Taming Release-Acquire Consistency

Ori Lahav Nick Giannarakis Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS), Germany
{orilahav,nickgian,viktor}@mpi-sws.org

Abstract

We introduce a strengthening of the release-acquire fragment of the C11 memory model that (i) forbids dubious behaviors that are not observed in any implementation; (ii) supports fence instructions that restore sequential consistency; and (iii) admits an equivalent intuitive operational semantics based on point-to-point communication. This strengthening has no additional implementation cost: it allows the same local optimizations as C11 release and acquire accesses, and has exactly the same compilation schemes to the x86-TSO and Power architectures. In fact, the compilation to Power is complete with respect to a recent axiomatic model of Power; that is, the compiled program exhibits exactly the same behaviors as the source one. Moreover, we provide criteria for placing enough fence instructions to ensure sequential consistency, and apply them to an efficient RCU implementation.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Weak memory model; release-acquire; C11; operational semantics

1. Introduction

Weak memory models for programming languages formalize the set of behaviors that multithreaded programs may exhibit taking into account the effect of both the hardware architectures and compiler optimizations. An important such model is the C11 model introduced in the 2011 revisions of the C and C++ standards [14, 15].

C11 provides several kinds of memory accesses, each having a different implementation cost and providing different synchronization guarantees. At one end of the spectrum, non-atomic accesses provide absolutely no guarantees in case of racy access (they are considered programming errors). At the other end, sequentially consistent accesses are globally synchronized following the simple and well-known model of *sequential consistency* (SC) [19]. Between these two extremes, the most useful access kinds are *release* stores and *acquire* loads that strike a good balance between performance and programmability.

While the full C11 model suffers from some problems (such as "out-of-thin-air executions" [8, 38]), its *release-acquire* fragment, obtained by restricting all writes and reads to be release and acquire accesses respectively, constitutes a particularly useful and (relatively) well-behaved model. This fragment, hereinafter referred to as RA, provides much weaker guarantees than SC, that allow high performance implementations, and still suffice for fundamental concurrent algorithms (e.g., a variant of the read-copy-update synchronization mechanism [13] presented in the sequel). To understand RA, consider the following two programs:

Store buffering (SB)

Initially, $x = y = 0$.

$$\begin{array}{l||l} x := 1 & y := 1 \\ \texttt{wait}\,(y = 0) & \texttt{wait}\,(x = 0) \end{array}$$

May terminate under RA.

Message passing (MP)

Initially, $x = y = 0$.

$$\begin{array}{r||l} x := 1 & \texttt{wait}\ (y = 1) \\ y := 1 & \texttt{wait}\ (x = 0) \end{array}$$

Never terminates under RA.

RA is designed to support "message passing", a common idiom for relatively cheap synchronization between threads. On the other hand, to allow efficient implementations, a non-SC behavior is allowed in the case of the "store buffering" program. An intuitive (but incomplete) explanation of these examples can be given in terms of *reorderings* (performed by the hardware and/or the compiler): consecutive *write and read* accesses may be arbitrarily reordered, but reordering of *two reads* or *two writes* is disallowed. This easily accounts for the non-SC result of the SB program, and explains why MP does not expose undesirable behaviors.

The precise formulation of RA is *declarative* (also known as *axiomatic*): the model associates a set of graphs (called: *executions*) to every program, and filters out disallowed executions by imposing certain formal restrictions. Roughly speaking, it requires that every read is justified by a corresponding write, that cannot appear later according to the *program order*. This induces a *happens-before* relation on the memory accesses, that, assuming only release/acquire accesses, is taken to be the union of the program order and the *reads-from* justification edges. An additional condition is needed to ensure that reads cannot observe overwritten values. For that, the model asserts the existence of a per-location *modification order* that orders write accesses, and requires that reads are justified only by writes that happened before and are maximal according to the modification order.

The RA model is appropriate as a rigorous foundation that is not tied to a particular compiler and architecture. Further, C11 in general, and RA in particular, has verified compilation schemes to the x86-TSO and Power/ARM architectures [6, 7, 29]. However, RA suffers from three problems.

Problem 1: Unobservable relaxed behaviors. First, there is a mismatch between the declarative model and the intuitive reordering account. The following program (again, assuming all accesses are release/acquire) may terminate under the formal model if the

$x := 2$ and $y := 2$ writes are placed before the $x := 1$ and $y := 1$ writes in the respective modification orders.

$$\begin{array}{r||l} x := 1 & y := 1 \\ y := 2 & x := 2 \\ \multicolumn{2}{c}{\texttt{wait } (x = 1 \land y = 1)} \end{array} \tag{2+2W}$$

Disallowing reordering of consecutive writes should, however, forbid this behavior. Furthermore, this behavior is not observable on any known hardware under any known sound compilation scheme of the C11 release writes.

Problem 2: Overly weak SC fences. C11 provides a memory fence construct, called *SC fences*, that enforces stronger ordering guarantees among memory accesses. Despite their name, however, C11's semantics for these fences is overly weak and fails to ensure sequential consistency when used in conjunction with release-acquire accesses. For example, consider the following program:

$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; \begin{array}{l}\texttt{wait } (x = 1) \\ \texttt{wait } (y = 0)\end{array} \;\Big\|\; \begin{array}{l}\texttt{wait } (y = 1) \\ \texttt{wait } (x = 0)\end{array} \tag{IRIW}$$

Assuming that initially $x = y = 0$, this program may terminate under RA. Furthermore, according to the C11 semantics, inserting SC fence instructions between the two waits on the reader threads does not rule out this weak behavior. Again, this is a result of too liberal specification: the compilation of SC fences generates hardware fence instructions (`mfence` on x86-TSO, `sync` on Power, `dmb` on ARM, `mf` on Itanium) that forbid such weak behaviors.[1]

Problem 3: No intuitive operational semantics. The axiomatic nature of RA is also a drawback: global restrictions on executions do not allow the traditional understanding of programs that simulates their step-by-step progress. Thus, both programming and developing techniques for reasoning about programs are substantially more difficult with a purely declarative memory model. For these tasks, a more *operational* approach can be very helpful. The most well-known example is, of course, SC, that is naturally modeled by a machine with one global shared memory and interleaved accesses.

For weak models, *total store ordering* (TSO) [25, 33] provides a good case in point. This weak memory model emerged from a concrete operational model provided by the SPARC and x86 architectures. In addition to a main memory, TSO-machines have per-processor store buffers, where write operations are enqueued, and non-deterministically propagate to the main memory. Based on this operational model, several reasoning methods and analysis techniques have been developed for TSO. This includes some of the most useful and well-studied verification techniques: model checking and abstract interpretation (see, e.g., [1, 5, 12, 17, 20]), program logics (see, e.g., [27, 32]), simulations for verifying program transformations (see, e.g., [16, 22, 30, 37]), and reductions to SC (see, e.g., [2, 9, 10, 24]).

In this paper, we attempt to overcome these drawbacks. We propose a simple strengthening of RA that solves problems 1 and 3, and an alternative modeling of SC fences to solve problem 2. Next, we briefly overview each of these contributions.

Strong release-acquire. We introduce a model, called SRA (for "Strong RA"), that strengthens RA by requiring the union of the per-location modification orders and the happens-before relation to be acyclic. As a result, SRA forbids behaviors that require reordering of two writes as in the (2+2W) example, but coincides with RA for programs containing no write-write races.

[1] Specifically for the IRIW program, we note that on x86-TSO and Itanium, its weak behavior is forbidden even without a fence. For Itanium, see `ftp://download.intel.com/design/Itanium/Downloads/25142901.pdf`, §3.3.5.

More importantly, alongside the declarative definition, SRA has an intuitive operational presentation, that, like the operational semantics of TSO, does not refer at all to formal restrictions on graphs and partial orders. Unlike TSO, SRA-machines are based on point-to-point communication: processors have local memories and they communicate with one another via ordered message buffers using global timestamps to order different writes to the same location. This operational model is not meant to mimic any real hardware implementation, but to serve as a formal foundation for understanding and reasoning about concurrent programs under SRA.

Moreover, we show that SRA allows the same program transformations as RA and has the same implementation cost over x86-TSO and Power. For the latter, we prove that the existing compilation schemes for the C11 release-acquire accesses to TSO and Power guarantee the stronger specification of SRA. In fact, under the declarative Power/ARM memory model of Alglave et al. [4] the compilation to Power corresponds exactly to SRA, which means that SRA cannot further be strengthened without performance implications. In contrast, TSO is strictly stronger than SRA (the difference can be observed even with two threads). Nevertheless, for a restricted class of programs following a certain client-server communication discipline, we prove that TSO and SRA coincide.

Stronger semantics for SC fences. To address problem 2 above, we suggest a new semantics for SC fences. Our key idea is to model SC fence commands using an existing mechanism: as if they were atomic (acquire-release) update commands to a special, otherwise unused, location with an arbitrary value. We show that inserting fences between every two potentially racy RA accesses in each thread restores SC. For a particular large and common class of programs, it suffices to have a fence between every potentially racy *write* and subsequent potentially racy *read*. To demonstrate the usefulness of this criterion, we show how it can be utilized to reduce reasoning about the correctness of an RA-based RCU implementation down to SC.

Moreover, we show that this modeling of SC fences bears no additional implementation cost, and again one can follow the existing compilation scheme used for SC fences to TSO and Power. In fact, we prove that the semantics of Power's `sync` instruction is *equivalent* to our modeling of SC fences as acquire-release updates.

The rest of this paper is organized as follows: §2 reviews the RA memory model, §3 presents our declarative SRA model, §4 provides an equivalent operational model for SRA, §5 discusses fences and reduction theorems to SC, §6 shows that TSO is stronger than SRA and presents conditions under which they are equivalent, §7 proves that Power's `sync` fences are equivalent to release-acquire updates to a distinguished location and that SRA and its compilation to Power are equivalent, §8 discusses related work, and §9 concludes.

Supplementary material including full proofs as well as Coq proof scripts is available at: `http://plv.mpi-sws.org/sra/`. Except for the results in §4, all propositions and theorems of this paper have been proved in Coq with minor presentational differences.

2. RA Memory Model

In this section, we present RA, the declarative model behind the release-acquire fragment of C11's memory model [6]. The basic approach in the C11 formalization is to define the semantics of a program P to be the set of *consistent executions* of P. For the simplicity of the presentation, we employ a simplified programming language. While our notations are slightly different, the declarative semantics presented in this section corresponds exactly to the semantics of C11 programs from [6] in which all reads are acquire

$$\frac{}{\texttt{skip}; c \to c} \qquad \frac{c_1 \xrightarrow{l} c_1'}{c_1; c_2 \xrightarrow{l} c_1'; c_2} \qquad \frac{c_1 \to c_1'}{c_1; c_2 \to c_1'; c_2} \qquad \frac{y \in \mathit{fv}[e] \qquad l = \langle \texttt{R}, y, v\rangle}{x := e \xrightarrow{l} x := e\{v/y\}} \qquad \frac{\mathit{fv}[e] = \emptyset \qquad l = \langle \texttt{W}, x, \llbracket e \rrbracket\rangle}{x := e \xrightarrow{l} \texttt{skip}} \qquad \frac{l = \langle \texttt{U}, x, v, \llbracket e \rrbracket(v)\rangle}{\langle x := e(x)\rangle \xrightarrow{l} \texttt{skip}}$$

$$\frac{y \in \mathit{fv}[e] \qquad l = \langle \texttt{R}, y, v\rangle}{\texttt{if } e \texttt{ then } c \texttt{ else } c' \xrightarrow{l} \texttt{if } e\{v/y\} \texttt{ then } c \texttt{ else } c'} \qquad \frac{\mathit{fv}[e] = \emptyset \qquad \llbracket e \rrbracket \neq 0}{\texttt{if } e \texttt{ then } c \texttt{ else } c' \to c} \qquad \frac{\mathit{fv}[e] = \emptyset \qquad \llbracket e \rrbracket = 0}{\texttt{if } e \texttt{ then } c \texttt{ else } c' \to c'}$$

$$\frac{}{\texttt{repeat } c \texttt{ until } e \to c; \texttt{if } e \texttt{ then } (\texttt{repeat } c \texttt{ until } e) \texttt{ else skip}}$$

$$\frac{l = \langle \texttt{R}, x, v\rangle \qquad \llbracket e \rrbracket(v) = 0}{\texttt{when } e(x) \texttt{ do } x := e'(x) \xrightarrow{l} \texttt{when } e(x) \texttt{ do } x := e'(x)} \qquad \frac{l = \langle \texttt{U}, x, v, \llbracket e' \rrbracket(v)\rangle \qquad \llbracket e \rrbracket(v) \neq 0}{\texttt{when } e(x) \texttt{ do } x := e'(x) \xrightarrow{l} \texttt{skip}}$$

Figure 1. Command steps.

$$\frac{P(i) \xrightarrow{l} c}{P \xrightarrow{l,i} P[i \mapsto c]} \qquad \frac{P(i) \to c}{P \to P[i \mapsto c]}$$

Figure 2. Program steps

$$\frac{\mathit{lab}(a) = l \qquad \mathit{tid}(a) = i}{G \xrightarrow{l,i} G; a}$$

Figure 3. Execution steps

$$\frac{P \xrightarrow{l,i} P' \qquad G \xrightarrow{l,i} G'}{\langle P, G\rangle \to \langle P', G'\rangle} \qquad \frac{P \to P'}{\langle P, G\rangle \to \langle P', G\rangle}$$

Figure 4. Program and execution combined steps

reads, writes are release writes, and atomic updates are acquire-release read-modify-writes (RMWs).

Basic notations. Given a relation R on a set A, $R^?$, R^+, and R^* respectively denote its reflexive, transitive, and reflexive-transitive closures. The inverse relation of R is denoted by R^{-1}. When R is a strict partial order, a pair $\langle a, b\rangle \in R$ is called an *immediate R-edge* if b immediately follows a in R, that is: no $c \in A$ satisfies both $\langle a, c\rangle \in R$ and $\langle c, b\rangle \in R$. We denote by $R_1; R_2$ the left composition of two relations R_1, R_2. Finally, Id_A denotes the identity relation on the set A.

A simplified programming language. We assume a finite set Loc of locations, a finite set Val of values with a distinguished value $0 \in \mathsf{Val}$, and any standard interpreted language for expressions containing at least all locations and values. We use x, y, z as metavariables for locations, v for values, e for expressions, and denote by $\mathit{fv}[e]$ the set of locations that appear free in e. The sequential fragment of the language is given by the following grammar:

$$\begin{aligned} c ::= {} & \texttt{skip} \mid \texttt{if } e \texttt{ then } c \texttt{ else } c \mid \texttt{repeat } c \texttt{ until } e \mid \texttt{wait } e \mid \\ & c \,;c \mid x := e \mid \langle x := e(x)\rangle \mid \texttt{when } e(x) \texttt{ do } x := e'(x) \end{aligned}$$

All commands are standard. The command `wait` e is a syntactic sugar for `repeat skip until` e. The command $\langle x := e(x)\rangle$ is an atomic assignment corresponding to a primitive RMW instruction and, as such, mentions only one location. The command `when` $e(x)$ `do` $x := e'(x)$ corresponds to a compare-and-swap loop, that loops until the value v of x satisfies $\llbracket e \rrbracket(v) \neq 0$, and then atomically assigns $\llbracket e' \rrbracket(v)$ to x.

To define multithreaded programs, we assume a constant number N of threads with thread identifiers being $1, \dots, N$, and take a program P to be a function that assigns a command c to every thread identifier. We use i, j as metavariables for thread identifiers.

Executions. We employ the following terminology:

- A *type* is either `R` ("Read"), `W` ("Write"), or `U` ("Update"). We use `T` as a metavariable for types.
- A *label* is either a triple of the form $\langle \texttt{R}, x, v_r\rangle$, a triple of the form $\langle \texttt{W}, x, v_w\rangle$, or a quadruple of the form $\langle \texttt{U}, x, v_r, v_w\rangle$. We denote by Lab the set of all labels.
- An *event* is a tuple of the form $\langle k, i, l\rangle$, where k is an event identifier (natural number), i is a thread identifier (natural number), and l is a label. The functions id, tid, lab, typ, loc, val_r, and val_w respectively return (when applicable) the k, i, l, `T`, x, v_r and v_w components of an event (or its label). We denote by $\mathcal{A}$ the set of all events, while $\mathcal{R}, \mathcal{W}$, and $\mathcal{U}$ respectively denote the set of events $a \in \mathcal{A}$ with $\mathit{typ}(a)$ being `R`, `W`, or `U`.
- A *reads-from edge* is a pair $\langle a, b\rangle \in (\mathcal{W} \cup \mathcal{U}) \times (\mathcal{R} \cup \mathcal{U})$ satisfying $\mathit{loc}(a) = \mathit{loc}(b)$ and $\mathit{val}_w(a) = \mathit{val}_r(b)$.

An *execution* G is a triple $\langle A, \mathit{po}, \mathit{rf}\rangle$ where:

- $A \subseteq \mathcal{A}$ is a finite set of events. We denote by $G.\texttt{T}_x$ the set of events $a \in A$ for which $\mathit{typ}(a) = \texttt{T}$ and $\mathit{loc}(a) = x$, while $G.\texttt{T}$ denotes the set $\bigcup_x G.\texttt{T}_x$.
- po, called *program order*, is a strict partial order on A.
- rf is a set of reads-from edges in $A \times A$.

Note that the program order of an execution may not respect the thread identifiers assigned to its events. This cannot happen in executions that are generated by programs (see Prop. 1 below).

Relating programs and executions. Figure 1 presents the semantics of sequential programs (commands in our language) as a labeled transition system. The system associates (some) *command steps* with labels in Lab. Note that in this stage the values of reads are completely arbitrary. Given the command steps, *program steps* are also labeled transitions, as presented in Fig. 2. These are associated with pairs of the form $\langle l, i\rangle \in \mathsf{Lab} \times \{1, \dots, N\}$.

To relate programs with executions, we combine the program steps with *execution steps*. The definition of execution steps, given in Fig. 3, employs the following notation:

Notation 1. Given two executions $G_1 = \langle A_1, \mathit{po}_1, \mathit{rf}_1\rangle$ and $G_2 = \langle A_2, \mathit{po}_2, \mathit{rf}_2\rangle$, with $A_1 \cap A_2 = \emptyset$, we denote by $G_1; G_2$ the execution $\langle A_1 \cup A_2, \mathit{po}_1 \cup \mathit{po}_2 \cup \mathit{po}, \mathit{rf}_1 \cup \mathit{rf}_2\rangle$, where $\mathit{po} = \{\langle a_1, a_2\rangle \in A_1 \times A_2 \mid \mathit{tid}(a_1) = \mathit{tid}(a_2)\}$. We identify an event a with the execution $\langle \{a\}, \emptyset, \emptyset\rangle$ when writing expressions like $G; a$.

Thus, each execution step labeled with l, i augments an execution G with an event a, whose label is l and thread identifier is i. Fig. 4 presents the combined semantics, where steps are either a labeled program step combined with an execution step with the same label, or an internal program step that does not affect the execution.

Definition 1. We say that G is an *execution of a program P* if $\langle P, G_\emptyset\rangle \to^* \langle P_{\text{final}}, G\rangle$, where $G_\emptyset$ denotes the empty execution (i.e., $G_\emptyset = \langle \emptyset, \emptyset, \emptyset\rangle$), and P_{final} is the program given by $\lambda i.$ `skip`.

$$\begin{array}{l||l} x := 1 & y := 1 \\ y := 2 & x := 2 \\ z_1 := 1 & z_2 := 1 \\ \texttt{wait}\ (z_2 = 1) & \texttt{wait}\ (z_1 = 1) \\ \texttt{wait}\ (x = 1) & \texttt{wait}\ (y = 1) \end{array}$$

$$\begin{tikzcd}
\langle \mathtt{W}, x, 1\rangle \arrow[d] & \langle \mathtt{W}, y, 1\rangle \arrow[d] \\
\langle \mathtt{W}, y, 2\rangle \arrow[d] & \langle \mathtt{W}, x, 2\rangle \arrow[d] \\
\langle \mathtt{W}, z_1, 1\rangle \arrow[d] & \langle \mathtt{W}, z_2, 1\rangle \arrow[d] \\
\langle \mathtt{R}, z_2, 1\rangle \arrow[d] & \langle \mathtt{R}, z_1, 1\rangle \arrow[d] \\
\langle \mathtt{R}, x, 1\rangle & \langle \mathtt{R}, y, 1\rangle
\end{tikzcd}$$

Figure 5. A program together with one of its executions. Arrows denote immediate program order edges.

Figure 5 provides an example of a program (a thread-partitioned version of the (2+2W) example from §1) and one of its executions. Note that if G is an execution of some program P then it does not include any reads-from edges, and it is well-structured with respect to tid. We refer to such executions as *plain*:

Definition 2. An execution $G = \langle A, po, rf \rangle$ is called *plain* if $rf = \emptyset$, $1 \leq tid(a) \leq N$ for every $a \in A$, and $tid(a) = tid(b)$ iff $\langle a, b\rangle \in po \cup po^{-1}$ for every two different events $a, b \in A$.

The last condition guarantees that po consists of a disjoint union of N strict total orders, one for each thread.

Proposition 1. If $\langle P, G_\emptyset\rangle \rightarrow^* \langle P', G\rangle$ then G is plain.

Consistency. Many of the executions associated with a program P are nonsensical as they can, for instance, read values never written in the program. Thus, the model restricts the attention to *consistent* executions. For the purpose of this paper, following [18], we find it technically convenient to define consistency using two properties, that we call *completeness* and *coherence*.

Definition 3. An execution $G = \langle A, po, rf \rangle$ is called *complete* if for every $b \in G.\mathtt{R} \cup G.\mathtt{U}$, we have $\langle a, b\rangle \in rf$ for some $a \in A$.

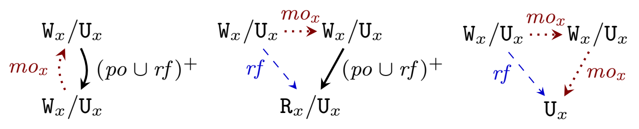
Completeness of an execution guarantees that every read/update event is justified by a corresponding write/update (recall that, by definition, reads-from edges are only from a write/update event to a read/update event with the same location and value). Obviously, not all reads-from edges are allowed. Roughly speaking, the model has to ensure that overwritten values are not read. RA does this by asserting the existence of a modification order on write accesses to the same location, as defined next.

Definition 4. Given $x \in \mathsf{Loc}$, a relation mo_x is an *x-modification order* in an execution $G = \langle A, po, rf \rangle$ if the following hold:

- mo_x is a strict total order on $G.\mathtt{W}_x \cup G.\mathtt{U}_x$.
- $mo_x; (po \cup rf)^+$ is irreflexive.
- If $\langle a, c\rangle \in rf$ and $\langle b, c\rangle \in (po \cup rf)^+ \cup mo_x$, then $\langle a, b\rangle \notin mo_x$.

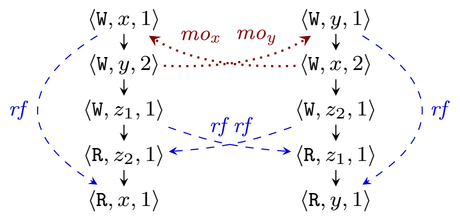
Figure 6 illustrates the conditions on mo_x imposed by this definition. First, $(po \cup rf)^+$ (that corresponds to C11's "happens-before" relation) between two write accesses enforces mo_x in the same direction. The second forbidden case corresponds to C11's coherence write-read axiom, that ensures that read/update events cannot read from an overwritten write/update event. The third case is needed to assert that update events read from the mo_x-latest event. Note that program order always appears together with the reads-from relation. This follows the basic principle of release/acquire consistency according to which reads-from edges induce synchronizations between events.

The following alternative more compact definition of an x-modification order is useful in some of the proofs below.



$\mathtt{W}_x/\mathtt{U}_x$
mo_x
$(po \cup rf)^+$
$\mathtt{W}_x/\mathtt{U}_x$
$\mathtt{W}_x/\mathtt{U}_x$
mo_x
$\mathtt{W}_x/\mathtt{U}_x$
rf
$(po \cup rf)^+$
$\mathtt{R}_x/\mathtt{U}_x$
$\mathtt{W}_x/\mathtt{U}_x$
mo_x
$\mathtt{W}_x/\mathtt{U}_x$
rf
mo_x
$\mathtt{U}_x$

Figure 6. Illustration of forbidden cases according to Def. 4.



$\langle \mathtt{W}, x, 1\rangle$
mo_x
mo_y
$\langle \mathtt{W}, y, 1\rangle$
$\langle \mathtt{W}, y, 2\rangle$
$\langle \mathtt{W}, x, 2\rangle$
rf
$\langle \mathtt{W}, z_1, 1\rangle$
rf rf
$\langle \mathtt{W}, z_2, 1\rangle$
rf
$\langle \mathtt{R}, z_2, 1\rangle$
$\langle \mathtt{R}, z_1, 1\rangle$
$\langle \mathtt{R}, x, 1\rangle$
$\langle \mathtt{R}, y, 1\rangle$

Figure 7. Evidence for RA-consistency of the execution in Fig. 5.

Proposition 2. A relation mo_x is an x-modification order in an execution $G = \langle A, po, rf \rangle$ iff it is a strict total order on $G.\mathtt{W}_x \cup G.\mathtt{U}_x$, and $po \cup rf \cup mo_x \cup fr_x$ is acyclic, where $fr_x = (rf^{-1}; mo_x) \setminus Id_A$.

Next, RA-coherent executions are those that never read "overwritten" values, and RA-consistent executions are defined by combining completeness and RA-coherence.

Definition 5. An execution $G = \langle A, po, rf \rangle$ is called *RA-coherent* if $po \cup rf$ is acyclic (i.e., $(po \cup rf)^+$ is irreflexive) and for every $x \in \mathsf{Loc}$, there exists an x-modification order in G.

Definition 6. A plain execution $G = \langle A, po, \emptyset \rangle$ is called *RA-consistent* if $G' = \langle A, po, rf \rangle$ is complete and RA-coherent for some set $rf \subseteq A \times A$ of reads-from edges.

Figure 7 presents four reads-from edges whose addition to the execution in Fig. 5 results in an RA-coherent execution. In the same figure, we also depict modification orders for each location, witnessing RA-coherence.

Finally, we define the semantics of programs under RA. The idea is to filter out the non-consistent executions among all executions of a given program. If we are left with at least one execution, then the program may terminate under RA. In the formal definition, we also support initialized locations. Thus, we assume some *initial state*, taken to be a function σ from Loc to $\mathsf{Val} \cup \{\bot\}$, where $\sigma(x) = \bot$ means that x is uninitialized. To attach an initialization part to program executions we employ the following notation.

Notation 2. Given an initial state σ, A_σ denotes the set of events $\{a_x \mid \sigma(x) \neq \bot\}$, where each a_x is the event defined by $id(a_x) = tid(a_x) = 0$ and $lab(a_x) = \langle \mathtt{W}, x, \sigma(x)\rangle$. Given an initial state σ and a plain execution $G = \langle A, po, \emptyset\rangle$, $\sigma; G$ denotes the execution $\langle A \cup A_\sigma, po \cup (A_\sigma \times A), \emptyset\rangle$.

Definition 7. A program P *may terminate under RA from an initial state* σ if $\sigma; G$ is RA-consistent for some execution G of P.

In particular, the program in Fig. 5 may terminate under RA from the initial state $\lambda x.\bot$ (no location is initialized), since its execution presented in the figure is RA-consistent.

3. SRA Memory Model

In this section we present the SRA model, a strengthening of RA. The main idea is simple: under RA, no condition relates modification orders of different locations. However, there are cases in which the behavior for each individual location follows the release/acquire paradigm, whereas the combination of the behaviors over all locations should be disallowed. More specifically, executions as the one in Fig. 7 include a cycle in $po \cup rf \cup \bigcup_x mo_x$. Forbidding such cycles is the only additional condition in SRA.

Definition 8. An execution $G = \langle A, po, rf \rangle$ is called *SRA-coherent* if there exist relations $\{mo_x\}_{x \in \mathsf{Loc}}$, such that:

- For every $x \in \mathsf{Loc}$, mo_x is an x-modification order in G.
- $po \cup rf \cup \bigcup_x mo_x$ is acyclic.

SRA-consistency is defined like RA-consistency (see Def. 6), but refers to SRA-coherence instead of RA-coherence. Similarly, termination under SRA is defined as under RA (see Def. 7), referring to SRA-consistency. (The same applies for the other model definitions that appear in this paper: we define X-coherence, and X-consistency and termination under X are defined as for RA.)

Clearly, we have that RA $\leq$ SRA, i.e., if G is SRA-coherent then it is also RA-coherent. The execution in Fig. 5 is RA-consistent but not SRA-consistent, and hence, SRA is strictly stronger than RA. Next, we show that the two models coincide for executions that do not have any write-write races.

Definition 9. Let $G = \langle A, po, rf \rangle$ be an execution. We say that two events $a, b \in A$ *race in G* if $loc(a) = loc(b)$, $a \neq b$, neither $\langle a, b \rangle \in (po \cup rf)^+$ nor $\langle b, a \rangle \in (po \cup rf)^+$, and either $a \in \mathcal{W} \cup \mathcal{U}$ or $b \in \mathcal{W} \cup \mathcal{U}$. The execution G is called *WW-race free* if no two write/update events race in G.

Proposition 3. WW-race free RA-coherent executions are also SRA-coherent.

Proof. Let $G = \langle A, po, rf \rangle$ be an RA-coherent WW-race free execution, and for every $x \in \mathsf{Loc}$, let mo_x be an x-modification order in G. Since $(po \cup rf)^+$ is total on $G.\mathtt{W}_x \cup G.\mathtt{U}_x$ and $mo_x; (po \cup rf)^+$ is irreflexive, we have that $mo_x \subseteq (po \cup rf)^+$ for every $x \in \mathsf{Loc}$. Hence, $po \cup rf \cup \bigcup_x mo_x$ is acyclic. □

The simplest syntactic way to enforce WW-race freedom in all possible executions of a given program is to disallow commands modifying the same location in different threads. All executions of programs obeying this discipline are WW-race free, and so are their extensions with reads-from edges. Except for the (2+2W) program, the examples above meet this criterion, and hence, for these programs RA and SRA exhibit exactly the same behaviors.

Furthermore, a common approach to ensure WW-race freedom when such a split does not exist is to use *critical sections*. More specifically, given a distinguished lock location l that is accessed by separate `lock()` and `unlock()` pairs of commands (implemented respectively by `when` $(l = 0)$ `do` $l := 1$ and $l := 0$), we refer to commands between consecutive lock and unlock commands as *protected*. Assuming that every command in a program P that modifies a location also modified in another thread, is protected, we have that any complete execution $G' = \langle A, po, rf \rangle$ that extends a plain execution $G = \langle A, po, \emptyset \rangle$ of P is WW-race free. Consequently, SRA and RA coincide for such programs.

3.1 Validity of Local Program Transformations

We now show that SRA does not harm compiler optimizations. We follow the development of Vafeiadis et al. [38], who studied the validity of common source-to-source program transformations under C11, and identified the set of valid reorderings of adjacent commands and eliminations of redundant commands. Here we show that SRA allows the same transformations that were proven to be sound for release/acquire accesses. The basic soundness claim is that a local transformation does not introduce new behaviors. Thus, showing correctness of such transformations amounts to providing a corresponding SRA-consistent execution of the source program for every SRA-consistent execution of the target. First, we consider the elimination of redundant adjacent accesses.

Notation 3. Given a plain execution $G = \langle A, po, \emptyset \rangle$, and an event $a \in A$, we denote by $G \setminus \{a\}$ the plain execution given by $\langle A', po \cap (A' \times A'), \emptyset \rangle$, where $A' = A \setminus \{a\}$.

Proposition 4. Let $\langle a, b \rangle$ be an immediate po-edge of a plain execution $G = \langle A, po, \emptyset \rangle$. Suppose that $loc(a) = loc(b)$, and one of the following holds:

- $a, b \in \mathcal{W}$ and $G \setminus \{a\}$ is SRA-consistent.
- $a, b \in \mathcal{R}$, $val_r(a) = val_r(b)$, and $G \setminus \{a\}$ is SRA-consistent.
- $a \in \mathcal{W}$, $b \in \mathcal{R}$, $val_w(a) = val_r(b)$, and $G \setminus \{b\}$ is SRA-consistent.

Then, G is SRA-coherent.

On the program level, Prop. 4 ensures the soundness of simplifications like:

$$\begin{array}{ccc} x := 1; x := 2 & \leadsto & x := 2 \\ \texttt{if } x = 1 \texttt{ then } y := x \texttt{ else } c & \leadsto & \texttt{if } x = 1 \texttt{ then } y := 1 \texttt{ else } c \\ x := 1; y := x & \leadsto & x := 1; y := 1 \end{array}$$

The second kind of sound transformations under RA are reordering of adjacent accesses. First, reordering of two accesses of different locations, such that at least one of them is local, is safe. Second, a write access and a consecutive read access to a different location can be safely reordered. The next proposition ensures that these transformations are sound also for SRA.

Proposition 5. Let $\langle a, b \rangle$ be an immediate po-edge of a plain execution $G = \langle A, po, \emptyset \rangle$. Suppose that $loc(a) \neq loc(b)$, the execution $\langle A, (po \setminus \{\langle a, b \rangle\}) \cup \{\langle b, a \rangle\}, \emptyset \rangle$ is SRA-consistent, and one of the following holds:

- Either $loc(a)$ or $loc(b)$ is local in G (a location x is *local* in G if po is total on $\{a \in A \mid loc(a) = x\}$).
- $a \in \mathcal{W}$ and $b \in \mathcal{R}$.

Then, G is SRA-consistent.

This allows basic reorderings, and they can be combined with the previous eliminations. For example, $x := 1; y := 1; z := x$ can be simplified to $x := 1; y := 1; z := 1$ under any context. Indeed, the resulting program executions will have three consecutive events a, b, c with labels $\langle \mathtt{W}, x, 1 \rangle$, $\langle \mathtt{W}, y, 1 \rangle$, $\langle \mathtt{W}, z, 1 \rangle$ (respectively). By the third part of Prop. 4, its consistency implies the consistency of the same execution with a new node a' between a and b whose label is $\langle \mathtt{R}, x, 1 \rangle$. Then, by the second part of Prop. 5, we have that the same execution with a reversed immediate po-edge between a' and b is also consistent. The last execution is an execution of the original program, and hence the transformation is sound.

Finally, note that from the definition of SRA (similarly to RA) it is clear that removing po-edges preserves SRA-consistency. Therefore, the basic sequentialization transformation "$c_1 \parallel c_2 \leadsto c_1; c_2$" is sound. Surprisingly, this transformation is unsound under the TSO model (see §6). Indeed, the (IRIW) program does not terminate under TSO, while its transformation that puts the two writes before the corresponding reads may terminate.

3.2 An Alternative Formulation

An equivalent definition of SRA can be obtained by requiring that there is a single modification order over *all* locations satisfying similar conditions to those of an x-modification order. In particular, this definition will be used to relate SRA and TSO in §6.

Definition 10. A relation mo is called a *modification order* in an execution $G = \langle A, po, rf \rangle$ if the following hold:

- mo is a strict total order on $G.\mathtt{W} \cup G.\mathtt{U}$.
- $mo; (po \cup rf)^+$ is irreflexive.
- If $\langle a, c \rangle \in rf$, $\langle b, c \rangle \in (po \cup rf)^+ \cup mo$, and $loc(a) = loc(b)$, then $\langle a, b \rangle \notin mo$.

Proposition 6. If mo is a modification order in G, then for every $x \in \mathsf{Loc}$, $mo_x = mo \cap ((G.\mathtt{W}_x \cup G.\mathtt{U}_x) \times (G.\mathtt{W}_x \cup G.\mathtt{U}_x))$ is an x-modification order.

Proposition 7. An execution $G = \langle A, po, rf \rangle$ is SRA-coherent iff $po \cup rf$ is acyclic and there exists a modification order in G.

4. An Equivalent Operational Model

In this section we develop a simple and intuitive operational semantics for our programming language that precisely corresponds to the SRA model. In other words, we propose a particular machine, define how it executes programs, and show that a program P may terminate under SRA iff it can be fully executed in the machine. The machine consists of N processors, each of which is running a particular program thread. Unlike program executions, machine executions are ordinary sequentially consistent executions, where steps are taken non-deterministically by different processors. In addition, the machine transitions do not refer at all to formal consistency requirements on graphs. We believe that these properties make this semantics easy to grasp by practitioners, as well as a solid foundation for adapting existing formal verification and testing methods for the SRA weak memory model.

There are two main ideas in the proposed machine structure. First, it is based on point-to-point communication. Each processor has a *local memory* and an outgoing *message buffer*, that consists of write instructions performed or observed by that processor. The processors non-deterministically choose between performing their own instructions or processing messages from other processors' buffers. When performing a read, while following some program instruction, the processor obtains the read value by inspecting its local memory. When performing a write, either when following a program instruction or when processing a message, the processor writes to its local memory and reports this write to the other processors by putting a corresponding message in its buffer. Every message can be processed at most once by every other processor, and messages should be processed in the same order in which they were issued. Thus, message buffers are taken to be lists, and each processor maintains a set of indices storing its current location in every other buffer, and constantly progressing while processing messages. Naturally, a message can be removed from a buffer when all processors have processed it.

This already gives intuitive account for the possible non-SC behavior in the store buffering example, that cannot occur in the message passing example (see §1). In a terminating run of the (SB) program, both processors follow their own instructions, ignoring the other message buffer. On the other hand, in every run of the (MP) program, for the first wait command to terminate, the second processor must process the message from the first processor that sets y to 1. Since outgoing messages are ordered, it will first process the message that sets x to 1, assigning 1 to x in its local memory. Thus, the second wait will never terminate.

Nevertheless, so far, the proposed machine fails to explain the following example (litmus test CoRR2 in [23]):

$$x := 1 \;\Big\|\; x := 2 \;\Big\|\; \begin{array}{l}\texttt{wait}\ (x = 1)\\ \texttt{wait}\ (x = 2)\end{array} \;\Big\|\; \begin{array}{l}\texttt{wait}\ (x = 2)\\ \texttt{wait}\ (x = 1)\end{array} \qquad \text{(CoRR2)}$$

Assuming that initially all variables are 0, under SRA (or, similarly, under RA or SC, as they coincide on programs with a single variable) this program cannot terminate. Indeed, in its executions, the x-modification order must order the two writes, forcing one particular order in which the two values can be observed. However, according to the description above, nothing forbids the two readers to process the messages from the two buffers in opposite orders.

To address this mismatch, the second main idea in the machine is the use of global timestamps counting the number of writes to every location. Whenever some local write is performed to a location x, the global timestamp for x is increased, and attached both to the local stored value and to the message reporting this write. When processing a write message to location x from another buffer, the processor compares the timestamp stored in its local memory for x with the timestamp of the message, and performs the write only if the message has a greater timestamp. Otherwise, the processor skips the message. For the example above, assuming that the first reader terminated, it must be the case that the $x := 2$ was performed after the $x := 1$ write (otherwise, this reader would skip the message for $x := 2$). Hence, the second reader can either process the $x := 1$ message first, and then the $x := 2$ one (exactly as the first reader), or process first the $x := 2$ message, but then it is forced to skip the older message, and cannot observe the value 1.

As shown below, a machine based on these two simple ideas provides a precise account for SRA. Note that this operational model is too strong for RA. Indeed, getting back to the (2+2W) example (see Fig. 5), assuming that the first processor terminated and the second arrived just before the last wait, note that before processing the $z_2 := 1$ message, the first processor must have processed the $x := 2$ message before its own $x := 1$ instruction, or skipped the $x := 2$ message. Thus, $x := 2$ had a smaller timestamp than $x := 1$, and so, $y := 1$ had a smaller timestamp than $y := 2$. This implies, however, that between performing the local $y := 1$ write instruction and processing the $z_1 := 1$ message, the second processor must have processed the $y := 2$ message, and replace the value locally stored for y from 1 to 2. Thus, the last wait of the second processor cannot terminate.

Next, we turn to the formal development of the operational semantics and its soundness and completeness proof. We define the machine as a labeled transition system: a set of states and a labeled transition relation. A *machine state* takes the form of a pair $[S, T]$, where S assigns a *processor state* to every processor, and T is the global *timestamps* table. In turn, a processor state consists of a local memory M, an outgoing message buffer L, and a function I that records the current location in every other message buffer. For every $1 \leq i \leq N$, $I(i)$ is the index of the first message in the buffer of processor i which was not yet observed by the current processor.

Definition 11. A *processor state* is a tuple $\langle M, L, I \rangle$, where:

- $M : \mathsf{Loc} \to ((\mathsf{Val} \times \mathbb{N}) \cup \{\langle \bot, 0 \rangle\})$ is the *local memory*. We write $M(x) = v@t$ for $M(x) = \langle v, t \rangle$, and $M(x) = \bot@0$ means that x is uninitialized.
- L is an *outgoing message buffer* that takes the form of a list of messages, where a *message* m has the form $\lfloor x := v@t \rfloor$, where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and $t \in \mathbb{N}$. We use ";" for concatenation of such lists (and write, e.g., $m; L$ or $L; m$), and often refer to L as an array. $L[k]$ denotes the kth message in L (starting from 1), and $|L|$ is the number of messages in L.
- $I : \{1, \dots, N\} \to \mathbb{N}^+$ is a function assigning an index in every other message list.

Definition 12. A *machine state* is a pair of the form $[S, T]$, where S is a function assigning a processor state to every processor $1 \leq i \leq N$, and $T : \mathsf{Loc} \to \mathbb{N}$ assigns a timestamp to every location. Given an initial state σ, the *initial machine state* $[S_\sigma, T_\emptyset]$ is given by $S_\sigma = \lambda i.\ \langle M_\sigma, L_\emptyset, I_\emptyset \rangle$ and $T_\emptyset = \lambda x.\ 0$, where $M_\sigma = \lambda x.\ \sigma(x)@0$, $L_\emptyset = \epsilon$, and $I_\emptyset = \lambda i.\ 1$.

The machine semantics is given in Fig. 8. Its steps follow the informal description above: (READ) steps read from the local memory; (WRITE) steps write to the local memory, add a corresponding message, and increment the global stored timestamp for the specific location; (PROCESS) steps pull a message from some buffer with a timestamp that is greater than the local stored one, perform the local write, and increment the current location in the buffer; (SKIP) steps

$$\text{(READ)}\ \frac{\begin{array}{c} S(i) = \langle M, -, - \rangle \\ M(x) = v@- \end{array}}{[S,T] \xrightarrow{\langle \mathtt{R}, x, v \rangle, i} [S,T]} \qquad \text{(WRITE)}\ \frac{\begin{array}{c} S(i) = \langle M, L, I \rangle \qquad T(x) = t \\ M' = M[x \mapsto v@t+1] \qquad L' = L; \lfloor x := v@t+1 \rfloor \\ T' = T[x \mapsto t+1] \end{array}}{[S,T] \xrightarrow{\langle \mathtt{W}, x, v \rangle, i} [S[i \mapsto \langle M', L', I \rangle], T']}$$

$$\text{(UPDATE)}\ \frac{\begin{array}{c} S(i) = \langle M, L, I \rangle \qquad T(x) = t \qquad M(x) = v_r@t \\ M' = M[x \mapsto v_w@t+1] \qquad L' = L; \lfloor x := v_w@t+1 \rfloor \\ T' = T[x \mapsto t+1] \end{array}}{[S,T] \xrightarrow{\langle \mathtt{U}, x, v_r, v_w \rangle, i} [S[i \mapsto \langle M', L', I \rangle], T']}$$

$$\text{(PROCESS)}\ \frac{\begin{array}{c} i \neq j \qquad S(i) = \langle M, L, I \rangle \qquad S(j) = \langle -, L_j, - \rangle \\ I(j) = k \qquad k \leq |L_j| \qquad L_j[k] = \lfloor x := v@t_j \rfloor \\ M(x) = -@t_i \qquad t_i < t_j \\ M' = M[x \mapsto v@t_j] \qquad L' = L; \lfloor x := v@t_j \rfloor \qquad I' = I[j \mapsto k+1] \end{array}}{[S,T] \to [S[i \mapsto \langle M', L', I' \rangle], T]}$$

$$\text{(SKIP)}\ \frac{\begin{array}{c} i \neq j \qquad S(i) = \langle M, L, I \rangle \qquad S(j) = \langle -, L_j, - \rangle \\ I(j) = k \qquad k \leq |L_j| \qquad L_j[k] = \lfloor x := -@t_j \rfloor \\ M(x) = -@t_i \qquad t_j \leq t_i \\ I' = I[j \mapsto k+1] \end{array}}{[S,T] \to [S[i \mapsto \langle M, L, I' \rangle], T]}$$

$$\text{(CLEAN)}\ \frac{\begin{array}{c} S(i) = \langle M, m; L, I \rangle \qquad \forall j \neq i.\ S(j) = \langle M_j, L_j, I_j \rangle \qquad \forall j \neq i.\ I_j(i) > 1 \\ \forall j \neq i.\ S'(j) = \langle M_j, L_j, I_j[i \mapsto I_j(i) - 1] \rangle \qquad S'(i) = \langle M, L, I \rangle \end{array}}{[S,T] \to [S', T]}$$

Figure 8. Machine steps

$$\frac{P \xrightarrow{l,i} P' \qquad [S,T] \xrightarrow{l,i} [S',T']}{\langle P, [S,T] \rangle \to \langle P', [S',T'] \rangle}$$

$$\frac{P \to P'}{\langle P, [S,T] \rangle \to \langle P', [S,T] \rangle} \qquad \frac{[S,T] \to [S',T']}{\langle P, [S,T] \rangle \to \langle P, [S',T'] \rangle}$$

Figure 9. Program and machine combined steps

pull a message from some buffer with a timestamp that is less than or equal to the local stored one, and skips the message by just incrementing the current location in the buffer; and (CLEAN) steps remove the first message in a buffer, provided that all processors have already processed or skipped it. Finally, (UPDATE) steps (which were not explained above) naturally combine reads and writes. In addition, for performing (UPDATE) the processor should have the most recent value of the updated location x. Thus, it verifies that the timestamp t stored in its memory for x ($M(x) = v_r@t$) is equal to the global timestamp of x ($T(x) = t$). Otherwise, the processor cannot continue and is forced to pull messages from other processors. This corresponds to the fact that SRA update events (exactly as RA ones, and unlike plain read events) should read from their immediate predecessor in the relevant modification order.

The combined machine and program semantics is given in Fig. 9. Note that it is defined exactly as the combined execution and program semantics (see Fig. 4), using machine steps instead of execution steps, and including internal machine steps that do not affect the program ((PROCESS), (SKIP), and (CLEAN) steps).

Theorem 1 (Soundness and Completeness)**.** A program P may terminate under SRA from an initial state σ iff $\langle P, [S_\sigma, T_\emptyset] \rangle \to^* \langle P_{\text{final}}, [S,T] \rangle$ for some machine state $[S,T]$.

A straightforward application of this theorem is to provide alternative accounts for the soundness of the source-to-source transformations mentioned in §3.1. Using the operational model, proving validity of such transformations becomes a more ordinary task: one should show that the behavior of the machine on the resulting program is also possible when running the original one. For example, we give an informal argument for the soundness of the transformation that eliminates the first assignment command in a pair of the form $x := v; x := v'$. To simulate an execution of the resulting program, the machine running the original program can perform the two writes in two consecutive steps when the resulting program performs the second write, and pulls the two corresponding messages in consecutive steps whenever the machine running the resulting program pulls the $x := v'$ message.

Note that two possible variants of the machine are equivalent to our definition. First, if we omit the (CLEAN) rule, then the local memory is uniquely defined by the message buffer, and instead of inspecting the value (and timestamp) of a location x in the memory, each processor can check for the last message modifying x that appears in its buffer. The second alternative is to have one message queue between every (ordered) pair of processors. In this case, there is no need for pointers in message lists. Instead, every write message is enqueued to all outgoing queues, and the processors asynchronously dequeue messages from their incoming queues.

Soundness and completeness proof. As the main tool in the proof of Thm. 1, we introduce the notion of a *history*, that intuitively correspond to a log of a machine execution. Formally, a history is a totally ordered set of *actions* – expressions of the form $a@i$ where a is an event and i is a thread identifier, such that $a \in \mathcal{W} \cup \mathcal{U}$ whenever $tid(a) \neq i$. An action $a@i$ is standing for "processor i observes event a". The event a can either be an event originated by thread i ($tid(a) = i$), or a write/update event of another thread (as reads are not reported in message buffers). Obviously, not every order of actions is possible. Next, we define *legal* histories.

Definition 13. A history $H = \langle B, \leq \rangle$ induces the following functions:

1. $H.before : \mathcal{R} \cup \mathcal{U} \to \mathcal{P}(\mathcal{W} \cup \mathcal{U})$ is given by $H.before(a) = \{b \in \mathcal{W} \cup \mathcal{U} \mid b@tid(a) < a@tid(a), loc(b) = loc(a)\}$.
2. $H.last : \mathcal{R} \cup \mathcal{U} \rightharpoonup \mathcal{W} \cup \mathcal{U}$ is a partial function assigning a write/update event b to every read/update event a such that $b@tid(b) = \max_{\leq}\{c@i \in B \mid c \in H.before(a), i = tid(c)\}$. If $H.before(a)$ is empty, then $H.last(a)$ is undefined.

Definition 14. A history $H = \langle B, \leq \rangle$ is called *legal* if the following hold:

1. $val_r(a) = val_w(H.last(a))$ for every $a \in dom(H.last)$.
2. For every $a \in \mathcal{W} \cup \mathcal{U}$ and $b@i \in B$, if $a@tid(b) < b@tid(b)$, then $a@i \in B$ and $a@i < b@i$.
3. For every $a \in \mathcal{W} \cup \mathcal{U}$ and $b \in \mathcal{U}$, if $loc(a) = loc(b)$ and $a@tid(a) < b@tid(b)$, then $b \in dom(H.last)$ and $a@tid(a) \leq H.last(b)@tid(H.last(b))$.
4. $a@tid(a) \leq a@i$ for every $a@i \in B$.

The first condition ensures that the performed reads/updates obtain the values of the latest write/update (to the relevant location) that was observed by the current thread. The second condition corresponds to the fact that message buffers are ordered—before thread i observes an event b originated by thread j, it must observe every event a that was observed by j before it originated b. The third requirement guarantees that updates are performed only while having the latest value. The last condition ensures that events are observed by other threads only after their origination. Next, we relate histories to a given plain execution and initial state.

Definition 15. Let $G = \langle A, po, \emptyset\rangle$ be a plain execution, and σ be an initial state. A history $H = \langle B, \leq\rangle$ is called:

- *G-suitable* if the following hold:
  1. For every $a@i \in B$, we have $a \in A$.
  2. For every $a \in A$, we have $a@tid(a) \in B$.
  3. For every $\langle a, b\rangle \in po$, we have $a@tid(a) < b@tid(b)$.
- *σ-suitable* if $val_r(a) = \sigma(loc(a))$ whenever $a@tid(a) \in B$ and $a \in (\mathcal{R} \cup \mathcal{U}) \setminus dom(H.last)$.
- *$\langle G, \sigma\rangle$-legal* if it is legal, G-suitable and σ-suitable.

The next key theorem is the formal correspondence between SRA-consistency and legal histories.

Theorem 2. Let G be a plain execution, and σ be an initial state. Then, $\sigma; G$ is SRA-consistent iff there exists a $\langle G, \sigma\rangle$-legal history.

Proof sketch. We consider here the case that $\sigma = \lambda x.\ \bot$. Assuming $G = \langle A, po, \emptyset\rangle$ is SRA-consistent, choose rf such that $G' = \langle A, po, rf\rangle$ is a complete and SRA-coherent execution, and let mo be a modification order in G' (see Prop. 7). For every $1 \leq i \leq N$, let $A_i = \{a \in A \mid tid(a) = i\}$, $A'_i = A_i \cup G.\mathtt{W} \cup G.\mathtt{U}$, and $\prec_i = ((hb \cap (A'_i \times A'_i)) \cup ((A_i \times (A'_i \setminus A_i)) \setminus hb^{-1}))^+$, where $hb = (po \cup rf)^+$. In addition, let $B = \{a@i \mid 1 \leq i \leq N, a \in A'_i\}$, $B' = \{a@i \in B \mid tid(a) = i, a \in \mathcal{W} \cup \mathcal{U}\}$, and define the following relations:

1. $T_1 = \{\langle a@i, b@j\rangle \in B \times B \mid i = j, a \prec_i b\}$.
2. $T_2 = \{\langle a@i, b@j\rangle \in B' \times (B \setminus B') \mid a = b\}$.
3. $T_3 = \{\langle a@i, b@j\rangle \in B' \times B' \mid \langle a, b\rangle \in mo\}$.

We show that $T_1 \cup T_2 \cup T_3$ is acyclic, take $\leq$ to be a total order on B extending $(T_1 \cup T_2 \cup T_3)^+$, and prove that H is $\langle G, \sigma\rangle$-legal.

For the converse, given a $\langle G, \sigma\rangle$-legal history $H = \langle B, \leq\rangle$, we show that $G' = \langle A, po, H.rf\rangle$ is a complete execution, where $H.rf = \{\langle a, b\rangle \in \mathcal{A} \times \mathcal{A} \mid H.last(b) = a\}$. To prove that it is also SRA-coherent, we choose $mo = \{\langle a, b\rangle \in (G.\mathtt{W} \cup G.\mathtt{U}) \times (G.\mathtt{W} \cup G.\mathtt{U}) \mid a@tid(a) < b@tid(b)\}$, and use Prop. 7. □

Roughly speaking, using Theorem 2, the soundness proof proceeds by showing that a log of a terminating execution of the machine from an initial state $[S_\sigma, T_\emptyset]$ of a program P forms a $\langle G, \sigma\rangle$-legal history, where G is an execution of P. The completeness proof makes use of the other direction of Theorem 2, and shows that histories are "executable", i.e., every $\langle G, \sigma\rangle$-legal history, where G is an execution of a program P, can be followed by the machine executing P starting from $[S_\sigma, T_\emptyset]$.

5. Fence Commands

In this section, we extend our programming language with a memory fence command, denoted by `fence()`, and show that these commands can be used to enforce sequential consistency under RA (or SRA). Our semantics for fence commands is as if they were atomic assignments (RMWs) to a particular otherwise unused location. In other words, `fence()` is simply syntactic sugar for the atomic assignment $\langle f := 0\rangle$, where f is a distinguished location that is not mentioned in any non-fence command. As a result, `fence()` instructions induce *fence events*, that is events with labels $\langle \mathtt{U}, f, 0, 0\rangle$. We further require that any initial state σ has $\sigma(f) = 0$.

Accordingly, executions of programs (together with the initialization part) are all well-formed, as defined next:

Definition 16. An execution $G = \langle A, po, rf\rangle$ is *well-formed* if the set $A_f = \{a \in A \mid loc(a) = f\}$ consists only of fence events, except for one event a with $lab(a) = \langle \mathtt{W}, f, 0\rangle$ that initializes f (i.e., $\langle a, b\rangle \in po$ for every $b \in A_f \setminus \{a\}$).

The fundamental property, that allows the reduction to SC, is that fence events are totally ordered by rf^+ as shown by the following proposition.

Proposition 8. Given a well-formed complete RA-coherent execution $G = \langle A, po, rf\rangle$, the relation rf^+ is total on the set of fence events in G.

Based on this property, we show that our fences are sufficient for restoring SC. More specifically, we call an event *G-racy* if it races with some other event in an execution G (see Def. 9), and show that having a fence event between every two $(po \cup rf)^+$-related racy events in a well-formed complete RA-coherent execution means that the execution is also SC-coherent. Following Shasha and Snir [31], we employ the following definition of SC-coherence:

Definition 17. An execution $G = \langle A, po, rf\rangle$ is called *SC-coherent* if there exist relations $\{mo_x\}_{x \in \mathsf{Loc}}$, such that:

- For every $x \in \mathsf{Loc}$, mo_x is an x-modification order in G.
- The relation $po \cup rf \cup \bigcup_x mo_x \cup \bigcup_x fr_x$ is acyclic, where $fr_x = (rf^{-1}; mo_x) \setminus Id_A$.

Theorem 3. Let $G = \langle A, po, rf\rangle$ be a well-formed complete RA-coherent execution. Suppose that for every two G-racy events a, b, if $loc(a) \neq loc(b)$, and $\langle a, b\rangle \in po \cup (po; (po \cup rf)^*; po)$, then $\langle a, c\rangle, \langle c, b\rangle \in (po \cup rf)^+$ for some fence event c. Then, G is SC-coherent.

A simple corollary of this theorem is a per-execution data race freedom (DRF) property:

Corollary 1. Well-formed complete RA-coherent executions that have no racy events are also SC-coherent.

The simplest way to enforce executions that satisfy the condition of Thm. 3 in a given program is for each thread to include a fence command between every two accesses of different shared variables. In general, all these fences are necessary: the SB program shows fences may be needed between writes and subsequent reads, IRIW shows that fences are needed between two reads, and the following two programs show that fences are required between two writes, and between a read and a subsequent write. Assuming all variables are initialized to 0, these programs (without the crossed out fences) may terminate under RA (or SRA) but not under SC.

$$\begin{array}{l||l||l}
x := 1 & y := 2 & \texttt{wait}\ (y = 1) \\
\cancel{\texttt{fence()}} & \texttt{fence()} & \texttt{fence()} \\
y := 1 & \texttt{wait}\ (x = 0) & \texttt{wait}\ (y = 2)
\end{array}$$

$$\begin{array}{l||l||l||l}
 & \texttt{wait}\ (x = 1) & y := 2 & \texttt{wait}\ (y = 1) \\
x := 1 & \cancel{\texttt{fence()}} & \texttt{fence()} & \texttt{fence()} \\
 & y := 1 & \texttt{wait}\ (x = 0) & \texttt{wait}\ (y = 2)
\end{array}$$

It is worth contrasting Thm. 3 with the corresponding one for TSO. Under TSO, fences between every shared variable write and every subsequent shared variable read suffice to restore SC [24]. For a particular common class of programs, however, we can get a reduction to SC that requires fewer fences similar to those required for TSO. This is based on the following theorem:

Theorem 4. Let $G = \langle A, po, rf\rangle$ be a well-formed complete RA-coherent execution. Assume that G is WW-race free and there exists a set $B \subseteq A$ of *protected events* such that the following hold:

1. $(po \cup rf)^+$ is total on B.
2. If a races with b in G, then either $a \in B$ or $b \in B$.
3. For every G-racy write/update event $a \in B$ and G-racy read event $b \in B$, if $loc(a) \neq loc(b)$ and $\langle a, b\rangle \in (po \cup rf)^+$, then $\langle a, c\rangle, \langle c, b\rangle \in (po \cup rf)^+$ for some fence event c.
4. For every G-racy write/update event $a \notin B$ and G-racy read event $b \notin B$, if $loc(a) \neq loc(b)$ and $\langle a, b\rangle \in (po \cup rf)^+$, then $\langle a, c\rangle, \langle c, b\rangle \in (po \cup rf)^+$ for some fence *or* protected event c.

Then, G is SC-coherent.

Theorem 4 can be applied in various cases, e.g., two-threaded programs (by taking B to consist of the events of one of the threads), and concurrent data structures with a single writer and multiple readers (by taking B to consist of the writer events).

A direct use of Thm. 4 is for programs that include critical sections (see the discussion after Prop. 3 in §3). First, we lift the definitions of racy events to the program level by over-approximating the set of racy events: Say that two commands *race on x* if they both mention x, appear on different threads, at least one of them modifies x, and at least one of them is unprotected (appears outside a critical section). In addition, we say that a command is an *x-racy read command* if it reads x and races on x with some other command; and similarly, a command is an *x-racy write command* if it modifies x and races on x with some other command. Then, obeying the following conditions suffice to guarantee SC behavior:

(i) Every command modifying a location also modified in another thread, is protected.

(ii) No two unprotected commands race (on some location).

(iii) There exist fence commands:

- Outside critical sections, between every x-racy write command and subsequent y-racy read command for $x \neq y$.
- Inside critical sections, between every x-racy write command and subsequent y-racy read command for $x \neq y$.
- Either after the last racy write command or before the first racy read command inside all critical sections.

Indeed, suppose that a program P obeying this discipline may terminate under RA. Let $G' = \langle A, po, rf\rangle$ be an RA-coherent complete execution that extends a plain execution $G = \langle A, po, \emptyset\rangle$ of P. Condition (i) ensures that G' is WW-race free. We take the set B of protected events to be the set of all events induced by the commands in the critical sections. Then, $(po \cup rf)^+$ is total on B. Further, we can assume that all `lock()` commands immediately succeed and induce a single update event in G' (otherwise, remove the read events that correspond to failed attempts to acquire the lock, and G' remains an RA-coherent complete execution that extends an execution of P). Condition (ii) ensures no races between two events outside B. Additionally, conditions (ii)-(iii) guarantee that conditions 3 and 4 of Thm. 4 are met. The theorem implies that G' is SC-coherent, and hence P may terminate under SC.

5.1 Verifying a Read-Copy-Update Implementation

We demonstrate the application of Thm. 4 to a practical weak memory algorithm, the user-mode read-copy-update (RCU) implementation of Desnoyers et al. [13].

RCU is a mechanism, deployed heavily in the Linux kernel, that allows a single writer to manipulate a data structure, such as a linked list or a binary search tree, while multiple readers are concurrently accessing it. To ensure that a single writer updates the data structure at any given time, a global lock is used to protect the writes. To ensure correctness of the concurrent readers, the writer instead of directly modifying a piece of the structure, first copies that piece, modifies the copy, and then makes the new copy accessible and the old one inaccessible. To deallocate the inaccessible piece, it invokes the `synchronize_rcu()` procedure, which waits for all the readers to stop accessing the old copy.

```
   rcu_quiescent_state():
L1:  ~~fence();~~      // fence removed
L2:  rc[get_my_tid()] := gc;
L3:  fence();
   rcu_thread_offline():
L4:  ~~fence();~~      // fence removed
L5:  rc[get_my_tid()] := 0;
L6:  fence();     // fence inserted
   rcu_thread_online():
L7:  rc[get_my_tid()] := gc;
L8:  fence();
   synchronize_rcu():
L10:   local was_online := (rc[get_my_tid()] ≠ 0);
L11:   ~~fence();~~              // fence removed
L12:   if was_online then rc[get_my_tid()] := 0;
L13:   lock();
L14:   gc := gc + 1;
L15:   ~~barrier();~~ fence();  // barrier replaced by fence
L16:   for i := 1 to N do
L17:     wait (rc[i] ∈ {0,gc});
L18:   unlock();
L19:   if was_online then rc[get_my_tid()] := gc;
L20:   fence();
```

Figure 10. A user-mode read-copy-update implementation based on [13], where all variable accesses are release/acquire accesses. The fences are shown as in the original program: `fence()` is a memory fence, while `barrier()` is a compiler fence. In our variant, we replace the compiler fence with a (stronger) memory fence, insert a fence on line 6, and remove the unnecessary fences.

The readers, on the other hand, do not acquire any locks. They only need to periodically call `rcu_quiescent_state()` when they are not accessing the data structure and not keep any internal pointers to the data structure across calls to `rcu_quiescent_state()`. Alternatively, the readers may also call `rcu_thread_offline()` when they have stopped accessing the data structure and call `rcu_thread_online()` before accessing it again. The second alternative is slightly more expensive but avoids the need of having to call `rcu_quiescent_state()` every once in a while.

Figure 10 shows the implementations of the four aforementioned synchronization methods. The threads synchronize via two variables: the global counter, `gc`, and an array of read counters, `rc`, with one entry for each thread, and use a global lock to serialize accesses to the `synchronize_rcu` method.

RCU programs, based on the implementation in Fig. 10, follow the syntactic discipline that allows us to apply Thm. 4.

(i) The only commands that modify a location also modified in another thread, are the assignment to `gc` on line 14, and the writer's updates of the RCU-protected data structure, which are all protected by the lock.

(ii) Restricting attention to unprotected commands (lines 2, 5, 7, 10, 12, 19), the only locations that are modified are the `rc[i]`'s and `was_online`, but each of them is accessed only by one particular thread. The RCU-protected data structure itself is modified only by protected commands.

(iii) Outside the critical sections, there is a fence immediately after every racy write command (namely, the assignments to `rc[get_my_tid()]` on lines 2,5,7,19).

(iv) Within the critical section of `synchronize_rcu()` (lines 14-17), there is a fence immediately after every assignment

$$\begin{tikzcd}
a : \mathtt{W}_x/\mathtt{U}_x \arrow[rr, dotted, "mo"] \arrow[dr, dashed, "rf"'] & & b : \mathtt{W}_x/\mathtt{U}_x \arrow[rr, dotted, "mo"] & & \mathtt{W}_y/\mathtt{U}_y \arrow[dl, dashed, "rf \setminus po"] \\
& c : \mathtt{R}_x/\mathtt{U}_x & & \mathtt{R}_y/\mathtt{U}_y \arrow[ll, "po"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
a : \mathtt{W}_x/\mathtt{U}_x \arrow[r, dotted, "mo"] \arrow[d, dashed, "rf"'] & b : \mathtt{W}_x/\mathtt{U}_x \arrow[d, dotted, "mo"] \\
c : \mathtt{R}_x/\mathtt{U}_x & \mathtt{U}_y \arrow[l, "po"]
\end{tikzcd}$$

Figure 11. Illustration of the additional forbidden cases under TSO according to Thm. 5.

(namely, the assignment to gc on line 14), and before the first racy read (line 17).

(v) The critical section modifying the RCU-protected data structure has no racy reads.

Therefore, by Thm. 4, RCU programs have the same behaviors under SC as under RA (and SRA).

6. Relation to TSO

In this section we study the relationship between SRA and TSO, the *total store ordering* memory model provided by the x86 and SPARC architectures. TSO is known to be stronger than RA. We show that it is also (strictly) stronger than SRA. This entails that the ordinary compilation of C11 release/acquire accesses to TSO actually also guarantees SRA consistency.

To relate SRA and TSO, we use the declarative model of TSO of Owens et al. [25] that was used to prove correctness of the compilation of RA to x86-TSO.

Definition 18. A relation *tso* is called a *total store order* for an execution $G = \langle A, po, rf \rangle$ if it satisfies the following:

- *tso* is a strict partial order on A, that is total on $G.\mathtt{W} \cup G.\mathtt{U}$.
- $po \subseteq tso \cup (G.\mathtt{W} \times G.\mathtt{R})$ and $rf \subseteq tso \cup po$.
- If $\langle a, c\rangle \in rf$, $\langle b, c\rangle \in po \cup tso$, $b \in \mathcal{W} \cup \mathcal{U}$, and $loc(a) = loc(b)$, then $\langle a, b\rangle \notin tso$.

G is called *TSO-coherent* if there exists a total store order for it.

Note that unlike modification orders, the total store order relates also read events. Next, we provide an equivalent characterization that is based on a modification order and has a similar nature to the formulation of SRA given in Prop. 7 (see also Fig. 11).

Theorem 5. An execution $G = \langle A, po, rf \rangle$ is TSO-coherent iff $po \cup rf$ is acyclic, and there exists a modification order mo in G (see Def. 10) that satisfies the following additional condition:

- If $\langle a, c\rangle \in rf$, $\langle b, c\rangle \in ((mo; (rf \setminus po)) \cup (mo \cap (A \times \mathcal{U})))$; po, and $loc(a) = loc(b)$, then $\langle a, b\rangle \notin mo$.

Proof sketch. For one direction, we take $mo = tso \cap ((\mathcal{W} \cup \mathcal{U}) \times (\mathcal{W} \cup \mathcal{U}))$, and show that mo satisfies the required conditions. For the converse, we show that $(mo \cup (po \setminus (\mathcal{W} \times A)) \cup (rf \setminus po))^+$ is a total store order for G. □

Intuitively speaking, the modification order of Thm. 5 is the order in which the writes propagate to the main memory of the TSO-machine. When a processor of a TSO-machine reads from a location x, it obtains the value of the last write instruction to x that appears in its local store buffer, and if no such instruction exists, it obtains the value of the write to x that was the *last* to propagate to the memory. The condition on the pair $\langle b, c\rangle$ in the theorem ensures that the reads-from edge $\langle a, c\rangle$ corresponds to reading from the main memory, rather than from the local buffer. Note that taking just $\langle b, c\rangle \in ((mo; rf) \cup mo); po$ instead of $\langle b, c\rangle \in ((mo; (rf \setminus po)) \cup (mo \cap (A \times \mathcal{U}))); po$ in the condition given in Thm. 5 results in another formulation of SC.

Using this theorem and Prop. 7, the relation to SRA is an immediate corollary: We have that SRA $\leq$ TSO, i.e., if G is TSO-coherent then it is also SRA-coherent. Note that SRA is strictly weaker than TSO (even for WW-race free executions): the IRIW program (presented in §1) may terminate under SRA but not under TSO. In addition, as the following example shows, two threads suffice for observing the difference between SRA and TSO.

$$\begin{array}{l||l}
x := 1 & y := 1 \\
\langle f_1 := 1 \rangle & \langle f_2 := 1 \rangle \\
\mathtt{wait}\ (y = 0) & \mathtt{wait}\ (x = 0)
\end{array} \qquad \text{(SBU)}$$

This program may terminate under SRA but not under TSO (where initially all variables are 0). However, if we restrict our attention to executions without update events, then SRA and TSO coincide for two threaded executions. To see this, it suffices to note that $mo; (rf \setminus po) \subseteq (po \cup rf)^+$ for such executions.

An important consequence of the relation to SRA is that the same compilation scheme for RA (that maps writes, reads, and updates to plain TSO write, read, and RMW instructions [6]) guarantees SRA. Therefore, SRA has no additional implementation cost over RA on TSO-machines. Additionally, note that our fence instructions (updates to an unused location, see §5) can be compiled directly to an `mfence` instruction on x86-TSO. The correctness is obvious: Compiling the atomic assignment $\langle f := 0\rangle$ to a suitable x86-TSO RMW instruction (such as `lock xchg`) is sound. But then, according to the operational x86-TSO model, if we ignore the value of location f, the effect of an `mfence` and a RMW on f is identical: they wait for the store buffer to be flushed.

Next, we identify a condition that ensures that RA and TSO coincide.

Theorem 6. Let $G = \langle A, po, rf \rangle$ be a WW-race free RA-coherent execution. Suppose that $G.\mathtt{U} = \emptyset$, and there exists a set $B \subseteq A$ of protected events such that the following hold:

- $(po \cup rf)^+$ is total on B.
- If a races with b in G, then either $a \in B$ or $b \in B$.
- If $\langle a, b\rangle \in rf \setminus po$, then either $a \in B$ or $b \in B$.

Then, G is TSO-coherent.

Remark 1. As the following example shows, the claim of Thm. 6 does not hold if we allow even a single update to an otherwise unused location:

$$\begin{array}{l||l||l}
x := 1 & & \\
z_1 := 1 & \mathtt{wait}\ (z_1 = 1) & y := 1 \\
\mathtt{wait}\ (z_2 = 1) & z_2 := 1 & \langle f := 1 \rangle \\
\mathtt{wait}\ (y = 0) & & \mathtt{wait}\ (x = 0)
\end{array}$$

This program may terminate under RA but not under TSO (where initially all variables are 0). In fact, under RA (or SRA), adding one update event to an otherwise unused location has no effect (whereas in the TSO-machine it forces the buffer to be flushed). This may indicate that the current semantics of updates in TSO is too strong, and weaker update instructions, as studied in [26], can be useful.

A consequence of Thm. 6 is that TSO and RA coincide for update-free "client-server programs". We say that a program follows the *client-server* discipline iff its locations can be partitioned into disjoint sets $X_1, \ldots, X_N$ such that thread S ("the server") writes only to variables in X_{S}, and each ("client") thread $i \neq \mathrm{S}$ reads only from variables in $X_{\mathrm{S}} \cup X_i$ and writes only to variables in X_i. In other words, client threads cannot communicate directly with other client threads, but only via the distinguished server thread. By taking the set B to be the set of events generated by the server, we can apply Thm. 6 for executions of such programs. As an example, Thm. 6 can be applied to a fence-free implementation of a simple version of the RCU mechanism (presented in Fig. 12). This

```
                                synchronize_rcu():
  rcu_quiescent_state():         gc := gc + 1;
   rc[get_my_tid()] := gc;       for i := 1 to N do
                                   wait (rc[i] = gc);
```

Figure 12. Simple RCU implementation

$$\begin{aligned}
G.isync &= \{\langle a,b\rangle \mid \exists c \in G.\texttt{isync}.\ \langle a,c\rangle \in deps \wedge \langle c,b\rangle \in po\} \\
G.lwsync &= \{\langle a,b\rangle \mid \exists c \in G.\texttt{lwsync}.\ \langle a,c\rangle \in po \wedge \langle c,b\rangle \in po\} \\
G.sync &= \{\langle a,b\rangle \mid \exists c \in G.\texttt{sync}.\ \langle a,c\rangle \in po \wedge \langle c,b\rangle \in po\} \\
G.ppo &= (deps \cup G.isync) \cap ((\mathcal{R} \times (\mathcal{R} \cup \mathcal{W})) \\
G.fence &= G.sync \cup (G.lwsync \cap ((\mathcal{R} \times (\mathcal{R} \cup \mathcal{W})) \cup (\mathcal{W} \times \mathcal{W})) \\
G.rfe &= rf \setminus po \\
G.hb &= G.ppo \cup G.fence \cup G.rfe \\
G.base &= G.rfe^?; G.fence; G.hb^* \\
G.po\text{-}aa &= po \cap ((At \cap \mathcal{W}) \times (At \cap \mathcal{W})) \\
G.rmw &= ipo \cap ((At \cap \mathcal{R}) \times (At \cap \mathcal{W}))
\end{aligned}$$

Figure 13. Auxiliary notations for a Power execution $G = \langle A, po, deps, rf, At\rangle$. *ipo* denotes the set of immediate *po*-edges.

$$\begin{aligned}
(\!|x|\!) &= \texttt{lwz}\ r_0, x \cdot \texttt{cmpw}\ r_0, r_0 \cdot \texttt{beq}\ L \cdot L\!: \cdot \texttt{isync} \\
(\!|x := v|\!) &= \texttt{li}\ r_0, v \cdot \texttt{lwsync} \cdot \texttt{stw}\ r_0, x \\
& \quad \texttt{lwsync} \cdot Loop\!: \cdot \\
(\!|\langle x := x+1\rangle|\!) &= \texttt{lwarx}\ r_0, 0, x \cdot \texttt{addi}\ r_0, r_0, 1 \cdot \texttt{stwcx.}\ r_0, 0, x \cdot \\
& \quad \texttt{bne}\ Loop \cdot \texttt{cmpw}\ r_0, r_0 \cdot \texttt{beq}\ L \cdot L\!: \cdot \texttt{isync} \\
(\!|\texttt{fence()}|\!) &= \texttt{sync}
\end{aligned}$$

Figure 14. Compilation of reads, writes, atomic increments and fences to Power

version does not support readers going offline (and thus, requires them to constantly periodically call `rcu_quiescent_state()`), and assumes that one particular thread serves as the writer (that calls `synchronize_rcu()`). This allows reasoning about correctness under RA using existing techniques for TSO.

7. Relation to Power

In this section, we show that the SRA memory model is *equivalent* to the Power memory model of Alglave et al. [4] when following the standard compilation scheme of C11 release/acquire atomics to Power [7, 29], i.e., inserting an `lwsync` (lightweight synchronization) fence before every write and a conditional branch followed by an `isync` fence after every read. Atomic updates are compiled into two consecutive 'atomic' accesses: a load reserve (`lwarx`) instruction followed by a store conditional (`stwcx`) instruction, wrapped inside an "update loop" because the store-conditional may fail to perform the update. Fence commands are compiled to a single Power `sync` fence instruction. Figure 14 depicts some examples.

The Power model. To make our presentation self-contained, we briefly recall the definition of Alglave et al. [4]. The reader is referred to this paper for further explanations and details. A *Power execution* is taken to be a tuple $G = \langle A, po, deps, rf, At\rangle$ where:

- $A \subseteq \mathcal{A}_p$, where $\mathcal{A}_p$ is the set of *Power events*. $\mathcal{A}_p$ includes read and write events ($\mathcal{R}$ and $\mathcal{W}$, see §2), but no updates ($G.\mathtt{U} = \emptyset$). In addition, it includes events for the different Power fence instructions (whose types are `sync`, `lwsync`, or `isync`). We denote by $\mathcal{S}$ the set of all events with type `sync`.
- *po* and *rf* are program order and reads from edges (as before).
- $deps \subseteq po$ denotes the set of data, address and control *dependency edges* between instructions that Power implementations are guaranteed to preserve. In particular, the compilation of the acquire read induces a (control) dependency edge between the read and the `isync` events by introducing a "fake" compare-and-branch sequence.
- $At \subseteq A$ records the set of *atomic events* in G (i.e., the loads and stores belonging to an update).

A Power execution G induces a number of relations, defined in Fig. 13: instruction fence order ($G.isync$), lightweight fence order ($G.lwsync$), strong fence order ($G.sync$), preserved program order ($G.ppo$), fence order ($G.fence$), external reads-from ($G.rfe$), happens-before ($G.hb$), basic propagation ($G.base$), program order between atomic events ($G.po\text{-}aa$), and read-modify-write ($G.rmw$). For all these notations, we omit the "$G.$" prefix when it is clear from the context.

Definition 19. A Power execution $G = \langle A, po, deps, rf, At\rangle$ is called *Power-coherent* if *hb* is acyclic (*No-thin-air*) and there exist relations $\{co_x\}_{x\in\mathsf{Loc}}$, such that each co_x is a total strict order on $G.\mathtt{W}_x$ and the following hold:

- *SC-per-loc*: $po|_x \cup rf \cup fr \cup co$ is acyclic for every $x \in \mathsf{Loc}$, where $po|_x = \{\langle a,b\rangle \in po \mid loc(a) = loc(b) = x\}$.
- *Atomicity*: $rmw \cap (fre; coe) = \emptyset$.
- *Observation*: $fre; prop; hb^*$ is irreflexive.
- *Propagation*: $co \cup po\text{-}aa \cup prop$ is acyclic.

where $co = \bigcup_x co_x$, $coe = co \setminus po$, $fr = rf^{-1}; co$, $fre = fr \setminus po$, $prop = (base \cap (\mathcal{W} \times \mathcal{W})) \cup (chapo^?; base^*; sync; hb^*)$, and $chapo = coe \cup fre \cup rfe \cup (coe; rfe) \cup (fre; rfe)$.

The relation *co*, called *coherence order*, is used instead of the modification order in the SRA model (see Prop. 7). The *SC-per-loc* property is similar to the x-modification order definition (see Prop. 2), but weaker as it rules out cycles with accesses of only one location. Nevertheless, it suffices to rule out the weak behavior of the (CoRR2) example. The *Atomicity* property basically corresponds to the third case of Fig. 6. The *Observation* property rules out the weak behavior of the MP example, making use of the relation *prop*, called *propagation order*. Finally, *Propagation* rules out the weak behavior of the (2+2W) example.

Alternative semantics for `sync` fences. We first prove a general property of the Power model: its `sync` fences are equivalent to release-acquire RMWs to a distinguished location. This entails that `sync` instructions (and events) are redundant in the Power model, and justifies our treatment of fences as syntactic sugar for the atomic assignment $\langle f := 0\rangle$. To prove this property, we define a correspondence between Power executions with `sync` events and those obtained from them by replacing these events with release-acquire updates. Given plain Power executions $G_s = \langle A_s, po_s, deps_s, \emptyset, At_s\rangle$ and $G_r = \langle A_r, po_r, deps_r, \emptyset, At_r\rangle$, we write $G_s \approx G_r$ if no event in G_s accesses location f and G_r is obtained from G_s by (1) adding an initialization event with label $\langle \mathtt{W}, f, 0\rangle$ and (2) splitting each `sync` event of G_s into a sequence of four po_r-consecutive events with labels $\langle\texttt{lwsync}\rangle$, $\langle \mathtt{R}, f, 0\rangle$, $\langle \mathtt{W}, f, 0\rangle$ and $\langle\texttt{isync}\rangle$, (3) adding a $deps_r$ edge between each $\langle \mathtt{R}, f, 0\rangle$ event and its corresponding $\langle \mathtt{W}, f, 0\rangle$ and `isync` events, and (4) including the introduced read and write events in At_r.

Theorem 7. Let G_s, G_r be plain Power executions, such that $G_s \approx G_r$. Then, G_s is Power-consistent iff G_r is Power-consistent.

Proof sketch. We write $a_s \approx a_r$ for corresponding events a_s in G_s and a_r in G_r. Let rf_s, such that $G'_s = \langle A_s, po_s, deps_s, rf_s, At_s\rangle$ is complete and Power-coherent, and let $\{co_x\}_{x \in \mathsf{Loc}\setminus\{f\}}$ be the coherence orders for G'_s that satisfy the conditions of Def. 19. We construct rf_r and co_f, such that $G'_r = \langle A_r, po_r, deps_r, rf_r, At_r\rangle$ is complete and $\{co_x\}_{x\in\mathsf{Loc}}$ satisfy the conditions of Def. 19 for G'_r. We note that *Propagation* entails that the relation

$$\begin{pmatrix} po_s; hb^*; (co \cup po\text{-}aa \cup (base \cap (\mathcal{W}\times\mathcal{W})))^*; \\ chapo^?; base^*; po_s \end{pmatrix} \cap (\mathcal{S}\times\mathcal{S})$$

is acyclic (using the notations of Fig. 13 and Def. 19 for G_s). Take T to be a strict total order on $G_s.\mathtt{sync} \times G_s.\mathtt{sync}$ extending this relation. We set $co_f = \{\langle a_r, b_r\rangle \in G_r.\mathtt{W}_f \times G_r.\mathtt{W}_f \mid \exists\langle a_s, b_s\rangle \in T.\ a_s \approx a_r \wedge b_s \approx b_r\}$ and $rf_r = rf_s \cup (ico_f; rmw^{-1})$, where ico_f consists of the immediate co_f-edges, and show that the conditions of Def. 19 are satisfied.

For the converse, we pick $rf_s = \{\langle a, b\rangle \in rf_r \mid loc(a) \neq f\}$ and show that the conditions of Def. 19 are satisfied for the relations $\{co_x\}_{x\in\mathsf{Loc}\setminus\{f\}}$. □

An interesting consequence of this theorem is that a Power program with just one `sync` fence is equivalent to the program where that `sync` fence is replaced with an `lwsync` fence. Indeed, the effect of a release-acquire RMW to an otherwise unused location is equivalent to the effect of a single `lwsync` fence.

Compilation soundness and completeness. To relate the SRA and Power models, we first define a correspondence between executions defined in §2 (that we refer to as *SRA executions*) and Power executions. Here, we consider only Power executions that result from the standard compilation scheme of C11 release-acquire accesses to Power (see Fig. 14). Further, observing that a program may terminate under Power iff it has a Power-consistent execution in which all update loops succeed immediately, we assume that update commands generate just one read event.

Definition 20. Given an SRA execution $G = \langle A, po, rf\rangle$ and a Power execution $G_p = \langle A_p, po_p, deps, rf_p, At\rangle$ with $G_p.\mathtt{sync} = \emptyset$, we write $G \sim G_p$ if there exist bijections:

$$w : G.\mathtt{W}\cup G.\mathtt{U} \to G_p.\mathtt{W} \quad f_w : G.\mathtt{W}\cup G.\mathtt{U} \to G_p.\mathtt{lwsync}$$
$$r : G.\mathtt{R}\cup G.\mathtt{U} \to G_p.\mathtt{R} \quad f_r : G.\mathtt{R}\cup G.\mathtt{U} \to G_p.\mathtt{isync}$$

such that the following hold, where $g : A_p \to A$ denotes the function $w^{-1} \cup f_w^{-1} \cup r^{-1} \cup f_r^{-1}$:

- $loc \circ w = loc, val_w \circ w = val_w, loc \circ r = loc, val_r \circ r = val_r$
- $po_p = \begin{pmatrix} \{\langle a,b\rangle \in A_p\times A_p \mid \langle g(a), g(b)\rangle \in po\} \\ \cup\{\langle f_w(a), w(a)\rangle \mid a \in G.\mathtt{W}\} \\ \cup\{\langle r(a), f_r(a)\rangle \mid a \in G.\mathtt{R}\} \\ \cup\{\langle f_w(a), r(a)\rangle \mid a \in G.\mathtt{U}\} \\ \cup\{\langle r(a), w(a)\rangle \mid a \in G.\mathtt{U}\} \\ \cup\{\langle w(a), f_r(a)\rangle \mid a \in G.\mathtt{U}\} \end{pmatrix}^+$
- $deps \supseteq \{\langle r(a), w(a)\rangle \mid a \in G.\mathtt{U}\}$
- $deps \supseteq \{\langle r(a), f_r(a)\rangle \mid a \in G.\mathtt{U}\cup G.\mathtt{R}\}$
- $rf_p = \{\langle w(a), r(b)\rangle \mid \langle a,b\rangle \in rf\}$
- $At = \{r(a) \mid a \in G.\mathtt{U}\} \cup \{w(a) \mid a \in G.\mathtt{U}\}$

Informally, one can easily see that for every SRA execution G of a program P, there exists a Power execution G_p of the compilation of P such that $G \sim G_p$, and vice versa. For Power executions that relate to SRA ones, we can simplify the Power model as follows:

Proposition 9. Let $G_p = \langle A, po, deps, rf, At\rangle$ be a Power execution that is $\sim$-related to some SRA execution. Then:

- $ppo = po \cap (\mathcal{R} \times (\mathcal{R}\cup\mathcal{W}))$.
- $fence = \begin{pmatrix} ((po \setminus rmw) \cap ((\mathcal{R}\cup\mathcal{W})\times\mathcal{W})); \\ (po \cap (\mathcal{W}\times(\mathcal{R}\cup\mathcal{W})))^? \end{pmatrix}$.

Next, we prove that for related executions, SRA-coherence coincides with Power-coherence.

Theorem 8. Let $G \sim G_p$ be related SRA and Power executions. Then, G is SRA-coherent iff G_p is Power-coherent.

Proof sketch. For one direction, we show that any total order extending $\{\langle w^{-1}(a_p), w^{-1}(b_p)\rangle \mid \langle a_p, b_p\rangle \in (co \cup prop)^+\}$ is a modification order in G. For the converse, we satisfy the conditions of Def. 19 by taking $co_x = \{\langle w(a), w(b)\rangle \mid \langle a,b\rangle \in mo_x\}$. □

The exact correspondence between SRA and Power entails that SRA cannot be further strengthened without modifying the existing compilation scheme. Note that for RA only the right-to-left implication holds because SRA is strictly stronger than RA.

8. Related Work

Steinke and Nutt [34] developed a hierarchy of memory models expressed in terms of per-thread *view order*, that uniformly accounts for several classical consistency models (such as causal consistency [3] and PRAM consistency [21]). Except for SC, all these classical models are strictly weaker than RA. Roughly speaking, the reason is that a per-thread view induces a per-thread modification order, while RA assumes a global modification order for each location.

There are some other recent operational memory models. Many of these (e.g., [4, 8, 36]) are essentially wrappers around a declarative model: they build an execution graph in a stepwise fashion and check for consistency as new events are added to the execution. In contrast, our operational semantics for SRA does not refer to graphs and consistency axioms. An operational memory model for Power was defined by Sarkar et al. [28], and it is stronger than the model of [4]. Their operational model attempts to cover the entire Power architecture and is thus substantially more complex than ours. In addition, Zhang and Feng [40] present an operational weak memory model for Java that models weak behaviors by allowing event replay. The resulting model is much weaker than RA.

Wickerson et al. [39] proposed a new simpler semantics for SC accesses, including SC fences. While their semantics of SC fences is stronger than the C11's semantics, it is weaker than ours, and still allows weak behaviors for RA programs even when fences are placed between every two commands.

Weak consistency models appear also in the context of distributed systems, and more specifically, for replicated databases. These models include an additional construct of a *transaction*, which can be seen as an atomic block of instructions. Recently, Cerone et al. [11] proposed a framework for specifying such models, and studied several important ones in this framework. Among those, the closest to SRA is PSI ("Parallel Snapshot Isolation"). Stripping compound transactions out, and adapting to our terminology, PSI-coherence can be seen as a strengthening of SRA with the condition asserting that $po \cup rf \cup \bigcup_y mo_y \cup fr_x$ is acyclic for every $x \in \mathsf{Loc}$ (where, as above, $fr_x = (rf^{-1}; mo_x) \setminus Id_A$). The next example shows that PSI is strictly stronger than SRA:

$$\begin{array}{r||l} x := 1 & y := 2 \\ x := 2 & \mathtt{wait}\ (x = 1) \\ y := 1 & \mathtt{wait}\ (z = 1) \\ z := 1 & \mathtt{wait}\ (y = 2) \end{array}$$

This program may terminate under SRA (and, hence, also under Power), but not under PSI. In addition, it may terminate under TSO. Hence, we believe that PSI is too strong as a high-performance memory model for a programming language. Note, however, that PSI coincides with SRA for WW-race free executions.

Regarding the verification of RCU, Tassarotti et al. [35] and Lahav and Vafeiadis [18] use program logics (called GPS and

OGRA, respectively) to verify an RCU implementation under RA. However, their RCU implementation is based on the simple one shown in Fig. 12, while the more advanced implementation of Fig. 10 was not considered.

9. Conclusion

We have presented a new memory model, SRA, a slight strengthening of RA with many nice properties: (1) it is equivalent to RA in the absence of write-write races, (2) it has the same implementation cost on TSO and Power, (3) it allows the same local program transformations, and (4) it has an intuitive operational characterization. Moreover, we provided a simple notion of a strong memory fence that suffices for restoring sequential consistency assuming all (racy) accesses are RA, and presented reduction theorems to SC and TSO for particular classes of programs.

Practically speaking, the results of this paper suggest two independent strengthenings of the C11 memory model that do not incur any performance penalty: (1) adopting SRA instead of RA as the semantics of release-acquire accesses, and (2) modeling SC fences as release-acquire RMWs to a special location. Of course, a prerequisite for employing (1) in C11 is the investigation of our results in the broader context of the full C11 memory model, which includes more access kinds: non-atomic, SC atomic, relaxed atomic and consume atomic accesses. The most important of these are the non-atomic accesses because they avoid the synchronization overhead on Power, ARM, and Itanium. For them, the extension of our results is mostly straightforward (as sketched in the supplementary materiel). In fact, we believe that a programming language that has only release/acquire accesses (with SRA semantics) alongside non-atomic accesses (that should not be racy) should provide a reasonable balance between performance and programmability.

In addition, the simple operational model for SRA may pave the way for better understanding of programs running under weak memory, and development of essential verification methods and tools for these programs (e.g., model checking and program logics). Another theoretically interesting question is whether basic RA admits a similar machine semantics.

Acknowledgments

We would like to thank Marko Doko, Jeehoon Kang, Peter Sewell, Francesco Zappa Nardelli, and the POPL'16 reviewers for their helpful feedback. This work was supported by EC FET project ADVENT (308830).

References

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.

[2] P. A. Abdulla, M. F. Atig, and N. T. Phong. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP 2015: 24th European Symposium on Programming*, volume 9032 of *LNCS*, pages 308–332. Springer, 2015.

[3] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[4] J. Alglave, L. Maranget, and M. 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.

[5] J. Barnat, L. Brim, and V. Havel. LTL model checking of parallel programs with under-approximated TSO memory model. In *13th International Conference on Application of Concurrency to System Design, ACSD'13*, pages 51–59, July 2013.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66. ACM, 2011.

[7] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 509–520. ACM, 2012.

[8] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *24th European Symposium on Programming (ESOP 2015)*, volume 9032 of *LNCS*, pages 283–307. Springer, 2015.

[9] A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In *Automata, Languages and Programming*, volume 6756 of *LNCS*, pages 428–440. Springer, 2011.

[10] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *Programming Languages and Systems*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

[11] A. Cerone, G. Bernardi, and A. Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *LIPIcs*, pages 58–71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[12] A. Dan, Y. Meshman, M. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In *Verification, Model Checking, and Abstract Interpretation*, volume 8931 of *LNCS*, pages 449–466. Springer, 2015.

[13] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, 2012. .

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

[15] ISO/IEC 9899:2011. Programming language C, 2011.

[16] S. Jagannathan, V. Laporte, G. Petri, D. Pichardie, and J. Vitek. Atomicity refinement for verified compilation. *ACM Trans. Program. Lang. Syst.*, 36(2):6:1–6:30, 2014.

[17] M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 187–198. ACM, 2011.

[18] O. Lahav and V. Vafeiadis. Owicki-gries reasoning for weak memory models. In *Automata, Languages, and Programming, ICALP'15*, volume 9135 of *LNCS*, pages 311–323. Springer, 2015.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[20] A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *Model Checking Software*, volume 6349 of *LNCS*, pages 212–226. Springer, 2010.

[21] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical report, Technical Report CS-TR-180-88, Princeton University, 1988.

[22] W. Mansky and E. L. Gunter. Verifying optimizations for concurrent programs. In *First International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE 2014*, volume 40 of *OASICS*, pages 15–26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[23] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012.

[24] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP 2010: 24th European Conference on Object-Oriented Programming*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.

[25] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009.

[26] B. Rajaram, V. Nagarajan, S. Sarkar, and M. Elver. Fast RMWs for TSO: Semantics and implementation. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 61–72. ACM, 2013.

[27] T. Ridge. A rely-guarantee proof system for x86-TSO. In *VSTTE 2010*, volume 6217 of *LNCS*, pages 55–70. Springer, 2010.

[28] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186. ACM, 2011.

[29] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 311–322. ACM, 2012.

[30] J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[31] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10 (2):282–312, 1988.

[32] F. Sieczkowski, K. Svendsen, L. Birkedal, and J. Pichon-Pharabod. A separation logic for fictional sequential consistency. In *ESOP 2015*, volume 9032 of *LNCS*, pages 736–761. Springer, 2015.

[33] SPARC International Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., 1992. ISBN 0-13-825001-4.

[34] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, Sept. 2004.

[35] J. Tassarotti, D. Dreyer, and V. Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 110–120. ACM, 2015.

[36] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 691–707. ACM, 2014. .

[37] V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *18th International Conference on Static Analysis, SAS'11*, volume 6887 of *LNCS*, pages 146–162. Springer, 2011.

[38] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 209–220. ACM, 2015.

[39] J. Wickerson, M. Batty, and A. F. Donaldson. Overhauling SC atomics in C11 and OpenCL. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, 2016.

[40] Y. Zhang and X. Feng. An operational approach to happens-before memory model. In *7th International Symposium on Theoretical Aspects of Software Engineering, TASE 2013*, pages 121–128. IEEE Computer Society, 2013.