!/ 1 \\ \textbf{fence-rel}; & \textbf{fence-acq}; \\ y := 1; & b := x; \ /\!/ {\neq 0} \end{array} \qquad \text{(MP+fences)}$$

The release fence between the writes, together with the acquire fence between the reads, prevents the weak behavior of the example (*i.e.,* that of returning $a = 1$ and $b = 0$). Roughly speaking, the C++ model forbids this behavior by requiring that whenever a read before an acquire fence reads from a write after a release fence, the two fences synchronize, which in turn means that any write that happens-before the release fence must be visible to any read that happens-after the acquire fence. So, if T_2 reads $y = 1$, then after the acquire fence it *must* read $x = 1$.

To implement this semantics, we extend our model in two ways.

First, we refine each thread's view. Rather than having a single view of which messages it has observed, a thread now has three views: $\mathcal{V} = \langle cur, acq, rel\rangle$. We denote by $\mathcal{V}.\mathtt{cur}$, $\mathcal{V}.\mathtt{acq}$ and $\mathcal{V}.\mathtt{rel}$ the components of a thread view $\mathcal{V}$. A thread's *current view*, $\mathtt{cur}$, is as before: it records which messages the thread has observed and restricts which messages a read may return and a write may create. Its *release view*, $\mathtt{rel}$, records what the thread's $\mathtt{cur}$ view *was* at the point of its last release fence. Dually, its *acquire view*, $\mathtt{acq}$, records what the thread's $\mathtt{cur}$ view *will become* if it performs an acquire fence. Consequently, the views are related as follows: $\mathtt{rel} \le \mathtt{cur} \le \mathtt{acq}$.

Second, we extend write messages to additionally record the release view of the writing thread at the point when the write occurred. Thus, a message now takes the form $m = \langle x : v@(f, t], R\rangle$, where x, v, f, t are as before, and R is the *message view*, satisfying $R(x) \le t$. We write $m.\mathtt{view}$ to mean the message view R of m.

During execution of relaxed accesses, a thread's views drift apart. When a thread reads a message, it incorporates the message's view into the thread's $\mathtt{acq}$ view, but not into its $\mathtt{cur}$ or $\mathtt{rel}$ views. When a thread writes a message, it uses the thread's $\mathtt{rel}$ view as the basis for the message's view, but only incorporates the message itself into the thread's $\mathtt{cur}$ and $\mathtt{acq}$ views, not its $\mathtt{rel}$ view.

Fence commands bring these diverging views closer to one another. Specifically, an acquire fence increases the thread's $\mathtt{cur}$ view to match its $\mathtt{acq}$ view, thereby ensuring that the thread is up to date with respect to views of all the messages read before the fence. Symmetrically, a release fence increases the thread's $\mathtt{rel}$ view to match its $\mathtt{cur}$ view, thereby ensuring that the views of all messages the thread writes after the release fence will contain the messages observed before the fence.

Returning to the MP+fences program, suppose that T_1 emitted messages $\langle x : 1@(\_, t_x], \_\rangle$ and $\langle y : 1@(\_, t_y], R_y\rangle$. Then, T_1's $\mathtt{cur}$ view before the release fence is $[x@t_x, y@0]$. The fence then updates T_1's $\mathtt{rel}$ view to match its $\mathtt{cur}$ view, so that the subsequent message will carry the view $R_y = [x@t_x, y@0]$. (Without the release fence, we would have $R_y = [x@0, y@0]$.) On T_2's side, the read of $y = 1$ updates T_2's $\mathtt{cur}$ view to $[x@0, y@t_y]$, and its $\mathtt{acq}$ view to $[x@t_x, y@t_y]$. The acquire fence then updates T_2's $\mathtt{cur}$ view to match its $\mathtt{acq}$ view, and hence the subsequent read of x must see the $x := 1$ write. If either the release or the acquire fence were missing, then T_2's $\mathtt{cur}$ view at the read of x would have been $[x@0, y@t_y]$, allowing it to read $x = 0$.

Interaction with Promises Promises (like every other message) now carry a view, and threads reading a promise are subject to the same constraints as if they were reading a usual message. In particular, after reading a promise and performing an acquire fence, a thread can only read messages with timestamp greater or equal to the view carried in the message. In order to avoid cases where execution gets stuck, we must ensure that *some* message can be read for every location. Thus we require that the view attached to promises is an existing view (includes only timestamps of messages in the memory) at the time the promise was made.

Going back to MP+fences, note that T_1 cannot promise $y := 1$ before performing $x := 1$. Indeed, at that stage the only existing message for x in the memory is the initial one, but because of the release fence, the view in the $y := 1$ message must include the message that will be produced for the $x := 1$ assignment. Hence, release fences practically serve also as barriers for promises. We find it convenient to explicitly require this in our semantics, and we require that the set of promises of the executing thread is empty, whenever a release fence is performed. Recall that the main reason for introducing promises was to allow read-write reorderings, such as in the LB example of the introduction. However, if there is a release fence in between, then the reordering is no longer possible, and thus our motivation for promising the write is void.

Release/Acquire Accesses A more fine-grained way of achieving synchronization besides the release and acquire fences in C++ are *acquire reads* and *release writes*. Intuitively speaking, an acquire read is a relaxed read followed by an acquire fence, while a release write is a release fence followed by a relaxed write, with the restriction that these fences induce synchronization *only* on the location of the access. For example, in the following program,[1] only the second thread synchronizes with the first one.

$$\begin{array}{l||l||l} x := 1; & & \\ y_{\mathsf{rel}} := 1; & a := y_{\mathsf{acq}};\ /\!/\ 1 & c := z_{\mathsf{acq}};\ /\!/\ 1 \\ & b := x;\ \ /\!/\!\neq 0 & d := x;\ /\!/\ 0 \\ z := 1; & & \end{array}$$

Hence, b must get the value 1, while d may get 0.

To model these accesses, we treat the `rel` view of each thread not as a single view, but rather as one separate view per location, recording the thread's current view at the latest release fence or release write to that location. Performing a release write to location x increments the release view of x to match the *cur* view, while a release fence increment the release views of *all* locations. In addition, a release write to x attaches its new release view of x to the message. Performing an acquire read, then, increments the thread's current view to include the message's view.

In the example above, at the end of T_1's execution, its thread view has $\mathtt{rel}(y) = [x@t_x, y@t_y, z@0]$, whereas $\mathtt{rel}(z) = [x@0, y@0, z@0]$. As a result, the y_{acq} read increases T_2's `cur` view to $[x@t_x, y@t_y, z@0]$, which forces it to then read $x = 1$, whereas the z_{acq} read increases T_3's `cur` view to $[x@0, y@0, z@t_z]$, which allows it to later read $x = 0$.

Release Sequences Using the per-location release views, we can straightforwardly handle C++-style release sequences (following the definition of release sequences given in [28]). In C++, an acquire read synchronizes with a release write w to x not only if it reads from w but also if it reads from a write in w's *release sequence*. The release sequence of w is inductively defined to include all the same-thread writes/updates to x after w, as well as all updates reading from an event in the release sequence of w. For example, in the following program, the y_{acq} synchronizes with the $y_{\mathsf{rel}} := 1$ because it reads from the FAA$(y, 1)$, which in turn reads from the $y := 2$.

$$\begin{array}{l||l||l} x := 1; & & \\ y_{\mathsf{rel}} := 1; & \mathrm{FAA}(y,1); & a := y_{\mathsf{acq}};\ /\!/\ 3 \\ y := 2; & & b := x;\ \ /\!/\!\neq 0 \end{array}$$

Our operational semantics already handles the case of reading from a later write of the same thread, because the thread's release view for y is included in the message's view. To handle the updates that read from elements of the release sequence, we insist that the view of the write message of an update must incorporate the view of the read message of the update. Thus, in this example, the views of all the y messages contain $x@t_x$, and hence T_3 must read $x = 1$.

Promises over Release/Acquire Accesses We finally point out another delicate issue related to the interaction between promises and release/acquire accesses. Consider the following variants of the LB example:

$$\begin{array}{l||l} a := x;\ /\!/\!\neq 1 & x := y; \\ y_{\mathsf{rel}} := 1; & \end{array}\ \ (\mathrm{LBr}) \qquad \begin{array}{l||l} a := x_{\mathsf{acq}};\ /\!/\ 1 & x := y; \\ y := 1; & \end{array}\ \ (\mathrm{LBa})$$

In the first variant (LBr), the promise of $y_{\mathsf{rel}} := 1$ should be forbidden for the same reason that a promise over a release fence is forbidden, and hence the specified behavior is disallowed. We note that this behavior is possible under the C++ model, but is not possible under the usual compilation of release/acquire accesses to Power and ARM (using a `lwsync/dmb_sy` fence in the first thread).[2] More generally, our model forbids promises over release accesses to the same location.

In the second variant (LBa), we allow the promise of $y := 1$ and thus the $a = 1$ outcome. The reason is that we want to enable optimizations that result in the elimination of an acquire read and thus remove the reordering constraints of the acquire. Consider, for example, the following program transformation:

$$\begin{array}{l||l} a := x;\ /\!/\ 2 & \\ y := 1; & x := y; \\ b := y_{\mathsf{acq}}; & \\ y := 2; & \end{array} \ \rightsquigarrow\ \begin{array}{l||l} y := 1; & \\ b := 1; & x := y; \\ y := 2; & \\ a := x;\ /\!/\ 2 & \end{array}\quad (\mathrm{LBa}')$$

which may in effect reorder the $y := 2$ write before the $a := x$ read even though there is an acquire read in between (by first replacing y_{acq} with 1 and then reordering $a := x$ past both writes to y). Thus, our semantics has to allow promises over acquire actions. Note that there is no need to do so for release writes, because release writes cannot simply be eliminated in this way.

4.2 Sequentially Consistent (SC) Fences

We extend the model with SC fences, whose purpose is to allow the programmer to enforce strong ordering guarantees among memory accesses. In particular, one would expect that full sequential consistency is restored if an SC fence is placed between every two shared memory accesses of a program.[3]

To handle SC fences, we extend our machine state with a *global* timemap $\mathcal{S}$, which records the latest messages written by any thread before an SC fence. When a thread T executes an SC fence, in addition to the effect of both an acquire and a release fence, T increases both its `cur` view and the global timemap to the maximum of the two. Consider the following variant of the SB example:

$$\begin{array}{l||l} x := 1; & y := 1; \\ \textbf{fence-sc}; & \textbf{fence-sc}; \\ a := y;\ /\!/\ 0 & b := x;\ \ /\!/\!\neq 0 \end{array}\qquad (\mathrm{SB{+}fences})$$

Here, the current views of the two threads just before their SC fences are $[x@t_x, y@0]$ and $[x@0, y@t_y]$, respectively, while the global view is $[x@0, y@0]$. If the fence of T_1 is executed first, it will update $\mathcal{S}$ to $[x@t_x, y@0]$. So, when the fence of T_2 is executed, both its `cur` view and $\mathcal{S}$ become $[x@t_x, y@t_y]$, from which point onwards T_2 *must* read $x = 1$.

4.3 "Plain" Non-Synchronizing Accesses

Both C++ and Java provide some form of *non-synchronizing* accesses, *i.e.,* accesses that are meant to be used only for non-racy data accesses (C++'s non-atomic accesses and Java's normal accesses). Such accesses can never achieve synchronization, even together with fences. Consequently, compilers are free to reorder non-synchronizing reads across acquire fences, and to reorder release fences across non-synchronizing writes. These non-synchronizing accesses, which we refer to as *plain* accesses, are easily supported in our model. The difference from relaxed accesses is simple: a plain read from a message m should not incorporate m`.view` into the thread's `acq` view; and a message m produced by a plain write should only carry the 0-view (*i.e.,* $\perp$ in the lattice of views). Moreover, plain writes can be promised even beyond a release fence or a release write to the same location.

[1] In this and in following code snippets, we annotate non-relaxed accesses with their access mode; all non-annotated accesses are assumed to be relaxed.

[2] Moreover, we observe that even the C++ model forbids this outcome, if we additionally make the read of y in the second thread into a consume read (which is supposed to be compiled exactly as a relaxed read, but preserving syntactic dependencies).

[3] In this regard, our semantics is stronger than the C++ model [6], which fails to validate this basic property, and follows Lahav *et al.* [17, 19] instead.

Besides the reordering mentioned above, compilers can (and do) utilize further the assumption that some accesses are intended to be non-racy. Indeed, assuming two non-racy reads, a compiler may reorder them even if they are reading *the same* location. In a broader context, it may pave the way to further optimizations (*e.g.,* a compiler may like to unconditionally optimize $a := x; b := *p; c := x$ to $b := *p; a := x; c := a$, without the burden of analyzing whether the pointer p points to x or not). Since we followed C++'s assumption of full per-location coherence for our relaxed accesses, the reordering of two reads from the same location is unsound for them. Concretely, consider the following example:

$$\begin{array}{l\|l} x := 1; & a := x; \ /\!\!/ 2 \\ x := 2; & b := x; \ /\!\!/ 1 \end{array} \quad \rightsquigarrow \quad \begin{array}{l\|l} x := 1; & b := x; \ /\!\!/ 1 \\ x := 2; & a := x; \ /\!\!/ 2 \end{array}$$

The target program obviously allows the specified behavior, while the source does not. Fortunately, it is not hard to adapt our plain accesses to provide only partial per-location coherence (in C++11 terms, dropping "coherence-RR" for plain accesses), consequently allowing this reordering. The idea is to extend the notion of a view V—both message views R and the three component views of a thread ($\mathtt{cur}, \mathtt{acq}, \mathtt{rel}$)—from being a single timemap to a pair of timemaps: a "normal" one ($V.\mathtt{rlx}$) as before, and one for plain accesses ($V.\mathtt{pln}$). The $V.\mathtt{pln}$ timemap is generally smaller than the normal timemap ($V.\mathtt{pln} \le V.\mathtt{rlx}$), and restricts the possible timestamps available to plain reads. A plain read from a message m with location x and time t only consults this new timemap, checking that $\mathtt{cur}.\mathtt{pln}(x) \le t$, and only updates $\mathtt{cur}.\mathtt{rlx}(x)$ to include t. A plain write, on the other hand, cannot pick a timestamp smaller than $\mathtt{cur}.\mathtt{rlx}(x)$ (since we do maintain the coherence properties besides "coherence-RR").

Importantly, we do not exploit "catch-fire" semantics (à la C++) to accommodate our plain accesses, but rather give a well-defined semantics to arbitrary racy programs. In addition, we note that it is easy to decouple the two weaknesses of plain accesses compared to relaxed ones by introducing a middle access mode that allows synchronization (together with release and acquire fences), but supports only partial per-location coherence.

Remark 1. Our model handles only hardware-atomic memory accesses. To handle non-atomic reads/writes, such as Java double and C struct accesses, our semantics could be extended by introducing "garbage values" (LLVM-style undefined values [2]) as in [8].

4.4 System Calls

For the purpose of defining the behaviors of programs (as needed to prove soundness of transformations), we augment our language and semantics with system calls labeled with "SysCall(v)". These are operations that are visible to an external observer (*e.g.,* printing statements). For simplicity, we assume that these take one value (input or output), and more importantly, that they do not access the memory, and serve as the strongest barrier for reordering. Thus, we simply model system calls as SC fences.

4.5 Modifying Existing Promises

So far, our model does not allow promises, once made, to be changed. However, our full model does allow two forms of promise adjustment, both of which are defined in such a way that threads that have already read from the promised message are unaffected.

Split The first form of promise adjustment is *splitting*. Consider the following example:

$$\left.\begin{array}{l} a := x; \ /\!\!/ 2 \\ \textbf{if } a = 2 \textbf{ then } \mathrm{FAA}(y,1); \mathrm{FAA}(y,1); \ \textbf{else } \mathrm{FAA}(y,2); \end{array}\right\| \ x := y;$$

We find it natural to allow the specified behavior, as it can be obtained by benign compiler optimizations: first $\mathrm{FAA}(y,1); \mathrm{FAA}(y,1)$ can be merged to $\mathrm{FAA}(y,2)$, and then the whole if-then-else statement can be replaced by $\mathrm{FAA}(y,2)$. Nevertheless, the model described so far forbids this behavior. Indeed, clearly, an execution obtaining this behavior must start with T_1 promising a message of the form $\langle y : 2@(f,t]\rangle$. Since certification is needed for any future memory, it must take $f = 0$ (or else, it cannot fulfill its promise for a memory that includes, say, $\langle y : 42@(0,5]\rangle$). Then, T_2 can read the promise and add a message of the form $\langle x : 2@(\_,t_x]\rangle$ to the memory. Now, T_1 would like to read this message. However, if it does so, it will not be able to fulfill its promise $\langle y : 2@(0,t]\rangle$, simply because there is no available timestamp interval in which it can put the first $y = 1$ message. To solve this, we allow threads to split their own promises in two pieces, keeping the original promise with the same $m.\mathtt{to}$ value. For the example above, T_1 could proceed by splitting its promise $\langle y : 2@(0,t]\rangle$ into $\langle y : 1@(0,t/2]\rangle$ and $\langle y : 2@(t/2,t]\rangle$, reading the message $\langle x : 2@(\_,t_x]\rangle$ and fulfilling both promises.

Lower The second form of promise adjustment is *lowering* of the promised message's view. Note that by promising a message carrying a high view, a thread places *more* restrictions on the readers of that promise. Thus, changing the view of a promise m to a view $R' \le m.\mathtt{view}$ can never cause any harm. Technically, including this option simplifies our simulation arguments used to prove the soundness of program transformations, by allowing us to have a simpler simulation relation between the source and target memories. More generally speaking, it allows us to prove and use the following natural property: if all the views included in some machine state $\mathbf{MS}$ (in its memory's messages and its thread's views) are less than or equal to all views in another machine state $\mathbf{MS}'$, then every behavior of $\mathbf{MS}'$ is also a behavior of $\mathbf{MS}$.

4.6 Formal Model

Finally, we formally present our full model, combining and making precise all the ideas outlined above. The model employs three modes for memory accesses, naturally ordered as follows:

$$\mathtt{pln} \ \sqsubset \ \mathtt{rlx} \ \sqsubset \ \mathtt{ra}$$

We use o as a metavariable for access mode. The programming language is modeled by a transition system whose transition labels (see §2.2) are: "Silent" for local transitions; $\mathtt{R}(o,x,v)$ for reads; $\mathtt{W}(o,x,v)$ for writes; $\mathtt{U}(o_{\mathrm{r}}, o_{\mathrm{w}}, x, v_{\mathrm{r}}, v_{\mathrm{w}})$ for updates; $\mathtt{F}_{\mathtt{acq}}, \mathtt{F}_{\mathtt{rel}}, \mathtt{F}_{\mathtt{sc}}$ for fences; and SysCall(v) for system calls. Note that updates have two access modes, one for the read and one for the write; and that only fences may have the $\mathtt{sc}$ mode.

View A *view* is a pair $V = \langle T_{\mathtt{pln}}, T_{\mathtt{rlx}}\rangle$ of timemaps (see §2.2) satisfying $T_{\mathtt{pln}} \le T_{\mathtt{rlx}}$. We denote by $V.\mathtt{pln}$ and $V.\mathtt{rlx}$ the components of V. View denotes the set of all views.

Messages A *message* m is a tuple $\langle x : v@(f,t], R\rangle$, where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $f, t \in \mathsf{Time}$, and $R \in \mathsf{View}$, satisfying $f < t$ (unless $f = t = 0$) and $R.\mathtt{rlx}(x) \le t$. We denote by $m.\mathtt{loc}$, $m.\mathtt{val}$, $m.\mathtt{from}$, $m.\mathtt{to}$, and $m.\mathtt{view}$ the components of m.

Memory A *memory* is a (nonempty) pairwise disjoint finite set of messages (see §3 for def. of disjointness). A memory M supports the following insertions of a message $m = \langle x : v@(f,t], R\rangle$:

- The *additive insertion*, denoted by $M \overset{\mathrm{A}}{\hookleftarrow} m$, is only defined if $\{m\} \# M$, in which case it is given by $\{m\} \cup M$.
- The *splitting insertion*, denoted by $M \overset{\mathrm{S}}{\hookleftarrow} m$, is only defined if there exists $m' = \langle x : v'@(f,t'], R'\rangle$ with $t < t'$ in M, in which case it is given by $M \backslash \{m'\} \cup \{m, \langle x : v'@(t,t'], R'\rangle\}$.
- The *lowering insertion*, denoted by $M \overset{\mathrm{U}}{\hookleftarrow} m$, is only defined if there exists $m' = \langle x : v@(f,t], R'\rangle$ with $R \le R'$ in M, in which case it is given by $M \backslash \{m'\} \cup \{m\}$.

We write $M(x)$ for the sub-memory $\{ m \in M \mid m.\mathtt{loc} = x \}$.

(MEMORY: NEW)

$$\dfrac{}{\langle P, M\rangle \xrightarrow{m} \langle P, M \overset{\mathrm{A}}{\hookleftarrow} m\rangle}$$

(MEMORY: FULFILL)

$$\dfrac{\hookleftarrow \in \{\overset{\mathrm{S}}{\hookleftarrow}, \overset{\mathrm{U}}{\hookleftarrow}\} \qquad P' = P \hookleftarrow m \qquad M' = M \hookleftarrow m}{\langle P, M\rangle \xrightarrow{m} \langle P' \setminus \{m\}, M'\rangle}$$

(READ-HELPER)

$$\dfrac{\begin{array}{c} o = \mathtt{pln} \implies \mathit{cur}.\mathtt{pln}(x) \le t \\ o \in \{\mathtt{rlx}, \mathtt{ra}\} \implies \mathit{cur}.\mathtt{rlx}(x) \le t \\ \mathit{cur}' = \mathit{cur} \sqcup V \sqcup (o \sqsupseteq \mathtt{ra}\ ?\ R) \\ \mathit{acq}' = \mathit{acq} \sqcup V \sqcup (o \sqsupseteq \mathtt{rlx}\ ?\ R) \\ \textit{where } V = [\mathtt{pln} : (o \sqsupseteq \mathtt{rlx}\ ?\ \{x@t\}), \mathtt{rlx} : \{x@t\}] \end{array}}{\langle \mathit{cur}, \mathit{acq}, \mathit{rel}\rangle \xrightarrow{\mathtt{R}:o,x,t,R} \langle \mathit{cur}', \mathit{acq}', \mathit{rel}\rangle}$$

(WRITE-HELPER)

$$\dfrac{\begin{array}{c} \mathit{cur}.\mathtt{rlx}(x) < t \\ \mathit{cur}' = \mathit{cur} \sqcup V \qquad \mathit{acq}' = \mathit{acq} \sqcup \mathit{cur}' \\ \mathit{rel}' = \mathit{rel}[x \mapsto \mathit{rel}(x) \sqcup V \sqcup (o \sqsupseteq \mathtt{ra}\ ?\ \mathit{cur}')] \\ R_{\mathrm{w}} = (o \sqsupseteq \mathtt{rlx}\ ?\ (\mathit{rel}'(x) \sqcup R_{\mathrm{r}})) \\ \textit{where } V = [\mathtt{pln} : \{x@t\}, \mathtt{rlx} : \{x@t\}] \end{array}}{\langle \mathit{cur}, \mathit{acq}, \mathit{rel}\rangle \xrightarrow{\mathtt{W}:o,x,t,R_{\mathrm{r}},R_{\mathrm{w}}} \langle \mathit{cur}', \mathit{acq}', \mathit{rel}'\rangle}$$

(SC-FENCE-HELPER)

$$\dfrac{\begin{array}{c} \mathcal{S}' = \mathit{acq}.\mathtt{rlx} \sqcup \mathcal{S} \\ \mathit{cur}' = \mathit{acq}' = \langle \mathcal{S}', \mathcal{S}'\rangle \\ \mathit{rel}' = \lambda\_.\langle \mathcal{S}', \mathcal{S}'\rangle \end{array}}{\begin{array}{c}\langle\langle \mathit{cur}, \mathit{acq}, \mathit{rel}\rangle, \mathcal{S}\rangle \xrightarrow{\mathtt{F_{sc}}} \\ \langle\langle \mathit{cur}', \mathit{acq}', \mathit{rel}'\rangle, \mathcal{S}'\rangle\end{array}}$$

(READ)

$$\dfrac{\begin{array}{c} \sigma \xrightarrow{\mathtt{R}(o,x,v)} \sigma' \\ \langle x : v@(\_, t], R\rangle \in M \\ \mathcal{V} \xrightarrow{\mathtt{R}:o,x,t,R} \mathcal{V}' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}', P\rangle, \mathcal{S}, M\rangle}$$

(WRITE)

$$\dfrac{\begin{array}{c} \sigma \xrightarrow{\mathtt{W}(o,x,v)} \sigma' \\ o = \mathtt{ra} \implies \forall m' \in P(x).\ m'.\mathtt{view} = \bot \\ m = \langle x : v@(\_, t], R\rangle \\ \langle P, M\rangle \xrightarrow{m} \langle P', M'\rangle \\ \mathcal{V} \xrightarrow{\mathtt{W}:o,x,t,\bot,R} \mathcal{V}' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}', P'\rangle, \mathcal{S}, M'\rangle}$$

(UPDATE)

$$\dfrac{\begin{array}{c} \sigma \xrightarrow{\mathtt{U}(o_{\mathrm{r}},o_{\mathrm{w}},x,v_{\mathrm{r}},v_{\mathrm{w}})} \sigma' \\ o_{\mathrm{w}} = \mathtt{ra} \implies \forall m' \in P(x).\ m'.\mathtt{view} = \bot \\ \langle x : v_{\mathrm{r}}\ @(\_, t_{\mathrm{r}}\ ], R_{\mathrm{r}}\ \rangle \in M \\ m_{\mathrm{w}} = \langle x : v_{\mathrm{w}}@(t_{\mathrm{r}}, t_{\mathrm{w}}], R_{\mathrm{w}}\rangle \\ \langle P, M\rangle \xrightarrow{m_{\mathrm{w}}} \langle P', M'\rangle \\ \mathcal{V} \xrightarrow{\mathtt{R}:o_{\mathrm{r}},x,t_{\mathrm{r}},R_{\mathrm{r}}} \xrightarrow{\mathtt{W}:o_{\mathrm{w}},x,t_{\mathrm{w}},R_{\mathrm{r}},R_{\mathrm{w}}} \mathcal{V}' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}', P'\rangle, \mathcal{S}, M'\rangle}$$

(ACQ-FENCE)

$$\dfrac{\sigma \xrightarrow{\mathtt{F_{acq}}} \sigma' \qquad \mathit{cur}' = \mathit{acq}}{\begin{array}{c}\langle\langle \sigma, \langle \mathit{cur}, \mathit{acq}, \mathit{rel}\rangle, P\rangle, \mathcal{S}, M\rangle \to \\ \langle\langle \sigma', \langle\langle \mathit{cur}', \mathit{acq}, \mathit{rel}\rangle, P\rangle, \mathcal{S}, M\rangle\rangle\end{array}}$$

(REL-FENCE)

$$\dfrac{\begin{array}{c} \sigma \xrightarrow{\mathtt{F_{rel}}} \sigma' \qquad \mathit{rel}' = \lambda\_.\mathit{cur} \\ \forall m \in P.\ m.\mathtt{view} = \bot \end{array}}{\begin{array}{c}\langle\langle \sigma, \langle \mathit{cur}, \mathit{acq}, \mathit{rel}\rangle, P\rangle, \mathcal{S}, M\rangle \to \\ \langle\langle \sigma', \langle\langle \mathit{cur}, \mathit{acq}, \mathit{rel}'\rangle, P\rangle, \mathcal{S}, M\rangle\rangle\end{array}}$$

(SC-FENCE)

$$\dfrac{\begin{array}{c} \sigma \xrightarrow{\mathtt{F_{sc}}} \sigma' \\ \langle \mathcal{V}, \mathcal{S}\rangle \xrightarrow{\mathtt{F_{sc}}} \langle \mathcal{V}', \mathcal{S}'\rangle \\ \forall m \in P.\ m.\mathtt{view} = \bot \end{array}}{\begin{array}{c}\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \\ \langle\langle \sigma', \mathcal{V}', P\rangle, \mathcal{S}', M\rangle\end{array}}$$

(SYSTEM CALL)

$$\dfrac{\begin{array}{c} \sigma \xrightarrow{\mathrm{SysCall}(v)} \sigma' \\ \langle \mathcal{V}, \mathcal{S}\rangle \xrightarrow{\mathtt{F_{sc}}} \langle \mathcal{V}', \mathcal{S}'\rangle \\ \forall m \in P.\ m.\mathtt{view} = \bot \end{array}}{\begin{array}{c}\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \xrightarrow{\mathrm{SysCall}(v)} \\ \langle\langle \sigma', \mathcal{V}', P\rangle, \mathcal{S}', M\rangle\end{array}}$$

(SILENT)

$$\dfrac{\sigma \xrightarrow{\mathrm{Silent}} \sigma'}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}, P\rangle, \mathcal{S}, M\rangle}$$

(PROMISE)

$$\dfrac{\begin{array}{c} \hookleftarrow \in \{\overset{\mathrm{A}}{\hookleftarrow}, \overset{\mathrm{S}}{\hookleftarrow}, \overset{\mathrm{U}}{\hookleftarrow}\} \qquad P' = P \hookleftarrow m \\ M' = M \hookleftarrow m \qquad m.\mathtt{view} \in M' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma, \mathcal{V}, P'\rangle, \mathcal{S}, M'\rangle}$$

(MACHINE STEP)

$$\dfrac{\begin{array}{c} \langle \mathcal{TS}(i), \mathcal{S}, M\rangle \to^* \langle TS', \mathcal{S}', M'\rangle \\ \langle TS', \mathcal{S}', M'\rangle \xrightarrow{e} \langle TS'', \mathcal{S}'', M''\rangle \\ \langle TS'', \mathcal{S}'', M''\rangle \text{ is consistent} \end{array}}{\langle \mathcal{TS}, \mathcal{S}, M\rangle \xrightarrow{e} \langle \mathcal{TS}[i \mapsto TS''], \mathcal{S}'', M''\rangle}$$

Figure 3. Full operational semantics.

Closed Memory Given a timemap T and a memory M, we write $T \in M$ if, for every $x \in \mathsf{Loc}$, we have $T(x) = m.\mathtt{to}$ for some $m \in M$ with $m.\mathtt{loc} = x$. For a view V, we write $V \in M$ if $T \in M$ for each component timemap T of V. A memory M is *closed* if $m.\mathtt{view} \in M$ for every $m \in M$.

Future Memory For memories M, M', we write $M \to M'$ if $M' \in \{M \overset{\mathrm{A}}{\hookleftarrow} m, M \overset{\mathrm{S}}{\hookleftarrow} m, M \overset{\mathrm{U}}{\hookleftarrow} m\}$ for some message m, and M' is closed. We say M' is a *future memory* of M w.r.t. a memory P, if M' is closed, $P \subseteq M'$, and $M \to^* M'$.

Threads A *thread view* is a triple $\mathcal{V} = \langle \mathit{cur}, \mathit{acq}, \mathit{rel}\rangle$, where $\mathit{cur}, \mathit{acq} \in \mathsf{View}$ and $\mathit{rel} \in \mathsf{Loc} \to \mathsf{View}$ satisfying $\mathit{rel}(x) \le \mathit{cur} \le \mathit{acq}$ for all $x \in \mathsf{Loc}$. We denote by $\mathcal{V}.\mathtt{cur}$, $\mathcal{V}.\mathtt{acq}$ and $\mathcal{V}.\mathtt{rel}$ the components of $\mathcal{V}$. A *thread state* is a triple $TS = \langle \sigma, \mathcal{V}, P\rangle$ defined just as in §2.2 except with a thread view $\mathcal{V}$ instead of a single timemap (σ is a local state and P is a memory). We denote by $TS.\mathtt{st}$, $TS.\mathtt{view}$ and $TS.\mathtt{prm}$ the components of TS.

Thread Configuration Steps A *thread configuration* is a triple $\langle TS, \mathcal{S}, M\rangle$, where TS is a thread state, $\mathcal{S}$ is a timemap (the global SC timemap), and M is a memory.

Figure 3 presents the full list of thread configuration steps. To avoid repetition we use the additional rules READ-HELPER, WRITE-HELPER, and SC-FENCE-HELPER. These employ several helpful notations: $\bot$ and $\sqcup$ denote the natural bottom elements and join operations for timemaps and for views (pointwise extensions of the initial timestamp 0 and the $\sqcup$—*i.e.,* max—operation on timestamps); $\{x@t\}$ denotes the timemap assigning t to x and 0 to other locations; and $(\mathit{cond}\ ?\ X)$ is defined to be X if cond holds, and $\bot$ otherwise.

The write and the update steps cover two cases: a fresh write to memory (MEMORY:NEW) and a fulfillment of an outstanding promise (MEMORY:FULFILL). The latter allows to split the promise or lower its view before its fulfillment (note that when $m \in P \subseteq M$, we have $P = P \overset{\mathrm{U}}{\hookleftarrow} m$ and $M = M \overset{\mathrm{U}}{\hookleftarrow} m$ by def. of $\overset{\mathrm{U}}{\hookleftarrow}$).

Consistency A thread configuration $\langle TS, \mathcal{S}, M\rangle$ is called *consistent* if for every future memory M_{future} of M w.r.t. $TS.\mathtt{prm}$ and every timemap $\mathcal{S}_{\mathrm{future}}$ with $\mathcal{S} \le \mathcal{S}_{\mathrm{future}} \in M_{\mathrm{future}}$, there exist TS', $\mathcal{S}'$, M' such that:

$$\langle TS, \mathcal{S}_{\mathrm{future}}, M_{\mathrm{future}}\rangle \to^* \langle TS', \mathcal{S}', M'\rangle \ \wedge\ TS'.\mathtt{prm} = \emptyset$$

Machine and Behaviors A *machine state* is a triple $\mathbf{MS} = \langle \mathcal{TS}, \mathcal{S}, M\rangle$ consisting of a function $\mathcal{TS}$ assigning a thread state to every thread, an SC timemap $\mathcal{S}$, and a memory M. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state σ_i^0, the zero thread view (all timestamps in all timemaps are 0), and an empty set of promises; the zero timemap $\mathcal{S}^0$; and the initial memory M^0 consisting of one message $\langle x : 0@(0, 0], \bot\rangle$ for each location x. The machine step is defined by the last rule in Figure 3. The variable e in the final thread configuration step can either be a usual step (e is empty), or denote a system call ($e = \mathrm{SysCall}(v)$).

To define the set of behaviors of a program $\mathcal{P}$ (namely, what is externally observable during $\mathcal{P}$'s executions), we use the system calls that $\mathcal{P}$'s executions perform. More precisely, every execution induces a sequence of system calls (each includes a specific value for input/output), and the set of behaviors of $\mathcal{P}$ is taken to be the set of all system call sequences induced by executions of $\mathcal{P}$.

Promise-Free Machine In several of our results below, we make use of the fragment of our model obtained by revoking the ability to make promises (*i.e.,* omitting the PROMISE rule). We call this the *promise-free machine*.

5. Results

In this section, we outline a number of important results we have proven to hold of our "promising" model.

All the results of this section **are fully validated in Coq** except for §5.3 and Theorems 2 and 3, for which we provide proof outlines. All Coq development and proof outlines are available at [1].

5.1 Compiler Transformations

A transformation $\mathcal{P}_{\text{src}} \rightsquigarrow \mathcal{P}_{\text{tgt}}$ is *sound* if it does not introduce new behaviors under any (parallel and sequential) context, that is, for every context $\mathcal{C}$, every behavior of $\mathcal{C}[\mathcal{P}_{\text{tgt}}]$ is a behavior of $\mathcal{C}[\mathcal{P}_{\text{src}}]$.

Next, we list the program transformations proven to be sound in our model. To streamline the presentation, we refer to transformations on the *semantic* level, as if they are applied on *actions*, namely fences and (valueless) memory accesses. Thus, we presuppose adequate syntactic manipulations on the program level that implement these semantic transformations. For example, a syntactic transformation implementing $\mathtt{R}^x_{\mathtt{rlx}}; \mathtt{R}^y_{\mathtt{rlx}} \rightsquigarrow \mathtt{R}^y_{\mathtt{rlx}}; \mathtt{R}^x_{\mathtt{rlx}}$ is a reordering $a := x; b := y \rightsquigarrow b := y; a := x$ on the program code (assuming $a \neq b$); while a merge of a write and an update correspond, *e.g.,* to a transformation of the form $x := a; \text{FAA}(x, 1) \rightsquigarrow x := a + 1$. Nevertheless, our formal development proves soundness of transformations on the purely *syntactic* level, assuming a simple programming language with memory operations, conditionals, and loops.

Trace-Preserving Transformations Transformations that do not change the set of traces of actions in a given thread are clearly sound. For example, $y := a + 1 - a \rightsquigarrow y := 1$ is a sound transformation (recall that a denotes a local register; see §1:LBfd). Indeed, this is the crucial property that distinguishes a memory model for a higher-level language from a hardware memory model.

Strengthening A simple transformation that is sound in our model is strengthening of access modes. A read/write action $\mathtt{X}_o$ can be transformed to $\mathtt{X}_{o'}$ provided that $o \sqsubseteq o'$. Similarly, it is sound to replace $\mathtt{U}_{o_{\text{r}}, o_{\text{w}}}$ by $\mathtt{U}_{o'_{\text{r}}, o'_{\text{w}}}$ provided that $o_{\text{r}} \sqsubseteq o'_{\text{r}}$ and $o_{\text{w}} \sqsubseteq o'_{\text{w}}$, or to strengthen $\mathtt{F}_{\mathtt{rel}}$ or $\mathtt{F}_{\mathtt{acq}}$ to $\mathtt{F}_{\mathtt{sc}}$.

Reordering Next we consider transformations of the form $\mathtt{X}; \mathtt{Y} \rightsquigarrow \mathtt{Y}; \mathtt{X}$, and specify the set of *reorderable* pairs, that is the set of pairs $\mathtt{X}; \mathtt{Y}$ for which we proved this reordering transformation to be sound in our model. First, the following pairs are reorderable (where x and y denote distinct locations):

- $\mathtt{W}^x; \mathtt{R}^y$
- $\mathtt{W}^x; \mathtt{W}^y_{\sqsubseteq \mathtt{rlx}}$
- $\mathtt{W}; \mathtt{F}_{\mathtt{acq}}$
- $\mathtt{R}^x_{\sqsubseteq \mathtt{rlx}}; \mathtt{R}^y$ and $\mathtt{R}^x_{\mathtt{pln}}; \mathtt{R}^x_{\mathtt{pln}}$
- $\mathtt{R}^x_{\sqsubseteq \mathtt{rlx}}; \mathtt{W}^y_{\sqsubseteq \mathtt{rlx}}$
- $\mathtt{R}_{\neq \mathtt{rlx}}; \mathtt{F}_{\mathtt{acq}}$
- $\mathtt{F}_{\mathtt{rel}}; \mathtt{W}_{\neq \mathtt{rlx}}$
- $\mathtt{F}_{\mathtt{rel}}; \mathtt{R}$
- $\mathtt{F}_{\mathtt{rel}}; \mathtt{F}_{\mathtt{acq}}$

In addition, for the purpose of specifying reorderable pairs, an update is just a combination of a read and a write. Thus, $\mathtt{X}; \mathtt{U}_{o_{\text{r}}, o_{\text{w}}}$ is reorderable if both $\mathtt{X}; \mathtt{R}_{o_{\text{r}}}$ and $\mathtt{X}; \mathtt{W}_{o_{\text{w}}}$ are reorderable, and symmetrically $\mathtt{U}_{o_{\text{r}}, o_{\text{w}}}; \mathtt{X}$ is reorderable if both $\mathtt{R}_{o_{\text{r}}}; \mathtt{X}$ and $\mathtt{W}_{o_{\text{w}}}; \mathtt{X}$ are reorderable. In particular, a pair $\mathtt{U}^x_{o^x_{\text{r}}, o^x_{\text{w}}}; \mathtt{U}^y_{o^y_{\text{r}}, o^y_{\text{w}}}$ is reorderable if $x \neq y$, $o^x_{\text{r}} \sqsubseteq \mathtt{rlx}$, $o^y_{\text{w}} \sqsubseteq \mathtt{rlx}$.

The set of reorderable pairs in our model contains all pairs that are intended to be reorderable in the C++ and Java memory models, including in particular all "roach-motel reorderings" [28, 26].

Merging These are transformations that completely eliminate an action. Clearly, the two actions in *mergeable* pairs (pairs for which we proved the merge to be sound in our model) should access the same location. Now, the following four kinds of pairs are mergeable:

$$
\begin{array}{llll}
(\!|\mathtt{R}_{\sqsubseteq \mathtt{rlx}}|\!) & = \mathtt{ld} & (\!|\mathtt{R}_{\mathtt{ra}}|\!) & = \mathtt{ld;\ lwsync} \\
(\!|\mathtt{W}_{\sqsubseteq \mathtt{rlx}}|\!) & = \mathtt{st} & (\!|\mathtt{W}_{\mathtt{ra}}|\!) & = \mathtt{lwsync;\ st} \\
(\!|\mathtt{F}_{\sqsubset \mathtt{sc}}|\!) & = \mathtt{lwsync} & (\!|\mathtt{F}_{\mathtt{sc}}|\!) & = \mathtt{sync} \\
\multicolumn{4}{l}{(\!|\mathtt{U}_{\sqsubseteq \mathtt{rlx}, \sqsubseteq \mathtt{rlx}}|\!) = \mathtt{L\!:\ lwarx;\ cmp;\ bc\ Lout;\ stwcx.;\ bc\ L;\ Lout\!:}} \\
(\!|\mathtt{U}_{\mathtt{ra}, \sqsubseteq \mathtt{rlx}}|\!) & = (\!|\mathtt{U}_{\mathtt{rlx}, \mathtt{rlx}}|\!)\mathtt{;\ lwsync} & (\!|\mathtt{U}_{\sqsubseteq \mathtt{rlx}, \mathtt{ra}}|\!) & = \mathtt{lwsync;}\ (\!|\mathtt{U}_{\mathtt{rlx}, \mathtt{rlx}}|\!) \\
(\!|\mathtt{U}_{\mathtt{ra}, \mathtt{ra}}|\!) & \multicolumn{3}{l}{= \mathtt{lwsync;}\ (\!|\mathtt{U}_{\mathtt{rlx}, \mathtt{rlx}}|\!)\mathtt{;\ lwsync}}
\end{array}
$$

Figure 4. Compilation to Power.

R-after-R: $\mathtt{R}_o; \mathtt{R}_o$ W-after-W: $\mathtt{W}_o; \mathtt{W}_o$ R-after-W: $\mathtt{W}; \mathtt{R}$

Using the strengthening transformation, the access modes here can be read as upper bounds (*e.g.,* $\mathtt{R}_{\mathtt{ra}}; \mathtt{R}_{\mathtt{rlx}}$ can be first strengthened to $\mathtt{R}_{\mathtt{ra}}; \mathtt{R}_{\mathtt{ra}}$ and then merged). Note that the R-after-W merge allows even to eliminate a redundant acquire read after a plain/relaxed write (as in Example §4.1:LBa′).

In addition, the following pairs involving updates are mergeable:

R-after-U: $\mathtt{U}_{\mathtt{rlx}, o}; \mathtt{R}_{\mathtt{rlx}}$, and $\mathtt{U}_{\mathtt{ra}, o}; \mathtt{R}_{\mathtt{ra}}$ U-after-W: $\mathtt{W}_o; \mathtt{U}_{o_{\text{r}}, o}$

U-after-U: $\mathtt{U}_{o_1, o}; \mathtt{U}_{o_2, o}$, provided that $\mathtt{U}_{o_1, o}; \mathtt{R}_{o_2}$ is mergeable

Note that read-after-update does not allow the read to be an acquire read unless the update includes an acquire read (unlike read-after-write elimination). This is due to release sequences: eliminating an acquire read after a relaxed update may remove the synchronization due to a release sequence ending in this update.

Finally, two fences of the same type can obviously be merged.

The set of mergeable pairs in our model contains all pairs intended to be mergeable in the C++ and Java models [28, 26]. In particular, we support R-after-W merging, which is the effect of local satisfaction of reads in hardware like TSO, Power, and ARM.

Introducing and Eliminating Unused Reads Introduction of irrelevant read accesses is sound in our model, unlike in the Java memory model [26]. Eliminating plain read accesses whose read values are never used in the program is also sound in our model. In contrast, eliminating relaxed or acquire reads is not generally sound because it may remove synchronization.

Proof Technique Our proof of these results employs the well-known approach of simulation relations between the target and the source programs. Importantly, our definitions ensure thread-locality, thus allowing us to define a simulation relation on thread configurations, which (as we prove) can be composed into a simulation relation on full machine states. Additionally, for thread configurations, we prove the adequacy of simulation up-to context, which lets us to ignore the certification processes in the source and the target, and just provide simulations between simple "code snippets".

5.2 Compilation to TSO

Like C++11, our model can be efficiently compiled to x86-TSO. Since this architecture provides relatively strong guarantees, every memory access can be compiled to a primitive hardware instruction. Moreover, release/acquire fences are ignored during compilation, and SC fences are mapped to an MFENCE instruction. Correctness of this mapping follows from a recent result by Lahav and Vafeiadis [18], which shows that all weak behaviors of TSO are explained by store-load reordering and merging. Accordingly, it reduces the correctness proof of compilation to TSO to: (i) supporting write-read reordering and write-read merge; and (ii) a correctness proof of compilation to SC. Since we proved the soundness of write-read reordering and merge (regardless of the access modes of the two events), and since clearly our model is weaker than SC, we immediately derive the correctness of compilation to TSO.

5.3 Compilation to Power

Figure 4 provides the compilation scheme of our model to Power, following the C++11 one. We denote by $(\!|\mathcal{P}|\!)$ the Power program obtained by applying this mapping to a source program $\mathcal{P}$. To prove compilation correctness, we again utilize a result of [18], which shows that every allowed behavior of a Power program $\mathcal{Q}$ (assuming its recent declarative model of Alglave *et al.* [5]) is a behavior of a Power program $\mathcal{Q}'$ under a stronger model, called "StrongPower", where $\mathcal{Q}'$ is obtained from $\mathcal{Q}$ by applying a sequence of local reorderings of independent memory accesses to distinct locations. Accordingly, it suffices to show that, given a source program $\mathcal{P}$, our model allows all behaviors that StrongPower allows for some reordering of $(\!|\mathcal{P}|\!)$. We split this into two steps, outlined below.

Compilation to StrongPower First, we show that the compilation to StrongPower is correct, that is: every behavior of $(\!|\mathcal{P}|\!)$ under StrongPower is allowed for $\mathcal{P}$ in our model. StrongPower strengthens the Power model by forbidding "load buffering" behaviors (formally, it disallows cycles in the entire program order together with the reads-from relation). Consequently, we do not actually need promises in order to explain StrongPower behaviors—instead, we can just show that behaviors of $(\!|\mathcal{P}|\!)$ under StrongPower are allowed for $\mathcal{P}$ under our *promise-free* machine (see end of §4.6). (Note that this does not contradict the fact that promises are necessary to explain weak behaviors of the (non-strong) Power model, such as the LB.) To ease the proof, we use an alternative, declarative presentation of our promise-free machine (§5.6), which can straightforwardly be shown to be weaker than the StrongPower model under the compilation scheme in Figure 4. See [1] for details.

Reorderings in the Compiled Program Second, to account for sequences of reorderings of independent memory accesses to distinct locations in $(\!|\mathcal{P}|\!)$, we would like to relate them to the reorderings in the source program $\mathcal{P}$ that we proved sound in §5.1. But there is a subtle complication here: some reorderings in $(\!|\mathcal{P}|\!)$ do not correspond to reorderings in $\mathcal{P}$! For example, if $\mathcal{P}$ is $x :=_{\texttt{ra}} 1; y :=_{\texttt{rlx}} 2$, its compilation $(\!|\mathcal{P}|\!)$ has the form `lwsync;st` x 1`;st` y 2, but reordering the stores in $(\!|\mathcal{P}|\!)$ would produce `lwsync;st` y 2`;st` x 1, which does not correspond to the reordering of $\mathcal{P}$ to $y :=_{\texttt{rlx}} 2; x :=_{\texttt{ra}} 1$ (compiling the latter would yield `st` y 2`;lwsync;st` x 1).

To solve this issue, we slightly extend our model and consider compilation as if it happens in two stages. First, all release/acquire accesses in $\mathcal{P}$ are split into weaker accesses and corresponding fences as follows:

- $\texttt{R}_{\texttt{ra}} \rightsquigarrow \texttt{R}_{\texttt{rlx}}; \texttt{F}_{\texttt{acq}}$ and $\texttt{W}_{\texttt{ra}} \rightsquigarrow \texttt{F}_{\texttt{rel}}; \texttt{W}_{\texttt{srlx}}$
- $\texttt{U}_{\texttt{ra},o} \rightsquigarrow \texttt{U}_{\texttt{rlx},o}; \texttt{F}_{\texttt{acq}}$ and $\texttt{U}_{o,\texttt{ra}} \rightsquigarrow \texttt{F}_{\texttt{rel}}; \texttt{U}_{o,\texttt{srlx}}$ where $o \sqsubseteq \texttt{rlx}$

Here, we introduced a new `srlx` ("strong relaxed") mode, which has the same semantics as `rlx` but blocks promises like `ra` writes (*i.e.,* in write/update steps, we require that $\forall m' \in P(x).\ m'.\texttt{view} = \bot$ if $o \sqsupseteq \texttt{srlx}$). The reason for this is technical: we would have liked to just use `rlx` rather than `srlx`, but at least for $\texttt{U}_{o,\texttt{ra}}$, the mapping to $\texttt{F}_{\texttt{rel}}; \texttt{U}_{o,\texttt{rlx}}$ is unsound. Using `srlx`, however, we have proved the soundness of the above source-to-source mappings (in Coq) using a straightforward simulation argument (the target program's promises are subject to the same constraints as the source's), and `srlx` is sufficient for the rest of the proof.

After this first step, we obtain a program $\mathcal{P}_1$, which has no `ra` accesses except possibly for $\texttt{U}_{\texttt{ra},\texttt{ra}}$'s (which are surrounded by fences after compilation), and which has `srlx` accesses only immediately after release fences. (The relevance of this property will become clearer below.) For instance, in the example given above, $\mathcal{P}_1$ would be **fence-rel**; $x :=_{\texttt{srlx}} 1; y :=_{\texttt{rlx}} 2$. We then apply the compilation scheme of Figure 4, where `srlx` is compiled like `rlx`. Clearly, by construction of $\mathcal{P}_1$, the result $(\!|\mathcal{P}_1|\!)$ is identical to $(\!|\mathcal{P}|\!)$. Moreover, sequences of reorderings of accesses applied to $(\!|\mathcal{P}|\!)$ now correspond directly to sequences of reorderings in $\mathcal{P}_1$.

Thus, suppose Power program $\mathcal{Q}$ is the result of applying some sequence of reorderings of accesses to the compilation result $(\!|\mathcal{P}|\!) = (\!|\mathcal{P}_1|\!)$. Then, there exists a source program $\mathcal{P}_2$ such that $(\!|\mathcal{P}_2|\!) = \mathcal{Q}$, and $\mathcal{P}_2$ is obtained from $\mathcal{P}_1$ by applying a sequence of reorderings at the source level. Crucially, the reorderings that take $\mathcal{P}_1$ to $\mathcal{P}_2$ are all sound in our model: in addition to those already covered in §5.1, the reorderings of $\texttt{W}^x_{\texttt{srlx}}/\texttt{U}^x_{\texttt{rlx},\texttt{srlx}}$ past a $\texttt{R}^y_{\texttt{rlx}}/\texttt{W}^y_{\texttt{rlx}}/\texttt{U}^y_{\texttt{rlx},\texttt{rlx}}$ (where $x \neq y$) have also been proven sound in Coq. (Note that the reverse reorderings—moving accesses *past* a `srlx`—are not needed because of the aforementioned property that all `srlx`'s in $\mathcal{P}_1$ come immediately after a release fence.) Returning to the above example, when matching the reordering of `lwsync;st` x 1`;st` y 2 to `lwsync;st` y 2`;st` x 1 in the target, these new reorderings validate the transformation of **fence-rel**; $x :=_{\texttt{srlx}} 1; y :=_{\texttt{rlx}} 2$ to **fence-rel**; $y :=_{\texttt{rlx}} 2; x :=_{\texttt{srlx}} 1$ in the source.

Putting it all together: by the compilation to StrongPower result, we know that any behavior of $\mathcal{Q}$ under StrongPower is also a behavior of $\mathcal{P}_2$ in our model; by soundness of the reorderings, we know this is also a behavior of $\mathcal{P}_1$; and by soundness of the source-to-source mappings, it is also a behavior of the original $\mathcal{P}$.

Finally, we note that for C++11, there is a more efficient compilation scheme of acquire reads and updates, which uses a control dependency and an `isync` fence instead of a lightweight fence (`lwsync`). We believe that our model is also correctly compiled using this scheme. Nevertheless, this will require a more direct proof (`isync` fences are beyond the reach of [18]), which we leave to future work.

5.4 DRF Theorems

We proceed with an explanation of our DRF theorems. These theorems provide ways of restricting attention to better-behaved subsets of the model assuming certain conditions on programs.

Evidently, the most complicated part of our semantics is the promises. Without promises, our model amounts to a usual operational model, where thread steps only arise because of program instructions. Hence, our first DRF result (and the one that is by far the most challenging to prove) identifies a set of programs for which promises cannot introduce additional behaviors. Specifically, we show that this holds for programs in which all racy accesses are release/acquire, assuming a promise-free semantics. Crucially, as usual in DRF guarantees, the races are considered under the stronger semantics (promise-free), not the full model, thus allowing programmers to adhere to this programming discipline while being completely ignorant of the weak semantics (promises).

More precisely, we say that a machine state **MS** is *o-race-free*, if whenever two different threads may take a (non-promise) step accessing the same location, then both accesses are reads or both have access mode strictly stronger than o.

Theorem 1 (Promise-Free DRF)**.** Let $\Rightarrow$ denote the steps of the promise-free machine (see end of 4.6). Suppose that every machine state that is $\Rightarrow$-reachable from the initial state of a program $\mathcal{P}$ is `rlx`-race-free. Then, the behaviors of $\mathcal{P}$ according to the full machine coincide with those according to the $\Rightarrow$-machine.

Putting promises aside, a counter-intuitive part of weak memory models are the relaxed accesses, which allow threads to observe writes without observing previous writes to other locations. Removing `pln`/`rlx` accesses, namely keeping only `ra`, substantially simplifies our machine (in particular, its thread views would consist of just one view, the `cur` one). Accordingly, our second DRF result strengthens Theorem 1 and states that it suffices to show that there are only races on `ra` accesses *under release/acquire semantics* to conclude that a program has only release/acquire behaviors.

Theorem 2 (DRF-RA)**.** Let $\overset{\texttt{ra}}{\Longrightarrow}$ be identical to $\Rightarrow$ in Theorem 1, except for interpreting `rlx` and `pln` accesses in program transitions as if they are all `ra`-accesses. Suppose that every machine state that is $\overset{\texttt{ra}}{\Longrightarrow}$-reachable from the initial state of a program $\mathcal{P}$ is `rlx`-

race-free. Then, the behaviors of $\mathcal{P}$ according to the full machine coincide with those according to the $\overset{\text{ra}}{\Longrightarrow}$-machine.

We observe that promise re-certification is necessary for the proof of DRF-RA: in [1] we show a counterexample to DRF-RA in the absence of promise re-certification.

To state a more standard DRF theorem, we assume programs are *well-locked*: (1) locations are partitioned into *normal* and *lock* locations, and (2) lock locations are accessed only by matching pairs of the following lock/unlock operations:

$$\begin{aligned} \textit{lock}(l): &\quad \textbf{while } !\text{CAS}(l, 0, 1, \textbf{acqrel}) \textbf{ do skip}; \\ \textit{unlock}(l): &\quad l_{\textbf{rel}} := 0; \end{aligned}$$

The theorem forbids any weak behavior in programs that, under SC semantics, race only on lock locations. For the SC semantics, we consider "an interleaving machine", where reads read from the latest write to the appropriate location (regardless of the access modes).

Theorem 3 (DRF-LOCK)**.** Let $\overset{\text{sc}}{\Longrightarrow}$ denote the steps of the interleaving machine. Suppose that every machine state that is $\overset{\text{sc}}{\Longrightarrow}$-reachable from the initial state of a well-locked program $\mathcal{P}$ is race-free on normal locations. Then, the behaviors of $\mathcal{P}$ according to the full machine coincide with those according to the $\overset{\text{sc}}{\Longrightarrow}$-machine.

5.5 An Invariant-Based Program Logic

Besides the DRF guarantees, to demonstrate that our model does not suffer from the disastrous consequences of OOTA, we prove soundness of a very simple program logic for concurrent programs with respect to our model. In particular, it can be trivially used to show that LBd must return $a = 0$, and more generally, that programs cannot read values they never wrote. Note that even this basic logic is unsound for C++'s relaxed accesses (whereas it is sound in our model even if all accesses are plain).

We take a *program proof* to be a tuple $\langle J, S_1, S_2, ...\rangle$, where J is a global invariant over the shared variables and each $S_i \subseteq \mathsf{State}_i$ is a set of local states (intuitively describing the reachable states of thread i) such that the following conditions hold:

- $\sigma_i^0 \in S_i$ and $\bigwedge_{x \in \mathsf{Loc}} x = 0 \vdash J$.
- If $\sigma_i \xrightarrow{\mathtt{R}(o,x,v)} \sigma_i'$ then $\sigma_i \in S_i \wedge J \wedge x = v \vdash \sigma_i' \in S_i$.
- If $\sigma_i \xrightarrow{\mathtt{W}(o,x,v)} \sigma_i'$ then $\sigma_i \in S_i \wedge J \vdash \sigma_i' \in S_i \wedge J[v/x]$.
- If $\sigma_i \xrightarrow{\mathtt{U}(o_\mathrm{r}, o_\mathrm{w}, x, v_\mathrm{r}, v_\mathrm{w})} \sigma_i'$ then $\sigma_i \in S_i \wedge J \wedge x = v_\mathrm{r} \vdash \sigma_i' \in S_i \wedge J[v_\mathrm{w}/x]$.
- For $e \in \{\mathtt{F}_{\mathtt{acq}}, \mathtt{F}_{\mathtt{rel}}, \mathtt{F}_{\mathtt{sc}}, \text{Silent}, \text{SysCall}(v)\}$, if $\sigma_i \xrightarrow{e} \sigma_i'$ then $\sigma_i \in S_i \vdash \sigma_i' \in S_i$.

Figure 5 provides an illustration of a program proof showing that LBd does not exhibit weak behaviors.

Now, given a program proof for a program $\mathcal{P}$, we can show that all the reachable states $\mathbf{MS}$ from the initial state $\mathbf{MS}^0$ of $\mathcal{P}$ satisfy the global invariant J:

Theorem 4 (Soundness)**.** Let $\langle J, S_1, S_2, ...\rangle$ be a program proof, and let $\mathbf{MS} = \langle \mathit{TS}, \mathcal{S}, M\rangle$ such that $\mathbf{MS}^0 \to^* \mathbf{MS}$. Then, $\mathit{TS}(i).\mathtt{st} \in S_i$ for every thread i, and $\bigwedge_{x \in \mathsf{Loc}} x = f(x).\mathtt{val} \vdash J$ for every function f that assigns to every location x a message $m \in M$ such that $m.\mathtt{loc} = x$.

Our Coq proof of this theorem is simple: it holds trivially for promise-free executions, and extends easily to promise steps, since every promise step has a promise-free certification.

5.6 Declarative Presentation of the Promise-Free Machine

In this section, we provide a declarative presentation of our model, in the style of C++11, namely using sets of *execution graphs* to describe the possible behaviors of programs. This presentation abstracts away particular timestamp choices and thread views, and replaces them by formal conditions on partial orders in execution graphs. The promise mechanism has an inherent operational nature, and thus our declarative presentation only applies to the *promise-free machine* (see end of §4.6). Nevertheless, this presentation is useful for comparing our model to C++11, and it is used for establishing the correctness of compilation (see §5.3). We also find this as a technical device for analyzing possible behaviors of the promise-free machine and applying Theorem 1.

$$\begin{array}{l||l} \begin{array}{l} \{J\} \\ a := x; \\ \{J \wedge (a = 0)\} \\ y := a; \\ \{J\} \end{array} & \begin{array}{l} \{J\} \\ x := y; \\ \{J\} \end{array} \qquad J \overset{\text{def}}{=} (x = 0) \wedge (y = 0) \end{array}$$

Figure 5. A simple derivation in the invariant-based program logic.

The nodes of execution graphs are called *events*. An event consists of an identifier (natural number), a thread identifier (taken from a finite set Tid of thread identifiers, or 0 for initialization events), and a *label*. Labels have the form $\mathtt{R}(o, x, v)$ or $\mathtt{W}(o, x, v)$ (where o is the access mode, x is the location accessed, and v is the value read/written), as well as $\mathtt{F}_{\mathtt{acq}}$, $\mathtt{F}_{\mathtt{rel}}$, or $\mathtt{F}_{\mathtt{sc}}$ (for fences). The functions tid, lab, typ, and loc, return (when applicable) the thread identifier, label, type ($\mathtt{R}, \mathtt{W}, \mathtt{F}$), and location of an event.

In turn, an *execution graph* G consists of:

- a set $\mathtt{E}$ of events. This set always contains a set $\mathtt{E}_0$ of initialization events, consisting of one plain write event assigning the initial value for every location. We assume that all initial values are 0. For $\mathtt{T} \in \{\mathtt{R}, \mathtt{W}, \mathtt{F}\}$, we denote by $\mathtt{T}$ the set $\{a \in \mathtt{E} \mid \mathit{typ}(a) = \mathtt{T}\}$. We also use subscript for access modes (*e.g.,* $\mathtt{W}_{\sqsupseteq \mathtt{rlx}}$ denotes the set of all events $a \in \mathtt{W}$ with whose access mode is at least $\mathtt{rlx}$).
- a relation $\mathtt{sb}$, called *sequenced before*, which is a union of relations $(\mathtt{E}_0 \times (\mathtt{E} \setminus \mathtt{E}_0)) \cup \{\mathtt{sb}_i \mid i \in \mathsf{Tid}\}$, where every $\mathtt{sb}_i$ ($i \in \mathsf{Tid}$) is a strict total order on $\{a \in \mathtt{E} \mid \mathit{tid}(a) = i\}$.
- a relation $\mathtt{rmw}$, called *read-modify-write pairs*, consisting of *immediate* $\mathtt{sb}$-edges (*i.e.,* if $\langle a, b\rangle \in \mathtt{rmw}$ then no c satisfies both $\langle a, c\rangle \in \mathtt{sb}$ and $\langle c, b\rangle \in \mathtt{sb}$). In addition, for every $\langle a, b\rangle \in \mathtt{rmw}$, we have $\mathit{typ}(a) = \mathtt{R}$, $\mathit{typ}(b) = \mathtt{W}$, and $\mathit{loc}(a) = \mathit{loc}(b)$.
- a relation $\mathtt{rf}$, called *reads-from*, which relates every read event in $\mathtt{E}$ with one write event in $\mathtt{E}$ that has the same location and value.
- a relation $\mathtt{mo}$, called *modification order*, which is a disjoint union of relations $\{\mathtt{mo}_x\}_{x \in \mathsf{Loc}}$, such that each relation $\mathtt{mo}_x$ is a strict total order on $\mathtt{W}_x$.
- a relation $\mathtt{sc}$, called *SC order*, which is a strict total order on $\mathtt{F}_{\mathtt{sc}}$ (the set of all SC fence events in $\mathtt{E}$).

Notation 1. $R^?$, R^+, and R^* respectively denote the reflexive, transitive, and reflexive-transitive closures of a binary relation R. R^{-1} denotes its inverse relation. We denote by $R_1; R_2$ the left composition of two relations R_1, R_2. Finally, $[A]$ denotes the identity relation on a set A. In particular, $[A]; R; [B] = R \cap (A \times B)$.

Now, to define which execution graphs are allowed, we derive an *happens-before* order $\mathtt{hb}$, which is defined as in C++11 (after applying the correction suggested in [28] for release sequences):

$$\begin{aligned} \mathtt{sb}|_{\mathit{loc}} &= \{\langle a, b\rangle \in \mathtt{sb} \mid \mathit{loc}(a) = \mathit{loc}(b)\} && (\textit{sb-loc}) \\ \mathtt{rs} &= [\mathtt{W}]; \mathtt{sb}|_{\mathit{loc}}^?; [\mathtt{W}_{\sqsupseteq \mathtt{rlx}}]; (\mathtt{rf}; \mathtt{rmw}; [\mathtt{W}_{\sqsupseteq \mathtt{rlx}}])^* && (\textit{release-seq}) \\ \mathtt{rel} &= ([\mathtt{W}_{\mathtt{ra}}] \cup ([\mathtt{F}_{\mathtt{rel}} \cup \mathtt{F}_{\mathtt{sc}}]; \mathtt{sb})); \mathtt{rs} && (\textit{to-be-released}) \\ \mathtt{sw} &= \mathtt{rel}; \mathtt{rf}; ([\mathtt{R}_{\mathtt{ra}}] \cup ([\mathtt{R}_{\sqsupseteq \mathtt{rlx}}]; \mathtt{sb}; [\mathtt{F}_{\mathtt{acq}} \cup \mathtt{F}_{\mathtt{sc}}])) && (\textit{sync}) \\ \mathtt{hb} &= (\mathtt{sb} \cup \mathtt{sw})^+ && (\textit{happens-before}) \end{aligned}$$

Given the definition of $\mathtt{hb}$, an execution graph G is *consistent* if the following hold:

- $\mathtt{mo}; \mathtt{hb}$ is irreflexive. *(WW-coherence)*
- $\mathtt{mo}; \mathtt{rf}; \mathtt{hb}$ is irreflexive. *(RW-coherence)*
- $\mathtt{mo}; \mathtt{hb}; \mathtt{rf}^{-1}$ is irreflexive. *(WR-coherence)*
- $\mathtt{mo}; \mathtt{rf}; R; \mathtt{rf}^{-1}$ is irreflexive, *(RR-coherence)* where $R = ([\mathtt{R}_{\sqsupseteq \mathtt{rlx}}]; \mathtt{hb}) \cup (\mathtt{hb}; [\mathtt{R}_{\sqsupseteq \mathtt{rlx}}]) \cup (\mathtt{hb}; [\mathtt{F}_{\mathtt{sc}}]; \mathtt{hb})$.
- $(\mathtt{rf}; \mathtt{rmw}) \cap (\mathtt{mo}; \mathtt{mo}) = \emptyset$. *(Atomicity)*
- $\mathtt{mo}; \mathtt{rf}^?; \mathtt{hb}; \mathtt{sc}; \mathtt{hb}; (\mathtt{rf}^{-1})^?$ is irreflexive. *(SC)*
- $\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{sc}$ is acyclic. *(No-promises)*

Putting aside differences in presentation (*e.g.,* instead of a primitive RMW event, we have two events related by an $\mathtt{rmw}$-edge), there are three essential differences between this model and the C++11 one [6]:

- Our model disallows cycles in $\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{sc}$, and hence it does not permit load buffering behaviors (which are not possible in the promise-free machine).
- Our model lacks SC accesses, and its condition for SC fences is stronger than the one of C++11. In particular, unlike in C++11, placing an SC fence between every two commands *does* guarantee SC semantics.
- Our model does not have non-atomic accesses on which races imply undefined behavior. It does have plain accesses for which RR-coherence does not apply (unless an SC fence is $\mathtt{hb}$-between).

Now, to define behaviors of programs using consistent execution graphs and programs, we present a "declarative machine", which incrementally builds a consistent execution graph following some interleaving of the program's threads operations. Formally, the declarative machine state is a pair $\langle \Sigma, G\rangle$, where Σ assigns a local state σ to every thread (as described in §2), and G is some consistent execution graph. The possible steps of this machine are given in Figure 6, assuming the same abstract programming language discussed in §4.6. To define these steps we use the following notation for execution graph extension.

Notation 2. For two execution graphs G and G', we write $G' \in \mathsf{Add}(G, a)$ if G' extends G with one event a, which is $\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{sc}$ maximal in G'. We write $G' \in \mathsf{AddRMW}(G, a_{\mathrm{r}}, a_{\mathrm{w}})$ if G' extends some $G_{\mathrm{mid}} \in \mathsf{Add}(G, a_{\mathrm{r}})$ with one event a_{w} and an $\mathtt{rmw}$-edge $\langle a_{\mathrm{r}}, a_{\mathrm{w}}\rangle$, and a_{w} is $\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{sc}$ maximal in G'.

The initial machine state is given by $\langle \lambda i.\sigma_i^0, G_0\rangle$, where G_0 contains only the initialization events $\mathtt{E}_0$ (its relations are empty). A behavior of a program under the declarative machine is again taken to be the set of sequences of system calls induced by its executions.

Remark 2. In a purely declarative style, one considers only *full* runs of a program and checks consistency only once at the end. However, the definition of consistent execution graphs above is "prefix-closed" (that is, every $\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{sc}$-prefix of a consistent execution graph forms a consistent execution graph). As a result, we were able to present our declarative semantics in a more operational style, which can be conveniently related to the promise-free machine.

Theorem 5. For every program $\mathcal{P}$, the behaviors of $\mathcal{P}$ according to the promise-free machine coincide with the behaviors of $\mathcal{P}$ according to the declarative machine.

The Coq proof of this theorem involves a non-trivial simulation argument. The simulation relation is presented in [1].

6. Related Work

There have been many proposals for solving the "out of thin air" problem. Several of them have come with proofs of DRF guarantees,

$$\frac{\sigma \xrightarrow{\text{Silent}} \sigma'}{\langle \sigma, G\rangle \xrightarrow{i} \langle \sigma', G\rangle} \qquad \frac{\begin{array}{c}\sigma \xrightarrow{lab(a)} \sigma' \\ typ(a) \in \{\mathtt{R}, \mathtt{W}, \mathtt{F}\} \\ tid(a) = i \\ G' \in \mathsf{Add}(G, a)\end{array}}{\langle \sigma, G\rangle \xrightarrow{i} \langle \sigma', G'\rangle} \qquad \frac{\begin{array}{c}\sigma \xrightarrow{\mathtt{U}(x, v_{\mathrm{r}}, v_{\mathrm{w}}, o_{\mathrm{r}}, o_{\mathrm{w}})} \sigma' \\ lab(a_{\mathrm{r}}) = \mathtt{R}(o_{\mathrm{r}}, x, v_{\mathrm{r}}) \\ lab(a_{\mathrm{w}}) = \mathtt{W}(o_{\mathrm{w}}, x, v_{\mathrm{w}}) \\ tid(a_{\mathrm{r}}) = tid(a_{\mathrm{w}}) = i \\ G' \in \mathsf{AddRMW}(G, a_{\mathrm{r}}, a_{\mathrm{w}})\end{array}}{\langle \sigma, G\rangle \xrightarrow{i} \langle \sigma', G'\rangle}$$

$$\frac{\begin{array}{cc}\sigma \xrightarrow{\text{SysCall}(v)} \sigma' & lab(a) = \mathtt{F}_{\mathtt{sc}} \\ tid(a) = i & G' \in \mathsf{Add}(G, a)\end{array}}{\langle \sigma, G\rangle \xrightarrow{i, \text{SysCall}(v)} \langle \sigma', G'\rangle} \qquad \text{(MACHINE STEP)}\ \frac{\begin{array}{c}\langle \Sigma(i), G\rangle \xrightarrow{i,e} \langle \sigma', G'\rangle \\ G' \text{ is consistent}\end{array}}{\langle \Sigma, G\rangle \xrightarrow{e} \langle \Sigma[i \mapsto \sigma'], G'\rangle}$$

Figure 6. Operational semantics based on the declarative model.

but ours is the first to come with formal (and machine-checked) validation of a wide range of essential local transformations (§5.1) concerning a full spectrum of features from the C++ model.

The first major attempt to solve the "out of thin air" problem was by the Java memory model (JMM) [21] (see also [20]). The JMM intended to validate all the compiler optimizations that Java compilers and just-in-time compilers might perform, but its formal definition failed to validate them [26]. Subsequent fixes were proposed to the model, which improved the set of enabled optimizations, but still falling short of what actual Java compilers were performing.

Interestingly, an early glimpse of our idea of promises may be seen in version 1.0 of the JMM [12], which describes a form of "prescient store actions" (§17.8). However, their description is very brief and vague, and the feature was removed for JSR 133 [3].

To resolve some of the problems with the JMM definition, Jagadeesan *et al.* [14] proposed an operational model following quite closely the intended behavior of the JMM, but employing the notion of a *speculation*. Speculations are similar to our notion of promises, but unlike promises they are not certified thread-locally: whereas we model interference conservatively by quantifying over all future memories during certification, they model interference from other threads more precisely by executing multiple threads together during certification. We believe our conservative approach is sufficient for justifying standard compiler optimizations, which are typically thread-local, and moreover it simplifies the presentation of the semantics and the development of the meta-theory because it avoids the need of nested certifications.

Jagadeesan *et al.*'s model satisfies the standard DRF theorem, as well as a DRF theorem saying that speculations are unnecessary for programs without read-write races. They also develop a simulation proof technique, with which they verify three optimizations: write-write reordering, roach-motel reordering, and read-after-read elimination. We have applied our simulation method to a much wider variety of optimizations, and our proofs are machine-checked in Coq. They also do not provide any compilation correctness results, and their model omits release-acquire accesses, updates, and fences.

More recently, Jeffrey and Riely [15] presented a weak memory model based on event structures. Their model admits a standard DRF theorem, but does not fully allow the reordering of independent memory accesses, and thus cannot be compiled to Power/ARM without extra fences. The paper suggests an idea about how to fix the model to support such reorderings, but it is not known whether the suggested fixed model avoids OOTA behaviors. Relating to our work, their model seems to be "promising" reads (instead of writes) and restricting the quantification over possible futures to only those that could arise from further execution of the current program. The model only supports relaxed accesses and locks.

Pichon-Pharabod and Sewell [24] introduced an event structure model with both a normal reduction rule, which executes an initial event of the event structure, and special reduction rules that mimic the effect of standard compiler optimizations on the event structure. These optimization rules include a rather complex rule for non-thread-local optimizations that can declare a whole branch of the event structure unreachable. The paper does not present any formal results about the model. It is worth noting that the model does not support the weak behavior of the ARM-weak program and thus may not be compiled to ARM without additional fences. The model only handles relaxed and non-atomic accesses and locks.

Podkopaev *et al.* [25] proposed an operational model covering a large subset of the features of the C++ model. They provide many litmus tests to demonstrate the suitability of their model, but do not prove any formal results about it. Their model ensures per-location coherence in a very similar way to our model: using timestamps. In order to handle read-write reorderings, they allow reads to return symbolic values, which are then evaluated at a later point in time when their value is actually needed. This approach gives the expected behaviors to the LB and LBd programs, and may be extended with a set of *syntactic* symbolic simplification rules to also give the expected result to the LBfd program. It seems, however, very difficult to extend this approach to enable code motion optimizations, where some common code is pulled out of two branches of a conditional. What makes code motion more challenging is that the common code may become apparent only after some earlier transformations, like for example the $y := 1$ assignment in the following code:

$$\left.\begin{array}{l} a := x; \ /\!\!/ 1 \\ \textbf{if } a = 1 \textbf{ then } y := a; \ \textbf{else } (z := 1; y := z;) \end{array}\right\| \ x := y;$$

Our model allows the annotated behavior of the program above, precisely because our promises are semantic in nature and thus avoid the brittle tracking of syntactic data dependencies.

Zhang and Feng [29] suggested an operational model for Java accesses in which threads may re-execute some memory events. The model admits a standard DRF theorem, and its replay mechanism enables it to support local transformations. However, to avoid OOTA, this mechanism is limited by its tracking of syntactic dependencies between instructions, and thus it fails to validate behaviors resulting from trace-preserving transformations like the one above.

Other proposals for language-level memory models have tried not to solve the OOTA problem, but to avoid it, by introducing stronger models where read-write reordering is not allowed. For example, Ševčík *et al.* [27] and Demange *et al.* [10] proposed using TSO as the memory model for C and Java, respectively. These proposals may be reasonable compromises if the only target machines of interest also follow the TSO model, but are prohibitively expensive on weaker architectures, such as Power and ARM, because enforcing TSO on those machines requires essentially as many fences as enforcing SC. In a similar line of work, Lahav *et al.* [17] introduced a strengthening of the release/acquire fragment of the C++ memory model, which they called SRA, together with an operational semantics for SRA. Compiling SRA to Power and ARM is cheaper than TSO, but still requires some fences before or after every shared variable access, and may thus not be suitable for performance-critical code.

Another approach is to simply allow OOTA behaviors. This was the approach taken by Batty *et al.*'s formalization of C++ [6], and by the OpenCL model [16], as well as by Crary and Sullivan [9], who introduced a more fine-grained specification of the orders that the model is supposed to preserve. All of these models allow the weak behavior of the LBd example, thereby invalidating standard reasoning principles and DRF theorems.

Finally, Norris and Demsky [22] presented a tool that exhaustively enumerates the behaviors of concurrent C++ programs. To account for speculative reads, the tool may establish "promised future values", which a load can read from, and which must eventually be written by a future store. Norris and Demsky's promises look superficially quite similar to ours, but their purpose is to support practical model checking of C++ programs, not to change the semantics of the language, so the paper does not present any formal model or metatheory of promises.

7. Future Work

Besides extending our model with SC accesses (see §1.3), there are a number of interesting issues remaining for future work.

Compilation Correctness Establishing the correctness of compilation of our model to ARM, as well as to Power using the more efficient compilation scheme for acquire reads and updates (see §5.3), is an important future goal. To the best of our knowledge, we know of no counterexamples for correctness of compilation to these architectures.

Global Optimizations In our model, we insist that promises can always be certified thread-locally. This decision enables thread-local reasoning about our semantics and suffices to justify all the known thread-local program transformations that a compiler or the hardware may perform. It does, however, render unsound some transformations of a global nature, such as sequentialization (aka "thread inlining"), which merges threads together. To see this, consider the following:

$$\left.\begin{array}{l} a := x; \ /\!\!/ 1 \\ \textbf{if } a = 0 \textbf{ then} \\ \quad x := 1; \end{array}\right\| \ y := x; \ \left\| \ x := y; \right. \quad \rightsquigarrow \quad \left.\begin{array}{l} a := x; \ /\!\!/ 1 \\ \textbf{if } a = 0 \textbf{ then} \\ \quad x := 1; \\ y := x; \end{array}\right\| \ x := y;$$

This source program disallows the specified behavior because if T_1 reads 1 for x after promising $x := 1$, then it will not be able to fulfill its promise. Nevertheless, the result $a = 1$ *is* allowed in the target program (obtained by sequentializing T_1 before T_2). Here, T_1 can safely promise $y := 1$, and later read $x = 1$ from T_2's write.[4] While sequentialization seems like a transformation that no compiler would perform, there might be other more useful global optimizations. Investigating what global optimizations are supported in our model is left for future work.

Liveness It is natural to extend our operational model with liveness guarantees, and it is useful and interesting to study their interaction with program transformations and DRF theorems. Liveness properties are currently mostly ignored in weak memory research.

Program Logic The program logic presented in §5.5 only establishes the very basic sanity of our memory model. Developing a useful program logic for this model is a direction for future work.

Simulation and Model Checking The high degree of non determinism in our model makes it hard to exhaustively explore all possible behaviors of a given program. Further work is required to develop efficient methods and tools for this purpose.

Acknowledgments

We thank Doug Lea, Alan Jeffrey, Andreas Lochbihler, James Riely, Peter Sewell, and Joe Tassarotti for very helpful feedback. This research was supported in part by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-07, and in part by an ERC Consolidator Grant for the project "RustBelt" (grant agreement no. 683289). The first author has been supported by a Korea Foundation for Advanced Studies Scholarship.

[4] Though sequentialization is a very intuitive property that one might expect a memory model to validate, we observe that TSO [23], Power [5], ARMv8 [11], Java [21], and C/C++11 [6] (without the corrections proposed in [28]) all do not allow sequentialization.

References

[1] Coq development and supplementary material for this paper available at: http://sf.snu.ac.kr/promise-concurrency.

[2] LLVM documentation. LLVM atomic instructions and concurrency guide. http://llvm.org/docs/Atomics.html.

[3] JSR 133. Java memory model and thread specification revision, 2004. http://jcp.org/jsr/detail/133.jsp.

[4] Sarita V. Adve and Mark D. Hill. Weak ordering—A new definition. In *Proc. 17th Annual International Symposium on Computer Architecture*, ISCA 1990, pages 2–14. ACM, 1990.

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proc. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2011, pages 55–66. ACM, 2011.

[7] Hans-Juergen Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proc. Workshop on Memory Systems Performance and Correctness*, MSPC 2014, pages 7:1–7:6. ACM, 2014.

[8] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *Proc. 15th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2017, 2017.

[9] Karl Crary and Michael J. Sullivan. A calculus for relaxed memory. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, pages 623–636. ACM, 2015.

[10] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. Plan B: A buffered memory model for Java. In *Proc. 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2013, pages 329–342. ACM, 2013.

[11] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 608–621. ACM, 2016.

[12] James Gosling, Bill Joy, and Guy Steele. The Java language specification, Edition 1.0, August 1996. http://titanium.cs.berkeley.edu/doc/java-langspec-1.0/.

[13] ISO/IEC 14882:2011. Programming language C++, 2011.

[14] Radha Jagadeesan, Corin Pitcher, and James Riely. Generative operational semantics for relaxed memory models. In *ESOP*, pages 307–326, 2010.

[15] Alan Jeffrey and James Riely. On thin air reads: Towards an event structures model of relaxed memory. In *Proc. IEEE Logic in Computer Science*, LICS 2016, 2016.

[16] Khronos Group. The OpenCL specification, Version 2.1, 2015.

[17] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 649–662. ACM, 2016.

[18] Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In *Proc. 21st International Symposium on Formal Methods*, FM 2016, 2016.

[19] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. Technical Report MPI-SWS-2016-011, MPI-SWS, November 2016.

[20] Andreas Lochbihler. Making the Java memory model safe. *ACM Trans. Program. Lang. Syst.*, 35(4):12:1–12:65, 2014.

[21] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2005, pages 378–391. ACM, 2005.

[22] Brian Norris and Brian Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proc. 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013, pages 131–150. ACM, 2013.

[23] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs 2009, pages 391–407. Springer, 2009.

[24] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 622–633. ACM, 2016.

[25] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016.

[26] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In *Proc. 22nd European Conference on Object-Oriented Programming, ECOOP 2008*, volume 5142 of *LNCS*, pages 27–51. Springer, 2008.

[27] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[28] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, pages 209–220. ACM, 2015.

[29] Yang Zhang and Xinyu Feng. An operational happens-before memory model. *Frontiers of Computer Science*, 10(1):54–81, 2016.