Repairing Sequential Consistency in C/C++11

Ori Lahav
MPI-SWS, Germany [*]
orilahav@mpi-sws.org

Viktor Vafeiadis
MPI-SWS, Germany [*]
viktor@mpi-sws.org

Jeehoon Kang
Seoul National University, Korea
jeehoon.kang@sf.snu.ac.kr

Chung-Kil Hur
Seoul National University, Korea
gil.hur@sf.snu.ac.kr

Derek Dreyer
MPI-SWS, Germany [*]
dreyer@mpi-sws.org

Abstract

The C/C++11 memory model defines the semantics of concurrent memory accesses in C/C++, and in particular supports racy "atomic" accesses at a range of different consistency levels, from very weak consistency ("relaxed") to strong, sequential consistency ("SC"). Unfortunately, as we observe in this paper, the semantics of SC atomic accesses in C/C++11, as well as in all proposed strengthenings of the semantics, is flawed, in that (contrary to previously published results) both suggested compilation schemes to the Power architecture are unsound. We propose a model, called RC11 (for Repaired C11), with a better semantics for SC accesses that restores the soundness of the compilation schemes to Power, maintains the DRF-SC guarantee, and provides stronger, more useful, guarantees to SC fences. In addition, we formally prove, for the first time, the correctness of the proposed stronger compilation schemes to Power that preserve load-to-store ordering and avoid "out-of-thin-air" reads.

CCS Concepts • **Theory of computation** → **Program semantics**; • **Software and its engineering** → **Concurrent programming languages**

Keywords Weak memory models; C++11; declarative semantics; sequential consistency

1. Introduction

The C/C++11 memory model (C11 for short) [8] defines the semantics of concurrent memory accesses in C/C++, of which there are two general types: *non-atomic* and *atomic*. Non-atomic accesses are intended for normal data: races on such accesses are considered as programming errors and lead to undefined behavior, thus ensuring that they can be compiled to plain machine loads and stores and that it is sound to apply standard sequential optimizations on non-atomic accesses. In contrast, atomic accesses are specifically intended for communication between threads: thus, races on atomics are permitted, but at the cost of introducing hardware fence instructions during compilation and imposing restrictions on how such accesses may be merged or reordered.

The degree to which an atomic access may be reordered with other operations—and more generally, the implementation cost of an atomic access—depends on its *consistency* level, concerning which C11 offers programmers several options according to their needs. Strongest and most expensive are *sequentially consistent* (SC) accesses, whose primary purpose is to restore the simple interleaving semantics of sequential consistency [20] if a program (when executed under SC semantics) only has races on SC accesses. This property is called "DRF-SC" and was a main design goal for C11. To ensure DRF-SC, the standard compilation schemes for modern architectures typically insert hardware "fence" instructions appropriately into the compiled code, with those for weaker architectures (like Power and ARMv7) introducing a full (strong) fence adjacent to each SC access.

Weaker than SC atomics are *release-acquire* accesses, which can be used to perform "message passing" between threads without incurring the implementation cost of a full SC access; and weaker and cheaper still are *relaxed* accesses, which are intended to be compiled down to plain loads and stores at the machine level and which provide only the minimal synchronization guaranteed by the hardware. Finally, the C11 model also supports language-level *fence* instructions, which provide finer-grained control over where hardware fences are to be placed and serve as a barrier to prevent unwanted compiler optimizations.

[*] Saarland Informatics Campus.

In this paper, we are mainly concerned with the semantics of SC atomics (*i.e.,* SC accesses and SC fences), and their interplay with the rest of the model. Since sequential consistency is such a classical, well-understood notion, one might expect that the semantics of SC atomics should be totally straightforward, but sadly, as we shall see, it is not!

The main problem arises in programs that mix SC and non-SC accesses to the same location. Although not common, such mixing is freely permitted by the C11 standard, and has legitimate uses—*e.g.,* as a way of enabling faster (non-SC) reads from an otherwise quite strongly synchronized data structure. Indeed, we know of several examples of code in the wild that mixes SC accesses together with release/acquire or relaxed accesses to the same location: `seqlocks` [9] and Rust's `crossbeam` library [2]. Now, consider the following program due to Manerkar *et al.* [22]:

$$x :=_{\mathtt{sc}} 1 \;\left\|\; \begin{array}{l} a := x_{\mathtt{acq}} \;/\!\!/1 \\ c := y_{\mathtt{sc}} \;/\!\!/0 \end{array} \;\right\|\; \begin{array}{l} b := y_{\mathtt{acq}} \;/\!\!/1 \\ d := x_{\mathtt{sc}} \;/\!\!/0 \end{array} \;\left\|\; y :=_{\mathtt{sc}} 1 \right. \qquad \text{(IRIW-acq-sc)}$$

Here and in all other programs in this paper, we write $a, b, ...$ for local variables (registers), and assume that all variables are initialized to 0. The program contains two variables, x and y, which are accessed via SC atomic accesses and also read by acquire atomic accesses. The annotated behavior (reading $a = b = 1$ and $c = d = 0$) corresponds to the two threads observing the writes to x and y as occurring in different orders, and is forbidden by C11. (We defer the explanation of how C11 forbids this behavior to §2.)

Let's now consider how this program is compiled to Power. Two compilation schemes have been proposed [7]. Both use Power's strongest fence instruction, called `sync`, for the compilation of SC atomics. The first scheme, the one implemented in the `GCC` and `LLVM` compilers, inserts a `sync` fence *before* each SC access ("leading sync" convention), whereas the alternative scheme inserts a `sync` fence *after* each SC access ("trailing sync" convention). The intent of both schemes is to have a strong barrier between every pair of SC accesses, enforcing, in particular, sequential consistency on programs containing only SC accesses. Nevertheless, by mixing SC and release-acquire accesses, one can quickly get into trouble, as illustrated by IRIW-acq-sc.

In particular, if one compiles the program into Power using the trailing sync convention, then the behavior is allowed by Power.[1] Since all SC accesses are at the end of the threads, the trailing `sync` fences have no effect, and the example reduces to (the result of compilation of) IRIW with only acquire reads, which is allowed by the Power memory model. In §2.1, we show further examples illustrating that the other, leading sync scheme also leads to behaviors in the target of compilation to Power that are not permitted in the source.

Although the C11 model is known to have multiple problems (*e.g.,* the "out-of-thin-air" problem [31, 11], the lack of monotonicity [30]), none of them until now affected the correctness of compilation to the mainstream architectures. In contrast, the IRIW-acq-sc program from [22] and our examples in §2.1 show that both the suggested compilation schemes to Power are unsound with respect to the C11 model, thereby contradicting the results of [7, 27]. The same problem occurs in some compilation schemes to ARMv7 (see §6), as well as for ARMv8 (see [3] for an example).

In the remainder of the paper, we propose a way to repair the semantics of SC accesses that resolves the problems mentioned above. In particular, our corrected semantics restores the soundness of the suggested compilation schemes to Power. Moreover, it still satisfies the standard DRF-SC theorem in the absence of relaxed accesses: if a program's sequentially consistent executions only ever exhibit races on SC atomic accesses, then its semantics under full C11 is also sequentially consistent. It is worth noting that this correction only affects the semantics of programs mixing SC and non-SC accesses to the same location: we show that, without such mixing, it coincides with the strengthened model of Batty *et al.* [5].

We also apply two additional, orthogonal, corrections to the C11 model, which strengthen the semantics of SC fences. The first fix corrects a problem already noted before [27, 21, 17], namely that the current semantics of SC fences does not recover sequential consistency, even when SC fences are placed between every two commands in programs with only release/acquire atomic accesses. The second fix provides stronger "cumulativity" guarantees for programs with SC fences. We justify these strengthenings by proving that the existing compilation schemes for x86-TSO, Power, and ARMv7 remain sound with the stronger semantics.

Finally, we apply another, mostly orthogonal, correction to the C11 model, in order to address the well-known "out-of-thin-air" problem. The problem is that the C11 standard permits certain executions as a result of causality cycles, which break even basic invariant-based reasoning [11] and invalidate DRF-SC in the presence of relaxed accesses. The correction, which is simple to state formally, is to strengthen the model to enforce load-to-store ordering for atomic accesses, thereby ruling out such causality cycles, at the expense of requiring a less efficient compilation scheme for relaxed accesses. The idea of this correction is not novel—it has been extensively discussed in the literature [31, 11, 30]—but the suggested compilation schemes to Power and ARMv7 have not yet been proven sound. Here, we give the first proof that one of these compilation schemes—the one that places a fake control dependency after every relaxed read—is sound. The proof is surprisingly delicate, and involves a novel argument similar to that in DRF-SC proofs.

Putting all these corrections together, we propose a new model called RC11 (for Repaired C11) that supports nearly all features of the C11 model (§3). We prove correctness of compilation to x86-TSO (§4), Power (§5), and ARMv7 (§6),

[1] Formally, we use the recent declarative model of Power by Alglave *et al.* [4].

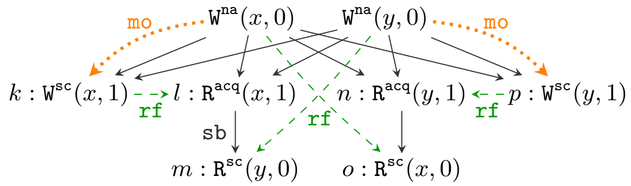

mo
$\mathtt{W^{na}}(x,0)$
$\mathtt{W^{na}}(y,0)$
mo
$k : \mathtt{W^{sc}}(x,1)$
rf
$l : \mathtt{R^{acq}}(x,1)$
$n : \mathtt{R^{acq}}(y,1)$
rf
$p : \mathtt{W^{sc}}(y,1)$
sb
rf
$m : \mathtt{R^{sc}}(y,0)$
$o : \mathtt{R^{sc}}(x,0)$

Figure 1. An execution of IRIW-acq-sc yielding the result $a = b = 1 \wedge c = d = 0$.

the soundness of a wide collection of program transformations (§7), and a DRF-SC theorem (§8).

2. The Semantics of SC Atomics in C11: What's Wrong, and How Can We Fix It?

The C11 memory model defines the semantics of a program as a set of consistent executions. Each execution is a graph. Its nodes, $\mathtt{E}$, are called *events* and represent the individual memory accesses and fences of the program, while its edges represent various relations among these events:

- The *sequenced-before* ($\mathtt{sb}$) relation, a.k.a. *program order*, captures the order of events in the program's control flow.
- The *reads-from* ($\mathtt{rf}$) relation associates each write with the set of reads that read from that write. In a consistent execution, the reads-from relation should be functional (and total) in the second argument: a read must read from exactly one write.
- Finally, the *modification order* ($\mathtt{mo}$) is a union of total orders, one for each memory address, totally ordering the writes to that address. Intuitively, it records for each memory address the globally agreed-upon order in which writes to that address happened.

As an example, in Fig. 1, we depict an execution of the IRIW-acq-sc program discussed in the introduction. In addition to the events corresponding to the accesses appearing in the program, the execution contains two events for the implicit non-atomic initialization writes to x and y, which are assumed to be $\mathtt{sb}$-before all other events.

Notation 1. Given a binary relation R, we write $R^?$, R^+, and R^* respectively to denote its reflexive, transitive, and reflexive-transitive closures. The inverse relation is denoted by R^{-1}. We denote by $R_1; R_2$ the left composition of two relations R_1, R_2, and assume that ; binds tighter than $\cup$ and $\setminus$. Finally, we denote by $[A]$ the identity relation on a set A. In particular, $[A]; R; [B] = R \cap (A \times B)$.

Based on these three basic relations, C11 defines some derived relations. First, whenever an acquire or SC read reads from a release or SC write, we say that the write *synchronizes with* ($\mathtt{sw}$) the read.[2] Next, we say that one event *happens before* ($\mathtt{hb}$) another event if they are connected by a sequence of $\mathtt{sb}$ or $\mathtt{sw}$ edges. Formally, $\mathtt{hb} \triangleq (\mathtt{sb} \cup \mathtt{sw})^+$. For example, in Fig. 1, event k synchronizes with l and therefore k happens-before l and m. Lastly, whenever a read event e reads from a write that is $\mathtt{mo}$-before another write f to the same location, we say that e *reads-before* ($\mathtt{rb}$) f (this relation is also called *"from-read"* [4], but we find *reads-before* more intuitive). Formally, $\mathtt{rb} \triangleq \mathtt{rf}^{-1}; \mathtt{mo} \setminus [\mathtt{E}]$. The "$\setminus [\mathtt{E}]$" part is needed so that RMW events ("read-modify-write", induced by atomic update operations like fetch-and-add and compare-and-swap) do not read-before themselves. For example, in Fig. 1, we have $\langle m, p \rangle \in \mathtt{rb}$ and $\langle o, k \rangle \in \mathtt{rb}$.

Consistent C11 executions require that $\mathtt{hb}$ is irreflexive (equivalently, $\mathtt{sb} \cup \mathtt{sw}$ is acyclic), and further guarantee *coherence* (aka *SC-per-location*) and *atomicity* of RMWs. Roughly speaking, coherence ensures that (i) the order of writes to the same location according to $\mathtt{mo}$ does not contradict $\mathtt{hb}$ (COHERENCE-WW); (ii) reads do not read values written in the future (NO-FUTURE-READ and COHERENCE-RW); (iii) reads do not read overwritten values (COHERENCE-WR); and (iv) two $\mathtt{hb}$-related reads from the same location cannot read from two writes in reversed $\mathtt{mo}$-order (COHERENCE-RR). We refer the reader to Prop. 1 in §3 for a formal definition of coherence.

Now, to give semantics to SC atomics, C11 stipulates that in consistent executions, there should be a strict total order, $\mathtt{S}$, over all SC events, intuitively corresponding to the order in which these events are executed. This order is required to satisfy a number of conditions (but see Remark 1 below), where $\mathtt{E^{sc}}$ denotes the set of all SC events in $\mathtt{E}$:

(S1) $\mathtt{S}$ must include $\mathtt{hb}$ restricted to SC events (formally: $[\mathtt{E^{sc}}]; \mathtt{hb}; [\mathtt{E^{sc}}] \subseteq \mathtt{S}$);

(S2) $\mathtt{S}$ must include $\mathtt{mo}$ restricted to SC events (formally: $[\mathtt{E^{sc}}]; \mathtt{mo}; [\mathtt{E^{sc}}] \subseteq \mathtt{S}$);

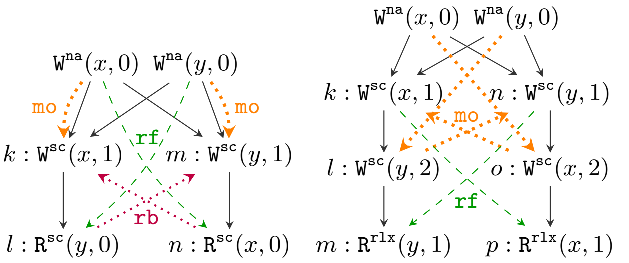
(S3) $\mathtt{S}$ must include $\mathtt{rb}$ restricted to SC events (formally: $[\mathtt{E^{sc}}]; \mathtt{rb}; [\mathtt{E^{sc}}] \subseteq \mathtt{S}$);

(S4-7) $\mathtt{S}$ must obey a few more conditions having to do with SC fences.

Remark 1. The S3 condition above, due to Batty *et al.* [5], is slightly simpler and stronger than the one imposed by the official C11. Crucially, however, **all the problems and counterexamples we observe in this section, concerning the C11 semantics of SC atomics, hold for both Batty *et al.*'s model and the original C11**. The reason we use Batty *et al.*'s version here is that it provides a cleaner starting point for our discussion, and our solution to the problems with C11's SC semantics will build on it.

Intuitively, the effect of the above conditions is to enforce that, since $\mathtt{S}$ corresponds to the order in which SC events are executed, it should agree with the other global orders of events: $\mathtt{hb}$, $\mathtt{mo}$, and $\mathtt{rb}$. However, as we will see shortly, condition S1 is too strong. Before we get there, let us first

[2] The actual definition of $\mathtt{sw}$ contains further cases, which are not relevant for the current discussion. These are included in our formal model in §3.



$\mathtt{W}^{\mathtt{na}}(x,0)$ $\mathtt{W}^{\mathtt{na}}(y,0)$
mo
mo
rf
$k : \mathtt{W}^{\mathtt{sc}}(x,1)$
$m : \mathtt{W}^{\mathtt{sc}}(y,1)$
rb
$l : \mathtt{R}^{\mathtt{sc}}(y,0)$
$n : \mathtt{R}^{\mathtt{sc}}(x,0)$
$\mathtt{W}^{\mathtt{na}}(x,0)$ $\mathtt{W}^{\mathtt{na}}(y,0)$
$k : \mathtt{W}^{\mathtt{sc}}(x,1)$
$n : \mathtt{W}^{\mathtt{sc}}(y,1)$
mo
$l : \mathtt{W}^{\mathtt{sc}}(y,2)$
$o : \mathtt{W}^{\mathtt{sc}}(x,2)$
rf
$m : \mathtt{R}^{\mathtt{rlx}}(y,1)$
$p : \mathtt{R}^{\mathtt{rlx}}(x,1)$

Figure 2. Inconsistent C11 executions of SB and 2+2W.

look at a few examples to illustrate how the conditions on $\mathtt{S}$ interact to enforce sequential consistency.

Consider the classic "store buffering" litmus test:

$$\begin{array}{l||l} x :=_{\mathtt{sc}} 1 & y :=_{\mathtt{sc}} 1 \\ a := y_{\mathtt{sc}} \ /\!/0 & b := x_{\mathtt{sc}} \ /\!/0 \end{array} \tag{SB}$$

Here, the annotated behavior is forbidden by C11. To see this, consider the first execution graph in Fig. 2. The $\mathtt{rf}$ edges are forced because of the values read, while the $\mathtt{mo}$ edges are forced because of COHERENCE-WW. Then, $\mathtt{S}(k,l)$ and $\mathtt{S}(m,n)$ hold because of condition S1; while $\mathtt{S}(l,m)$ and $\mathtt{S}(n,k)$ hold because of condition S3. This entails a cycle in $\mathtt{S}$, which is disallowed.

Similarly, C11's conditions guarantee that the following (variant given in [32] of the) 2+2W litmus test disallows the annotated weak behavior:

$$\begin{array}{l||l} x :=_{\mathtt{sc}} 1 & y :=_{\mathtt{sc}} 1 \\ y :=_{\mathtt{sc}} 2 & x :=_{\mathtt{sc}} 2 \\ a := y_{\mathtt{rlx}} \ /\!/1 & b := x_{\mathtt{rlx}} \ /\!/1 \end{array} \tag{2+2W}$$

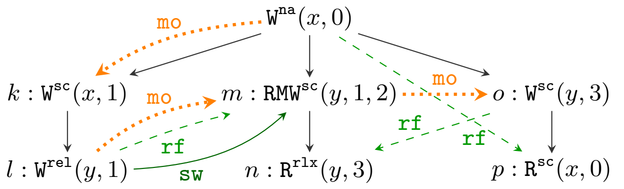
To see this, consider the second execution graph in Fig. 2, which has the outcome $a = b = 1$: the $\mathtt{rf}$ and $\mathtt{mo}$ edges are forced because of the values read and COHERENCE-WR. Now, $\mathtt{S}(k,l)$ and $\mathtt{S}(n,o)$ hold because of condition S1; while $\mathtt{S}(l,n)$ and $\mathtt{S}(o,k)$ hold because of condition S2. Again, this entails a cycle in $\mathtt{S}$.

Let us now move to the IRIW-acq-sc program from the introduction, whose annotated behavior is also forbidden by C11. To see that, suppose without loss of generality that $\mathtt{S}(p,k)$ in Fig. 1. We also know that $\mathtt{S}(k,m)$ because of happens-before via l (S1). Thus, by transitivity, $\mathtt{S}(p,m)$. However, if the second thread reads $y = 0$, then m reads-before p, in which case $\mathtt{S}(m,p)$ (S3), and $\mathtt{S}$ has a cycle.

2.1 First Problem: Compilation to Power is Broken

The IRIW-acq-sc example demonstrates that the trailing sync compilation to Power is unsound for the C11 model. We will now see an example showing that the leading sync compilation is also unsound. Consider the following behavior, where all variables are zero-initialized and FAI(y) represents an atomic fetch-and-increment of y returning its value before the increment:

$$\begin{array}{l||l||l} x :=_{\mathtt{sc}} 1 & b := \mathrm{FAI}(y)_{\mathtt{sc}} \ /\!/1 & y :=_{\mathtt{sc}} 3 \\ y :=_{\mathtt{rel}} 1 & c := y_{\mathtt{rlx}} \ /\!/3 & a := x_{\mathtt{sc}} \ /\!/0 \end{array} \tag{Z6.U}$$



mo
$\mathtt{W}^{\mathtt{na}}(x,0)$
$k : \mathtt{W}^{\mathtt{sc}}(x,1)$
mo
$m : \mathtt{RMW}^{\mathtt{sc}}(y,1,2)$
mo
$o : \mathtt{W}^{\mathtt{sc}}(y,3)$
rf
rf
rf
$l : \mathtt{W}^{\mathtt{rel}}(y,1)$
sw
$n : \mathtt{R}^{\mathtt{rlx}}(y,3)$
$p : \mathtt{R}^{\mathtt{sc}}(x,0)$

Figure 3. A C11 execution of Z6.U. The initialization of y is omitted as it is not relevant.

We will show that the behavior is disallowed according to C11, but allowed by its compilation to Power.

Fig. 3 depicts the only execution yielding the behavior in question that satisfies the coherence constraints. Again, the $\mathtt{rf}$ and $\mathtt{mo}$ edges are forced: even if all accesses in the program were relaxed atomic, they would have to go this way. $\mathtt{S}(k,m)$ holds because of condition S1 (k happens-before l, which happens-before m); $\mathtt{S}(m,o)$ holds because of condition S2 (m precedes o in modification order); $\mathtt{S}(o,p)$ holds because of condition S1 (o happens-before p). Finally, since p reads $x = 0$, we have that p reads-before k, so by S3, $\mathtt{S}(p,k)$, thus forming a cycle in $\mathtt{S}$.

Under the leading sync compilation to Power, however, the behavior is allowed. Intuitively, all but one of the `sync` fences because of the SC accesses are useless because they are at the beginning of a thread. In the absence of other `sync` fences, the only remaining `sync` fence, due to the $a := x_{\mathtt{sc}}$ load in the last thread, is equivalent to an `lwsync` fence (cf. [17, §7]).

In [3] we provide a similar example using SC fences instead of RMW instructions, which shows that even placing `sync` fences both before and after SC accesses is unsound.

What Went Wrong and How to Fix it Generally, in order to provide coherence, hardware memory models provide rather strong ordering guarantees on accesses to the same memory location. Consequently, for conditions S2 and S3, which only enforce orderings between accesses to the same location, ensuring that compilation preserves these conditions is not difficult, even for weaker architectures like Power and ARM.

When, however, it comes to ensuring a strong ordering between accesses of *different* memory locations, as S1 does, compiling to weaker hardware requires the insertion of appropriate memory fence instructions. In particular, for Power, to enforce a strong ordering between two $\mathtt{hb}$-related accesses to different locations, there should be a Power `sync` fence occurring somewhere in the $\mathtt{hb}$-path (the sequence of `sb` and $\mathtt{sw}$ edges) connecting the two accesses. Unfortunately, in the presence of mixed SC and non-SC accesses, the Power compilation schemes do not always ensure that a `sync` exists between $\mathtt{hb}$-related SC accesses. Specifically, if we follow the trailing sync convention, the $\mathtt{hb}$-path (in Fig. 1) from k to m starting with an $\mathtt{sw}$ edge avoids the `sync` fence placed after k. Conversely, if we follow the leading sync convention, the $\mathtt{hb}$-path (in Fig. 3) from k to m ending with an $\mathtt{sw}$ edge avoids the fence placed before m. The result is that S1 enforces more ordering than the hardware provides!

So, if requiring that $\mathtt{hb}$ (on SC events) be included in $\mathtt{S}$ is too strong a condition, what should we require instead? The essential insight is that, according to either compilation scheme, we know that a `sync` fence will necessarily exist between SC accesses a and b if the $\mathtt{hb}$ path from a to b starts and ends with an $\mathtt{sb}$ edge. Second, if a and b access the same location, then the hardware will preserve the ordering anyway. These two observations lead us to replace condition S1 with the following:

(S1fix) $\mathtt{S}$ must relate any two SC events that are related by $\mathtt{hb}$, *provided that* the $\mathtt{hb}$-path between the two events either starts and ends with $\mathtt{sb}$ edges, or starts and ends with accesses to the same location (formally: $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{sb} \cup \mathtt{sb}; \mathtt{hb}; \mathtt{sb} \cup \mathtt{hb}|_{\mathtt{loc}}); [\mathtt{E}^{\mathtt{sc}}] \subseteq \mathtt{S}$, where $\mathtt{hb}|_{\mathtt{loc}}$ denotes $\mathtt{hb}$ edges between accesses to the same location).

We note that condition S1fix, although weaker than S1, suffices to rule out the weak behaviors of the basic litmus tests (*i.e.,* SB and 2+2W). In fact, just to rule out these behaviors, it suffices to require $\mathtt{sb}$ (on SC events) to be included in $\mathtt{S}$.

In essence, according to S1fix, $\mathtt{S}$ must include all the $\mathtt{hb}$-paths between SC accesses to different locations that exist regardless of any synchronization induced by the SC accesses at their endpoints. If a program does not mix SC and non-SC accesses to the same location, then every *minimal* $\mathtt{hb}$-path between two SC accesses to the same location (*i.e.,* one which does not go through another SC access) must start and end with an $\mathtt{sb}$ edge, in which case S1 and S1fix coincide.

Fixing the Model Before formalizing our fix, let us first rephrase conditions S1–S3 in the more concise style suggested by Batty *et al.* [5]. Instead of expressing them as separate conditions on a total order $\mathtt{S}$, they require a single *acylicity* condition, namely that $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{mo} \cup \mathtt{rb}); [\mathtt{E}^{\mathtt{sc}}]$ be acyclic. (In general, acyclicity of $\bigcup R_i$ is equivalent to the existence of a total order that contains $R_1, R_2, ...$)

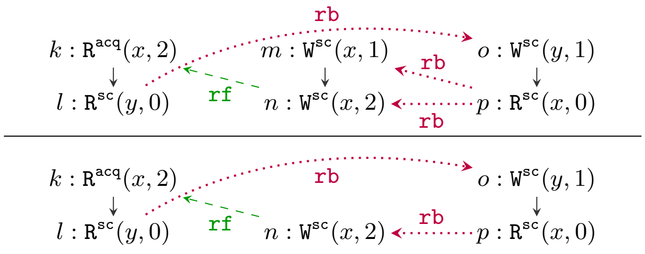
We propose to correct the condition by replacing $\mathtt{hb}$ with $\mathtt{sb} \cup \mathtt{sb}; \mathtt{hb}; \mathtt{sb} \cup \mathtt{hb}|_{\mathtt{loc}}$. Accordingly, we require that

$$[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{sb} \cup \mathtt{sb}; \mathtt{hb}; \mathtt{sb} \cup \mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}); [\mathtt{E}^{\mathtt{sc}}]$$

is acyclic. Note that this condition still ensures SC semantics for programs that have only SC accesses. Indeed, since $[\mathtt{E}^{\mathtt{sc}}]; \mathtt{rf}; [\mathtt{E}^{\mathtt{sc}}] \subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{sw}; [\mathtt{E}^{\mathtt{sc}}] \subseteq [\mathtt{E}^{\mathtt{sc}}]; \mathtt{hb}|_{\mathtt{loc}}; [\mathtt{E}^{\mathtt{sc}}]$, our condition implies acyclicity of $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}); [\mathtt{E}^{\mathtt{sc}}]$. The latter suffices for this purpose, as it corresponds exactly to the declarative definition of sequential consistency [28].

2.1.1 Enabling Elimination of SC Accesses

We observe that our condition disallows the elimination of an SC write immediately followed by another SC write to the same location, as well as of an SC read immediately preceded by an SC read from the same location. While neither `GCC` nor `LLVM` performs these eliminations, they are sound under sequential consistency, as well as under C11 (with the fixes of [30]), and one may wish to preserve their soundness.



rb
$k : \mathtt{R}^{\mathtt{acq}}(x,2)$
$m : \mathtt{W}^{\mathtt{sc}}(x,1)$
rb
$o : \mathtt{W}^{\mathtt{sc}}(y,1)$
$l : \mathtt{R}^{\mathtt{sc}}(y,0)$
rf
$n : \mathtt{W}^{\mathtt{sc}}(x,2)$
$p : \mathtt{R}^{\mathtt{sc}}(x,0)$
rb
$k : \mathtt{R}^{\mathtt{acq}}(x,2)$
rb
$o : \mathtt{W}^{\mathtt{sc}}(y,1)$
$l : \mathtt{R}^{\mathtt{sc}}(y,0)$
rf
$n : \mathtt{W}^{\mathtt{sc}}(x,2)$
rb
$p : \mathtt{R}^{\mathtt{sc}}(x,0)$

Figure 4. An abbreviated execution of WWmerge (source), and of the resulting program after eliminating the overwritten write m (target). The source execution has a disallowed cycle (m, l, o, p, m), while the target execution does not.

To see the unsoundness of eliminating an overwritten SC write, consider the following program. The annotated behavior is forbidden, but it will become allowed after eliminating $x :=_{\mathtt{sc}} 1$ (see Fig. 4).

$$\begin{array}{l||l||l} a := x_{\mathtt{acq}} \ /\!/2 & x :=_{\mathtt{sc}} 1 & y :=_{\mathtt{sc}} 1 \\ b := y_{\mathtt{sc}} \ /\!/0 & x :=_{\mathtt{sc}} 2 & c := x_{\mathtt{sc}} \ /\!/0 \end{array} \qquad \text{(WWmerge)}$$

Similarly, eliminating a repeated SC read is unsound (see example in [3]). The problem here is that these transformations remove an $\mathtt{sb}$ edge, and thus remove an $\mathtt{sb}; \mathtt{hb}; \mathtt{sb}$ path between two SC accesses.

Note that the removed $\mathtt{sb}$ edges are all edges between same-location accesses. Thus, supporting these transformations can be achieved by a slight weakening of our condition: we replace $\mathtt{sb}; \mathtt{hb}; \mathtt{sb}$ with $\mathtt{sb}|_{\neq\mathtt{loc}}; \mathtt{hb}; \mathtt{sb}|_{\neq\mathtt{loc}}$, where $\mathtt{sb}|_{\neq\mathtt{loc}}$ denotes $\mathtt{sb}$ edges that are not between accesses to the same location. Thus, we require acyclicity of $[\mathtt{E}^{\mathtt{sc}}]; \mathtt{scb}; [\mathtt{E}^{\mathtt{sc}}]$, where $\mathtt{scb}$ (*SC-before*) is given by:

$$\mathtt{scb} \triangleq \mathtt{sb} \cup \mathtt{sb}|_{\neq\mathtt{loc}}; \mathtt{hb}; \mathtt{sb}|_{\neq\mathtt{loc}} \cup \mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}.$$

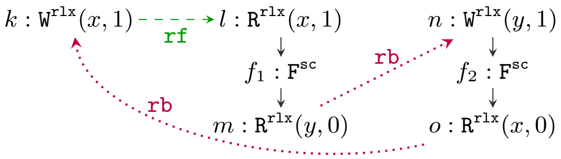
We note that this change does not affect programs that do not mix SC and non-SC accesses to the same location.

2.2 Second Problem: SC Fences are Too Weak

In this section we extend our model to cover SC fences, which were not considered so far. Denote by $\mathtt{F}^{\mathtt{sc}}$ the set of SC fences in $\mathtt{E}$. The straightforward adaptation of the condition of Batty *et al.* [5] for the full model (obtained by replacing $\mathtt{hb} \cup \mathtt{mo} \cup \mathtt{rb}$ with our $\mathtt{scb}$) is that

$$\mathtt{psc}_1 \triangleq \left([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{sb}^?\right); \mathtt{scb}; \left([\mathtt{E}^{\mathtt{sc}}] \cup \mathtt{sb}^?; [\mathtt{F}^{\mathtt{sc}}]\right)$$

is acyclic. This condition generalizes the earlier condition by forbidding $\mathtt{scb}$ cycles even between *non-SC* accesses provided they are preceded/followed by an SC fence. This condition rules out weak behaviors of examples such as SB and 2+2W where all accesses are relaxed and SC fences are placed between them in the two threads.



$k : \texttt{W}^{\texttt{rlx}}(x,1)$
rf
$l : \texttt{R}^{\texttt{rlx}}(x,1)$
$n : \texttt{W}^{\texttt{rlx}}(y,1)$
$f_1 : \texttt{F}^{\texttt{sc}}$
rb
$f_2 : \texttt{F}^{\texttt{sc}}$
rb
$m : \texttt{R}^{\texttt{rlx}}(y,0)$
$o : \texttt{R}^{\texttt{rlx}}(x,0)$

Figure 5. An execution of RWC+syncs yielding the annotated result. The $\texttt{rb}$ edges are due to the reading from the omitted initialization events and the $\texttt{mo}$ edges from those.

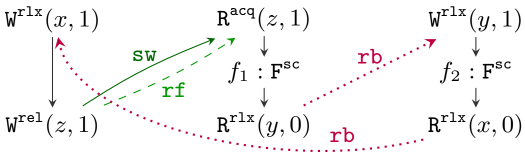
In general, one might expect that inserting an SC fence between every two instructions restores sequential consistency. This holds for hardware memory models, such as x86-TSO, Power, and ARM, for programs with aligned word-sized accesses (for their analogue of SC fences), but holds neither in the original C11 model nor in its strengthening [5] for two reasons. The first reason is that C11 declares that programs with racy non-atomic accesses have undefined behavior, and even if fences are placed everywhere such races may exist. There is, however, another way in which putting fences everywhere in C11 does not restore sequential consistency, even if all the accesses are atomic. Consider the following program:

$$x :=_{\texttt{rlx}} 1 \;\left\|\; \begin{array}{l} a := x_{\texttt{rlx}} \;/\!/1 \\ \texttt{fence}_{\texttt{sc}} \\ b := y_{\texttt{rlx}} \;/\!/0 \end{array} \;\right\|\; \begin{array}{l} y :=_{\texttt{rlx}} 1 \\ \texttt{fence}_{\texttt{sc}} \\ c := x_{\texttt{rlx}} \;/\!/0 \end{array} \qquad \text{(RWC+syncs)}$$

The annotated behavior is allowed according to the model of Batty *et al.* [5] (and so, also by our weaker condition above). Fig. 5 depicts a consistent execution yielding this behavior, as the only $\texttt{psc}_1$ edge is from f_1 to f_2. Yet, this behavior is disallowed by all implementations of C11. We believe that this is a serious omission of the standard rendering the SC fences too weak, as they cannot be used to enforce sequential consistency. This weakness has also been observed in a C11 implementation of the Chase-Lev deque by Lê *et al.* [21], who report that the weak semantics of SC fences in C11 requires them to unnecessarily strengthen the access modes of certain relaxed writes to SC. (In the context of the RWC+syncs, it would amount to making the write to x in the first thread into an SC write.)

Remark 2 (Itanium)**.** This particular weakness of the standard is attributed to Itanium, whose fences do not guarantee sequential consistency when inserted everywhere. While this would be a problem if C11 relaxed accesses were compiled to plain Itanium accesses, they actually have to be compiled to release/acquire Itanium accesses to guarantee read-read coherence. In this case, Itanium fences guarantee ordering. In fact, Itanium implementations provide multi-copy atomicity for release stores, and thus cannot yield the weak outcome of IRIW even without fences [14, §3.3.7.1].

Fixing the Semantics of SC Fences Analyzing the execution of RWC+syncs, we note that there is a $\texttt{sb};\texttt{rb};\texttt{rf};\texttt{sb}$ path from f_2 to f_1, but this path does not contribute to $\texttt{psc}_1$.



$\texttt{W}^{\texttt{rlx}}(x,1)$
$\texttt{R}^{\texttt{acq}}(z,1)$
$\texttt{W}^{\texttt{rlx}}(y,1)$
sw
$f_1 : \texttt{F}^{\texttt{sc}}$
rf
rb
$f_2 : \texttt{F}^{\texttt{sc}}$
$\texttt{W}^{\texttt{rel}}(z,1)$
$\texttt{R}^{\texttt{rlx}}(y,0)$
rb
$\texttt{R}^{\texttt{rlx}}(x,0)$

Figure 6. An abbreviated execution of W+RWC.

Although both $\texttt{rb}$ and $\texttt{rf}$ edges contribute to $\texttt{psc}_1$, their composition $\texttt{rb};\texttt{rf}$ does not.

To repair the model, we define the *extended coherence order*, $\texttt{eco} \triangleq (\texttt{rf} \cup \texttt{mo} \cup \texttt{rb})^+$. This order includes the reads-from relation, $\texttt{rf}$, the modification order, $\texttt{mo}$, the reads-before relation, $\texttt{rb}$, and also all the compositions of these relations with one another—namely, all orders forced because of the coherence axioms. Then, we require that $\texttt{psc}_1 \cup [\texttt{F}^{\texttt{sc}}];\texttt{sb};\texttt{eco};\texttt{sb};[\texttt{F}^{\texttt{sc}}]$ is acyclic.

This stronger condition rules out the weak behavior of RWC+syncs because there are $\texttt{sb};\texttt{eco};\texttt{sb}$ paths from one fence to another and vice versa (in one direction via the x accesses and in the other direction via the y accesses). Intuitively speaking, compilation remains correct with this stronger model since $\texttt{eco}$ exists only between accesses to the same location, on which the hardware provides strong ordering guarantees.

Now it is easy to see that, given a program without non-atomic accesses, placing an SC fence between every two accesses guarantees SC. Indeed, by the definition of SC, it suffices to show that $\texttt{eco} \cup \texttt{sb}$ is acyclic. Consider a $\texttt{eco} \cup \texttt{sb}$ cycle. Since $\texttt{eco}$ and $\texttt{sb}$ are irreflexive and transitive, the cycle necessarily has the form $(\texttt{eco};\texttt{sb})^+$. Thus, between every two $\texttt{eco}$ steps, there must be an SC fence. So in effect, we have a cycle in $\texttt{eco};\texttt{sb};[\texttt{F}^{\texttt{sc}}];\texttt{sb}$, which can be regrouped to a cycle in $[\texttt{F}^{\texttt{sc}}];\texttt{sb};\texttt{eco};\texttt{sb};[\texttt{F}^{\texttt{sc}}]$, which is forbidden by our model.

Finally, one might further consider strengthening the model by including $\texttt{eco}$ in $\texttt{scb}$ (which is used to define $\texttt{psc}_1$), thereby ruling out the weak behavior of a variant of RWC+syncs using SC accesses instead of SC fences in threads 2 and 3. We note, however, that this strengthening is unsound for the default compilation scheme to x86-TSO (see Remark 4 in §4).

2.2.1 Restoring Fence Cumulativity

Consider the following variant of the store buffering program, where the write of $x := 1$ has been moved to another thread with a release-acquire synchronization.

$$\begin{array}{l} x :=_{\texttt{rlx}} 1 \\ z :=_{\texttt{rel}} 1 \end{array} \;\left\|\; \begin{array}{l} a := z_{\texttt{acq}} \;/\!/1 \\ \texttt{fence}_{\texttt{sc}} \\ b := y_{\texttt{rlx}} \;/\!/0 \end{array} \;\right\|\; \begin{array}{l} y :=_{\texttt{rlx}} 1 \\ \texttt{fence}_{\texttt{sc}} \\ c := x_{\texttt{rlx}} \;/\!/0 \end{array} \qquad \text{(W+RWC)}$$

The annotated behavior corresponds to the writes of x and y being observed in different orders by the reads, although SC fences have been used in the observer threads. This behavior is disallowed on x86, Power, and ARM because their fences are cumulative: the fences order not only the writes performed by the thread with the fence instruction, but also the writes of other threads that are observed by the thread in question [23].

In contrast, the behavior is allowed by the model described thus far. Consider the execution shown in Fig. 6. While there is a $\mathtt{sb}; \mathtt{rb}; \mathtt{sb}$ path from f_1 to f_2, the only path from f_2 back to f_1 is $\mathtt{sb}; \mathtt{rb}; \mathtt{sb}; \mathtt{sw}; \mathtt{sb}$ (or, more generally, $\mathtt{hb}; \mathtt{rb}; \mathtt{hb}$), and so the execution is allowed.

To disallow such behaviors, we can replace $[\mathtt{F}^{\mathtt{sc}}]; \mathtt{sb}$ and $\mathtt{sb}; [\mathtt{F}^{\mathtt{sc}}]$ in the definitions above by $[\mathtt{F}^{\mathtt{sc}}]; \mathtt{hb}$ and $\mathtt{hb}; [\mathtt{F}^{\mathtt{sc}}]$.[3] This leads us to our final condition that requires that $\mathtt{psc}_{\mathrm{base}} \cup \mathtt{psc}_{\mathtt{F}}$ is acyclic, where:

$$\mathtt{psc}_{\mathrm{base}} \triangleq \left([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{hb}^{?}\right); \mathtt{scb}; \left([\mathtt{E}^{\mathtt{sc}}] \cup \mathtt{hb}^{?}; [\mathtt{F}^{\mathtt{sc}}]\right)$$
$$\mathtt{psc}_{\mathtt{F}} \triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathtt{sc}}]$$

We note that $[\mathtt{F}^{\mathtt{sc}}]; \mathtt{psc}_{\mathrm{base}}; [\mathtt{F}^{\mathtt{sc}}] \subseteq \mathtt{psc}_{\mathtt{F}}$. Hence, in programs without SC accesses (but with SC fences) it suffices to require that $\mathtt{psc}_{\mathtt{F}}$ is acyclic.

2.3 A Final Problem: Out-of-Thin-Air Reads

The C11 memory model suffers from a major problem, known as the "out-of-thin-air problem" [31, 11]. Designed to allow efficient compilation and many optimization opportunities for relaxed accesses, the model happened to be too weak, admitting "out-of-thin-air" behaviors, which no implementation exhibits. The standard example is load buffering with some form of dependencies in both threads:

$$\begin{array}{l||l} a := x_{\mathtt{rlx}} \;/\!/ 1 & b := y_{\mathtt{rlx}} \;/\!/ 1 \\ \textbf{if}\,(a)\; y :=_{\mathtt{rlx}} a & \textbf{if}\,(b)\; x :=_{\mathtt{rlx}} b \end{array} \qquad \text{(LB+deps)}$$

In this program, the formalized C11 model by Batty *et al.* [8] allows reading $a = b = 1$ even though the value 1 does not appear in the program. The reason is that the execution where both threads read and write the value 1 is consistent: each read reads from the write of the other thread. As one might expect, such behaviors are very problematic because they invalidate almost all forms of formal reasoning about programs. In particular, the example above demonstrates a violation of DRF-SC, the most basic guarantee that users of C11 were intended to assume: LB+deps has no races under sequential consistency, and yet has some non-SC behavior.

Fixing the model in a way that forbids all "out-of-thin-air" behaviors and still allows the most efficient compilation is beyond the scope of the current paper (see [16] for a possible solution). In this paper, we settle for a simpler solution of requiring $\mathtt{sb} \cup \mathtt{rf}$ to be acyclic. This is a relatively straightforward way to avoid the problem, although it carries some performance cost. Clearly, it rules out the weak behavior of LB+deps, but also of the following load-buffering program, which is nevertheless permitted by the Power and ARM architectures.

$$\begin{array}{l||l} a := x_{\mathtt{rlx}} \;/\!/ 1 & b := y_{\mathtt{rlx}} \;/\!/ 1 \\ y :=_{\mathtt{rlx}} 1 & x :=_{\mathtt{rlx}} 1 \end{array} \qquad \text{(LB)}$$

To correctly compile the stronger model to Power and ARM, one has to either introduce a fence between a relaxed atomic read and a subsequent relaxed atomic write or a forced dependency between every such pair of accesses [11]. The latter can be achieved by inserting a dummy control-dependent branch after every relaxed atomic read.

While the idea of strengthening C11 to require acyclicity of $\mathtt{sb} \cup \mathtt{rf}$ is well known [31, 11], we are not aware of any proof showing that the proposed compilation schemes of Boehm and Demsky [11] are correct, nor that DRF-SC holds under this assumption. The latter is essential for assessing our corrected model, as it is a key piece of evidence showing that our semantics for SC accesses is not overly weak.

Importantly, even in this stronger model, non-atomic accesses are compiled to plain machine loads and stores. This is what makes the compilation correctness proof highly non-trivial, as the hardware models allow certain $\mathtt{sb} \cup \mathtt{rf}$ cycles involving plain loads and stores. As a result, one has to rely on the *"catch-fire"* semantics (races on non-atomic accesses result in undefined behavior) for explaining behaviors that involve such cycles. A similar argument is needed for proving the correctness of non-atomic read-write reordering.

3. The Proposed Memory Model

In this section, we formally define our proposed corrected version of the C11 model, which we call RC11. Similar to C11, the RC11 model is given in a "declarative" style in three steps: we associate a set of graphs (called *executions*) to every program (§3.1), filter this set by imposing a consistency predicate (§3.2), and finally define the outcomes of a program based on the set of its consistent executions (§3.3). At the end of the section, we compare our model with C11 (§3.4).

Before we start, we introduce some further notation. Given a binary relation R, $\mathit{dom}(R)$ and $\mathit{codom}(R)$ denote its domain and codomain. Given a function f, $=_f$ denotes the set of f-equivalent pairs ($=_f \triangleq \{\langle a, b\rangle \mid f(a) = f(b)\}$), and $R|_f$ denotes the restriction of R to f-equivalent pairs ($R|_f \triangleq R \cap =_f$). When R is a strict partial order, $R|_{\mathrm{imm}}$ denotes the set of all *immediate R edges*, *i.e.*, pairs $\langle a, b\rangle \in R$ such that for every c, $\langle c, b\rangle \in R$ implies $\langle c, a\rangle \in R^?$, and $\langle a, c\rangle \in R$ implies $\langle b, c\rangle \in R^?$.

We assume finite sets Loc and Val of locations and values. We use x, y, z as metavariables for locations and v for values. The model supports several modes for accesses and fences, partially ordered by $\sqsubset$ as follows:

$$\begin{tikzcd}[column sep=small, row sep=small] & & \mathtt{rel} \arrow[dr] & & \\ \mathtt{na} \arrow[r] & \mathtt{rlx} \arrow[ur] \arrow[dr] & & \mathtt{acqrel} \arrow[r] & \mathtt{sc} \\ & & \mathtt{acq} \arrow[ur] & & \end{tikzcd}$$

[3] To rule out only the cycle shown in Fig. 6, it would suffice to have replaced only the $\mathtt{sb}$ *to* a fence by an $\mathtt{hb}$. We can, however, also construct examples, where it is useful for the $\mathtt{sb}$ *from* a fence to be replaced by $\mathtt{hb}$.

3.1 From Programs to Executions

First, the program is translated into a set of executions. An *execution* G consists of:

1. a finite set of events $\mathtt{E} \subseteq \mathbb{N}$ containing a distinguished set $\mathtt{E}_0 = \{a_0^x \mid x \in \mathsf{Loc}\}$ of initialization events. We use $a, b, \ldots$ as metavariables for events.
2. a function $\mathtt{lab}$ assigning a *label* to every event in $\mathtt{E}$. Labels are of one of the following forms:
   - $\mathtt{R}^o(x, v)$ where $o \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}, \mathtt{sc}\}$.
   - $\mathtt{W}^o(x, v)$ where $o \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{rel}, \mathtt{sc}\}$.
   - $\mathtt{F}^o$ where $o \in \{\mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$.

   We assume that $\mathtt{lab}(a_0^x) = \mathtt{W}^{\mathtt{na}}(x, 0)$ for every $a_0^x \in \mathtt{E}_0$.

   $\mathtt{lab}$ naturally induces the functions $\mathtt{typ}$, $\mathtt{mod}$, $\mathtt{loc}$, $\mathtt{val}_\mathtt{r}$, and $\mathtt{val}_\mathtt{w}$ that return (when applicable) the type ($\mathtt{R}$, $\mathtt{W}$ or $\mathtt{F}$), mode, location, and read/written value of an event.

   For $\mathtt{T} \in \{\mathtt{R}, \mathtt{W}, \mathtt{F}\}$, $\mathtt{T}$ denotes the set $\{e \in \mathtt{E} \mid \mathtt{typ}(e) = \mathtt{T}\}$. We also concatenate the event sets notations, use subscripts to denote the accessed location, and superscripts for modes (*e.g.,* $\mathtt{RW} = \mathtt{R} \cup \mathtt{W}$ and $\mathtt{W}_x^{\sqsupseteq \mathtt{rel}}$ denotes all events $a \in \mathtt{W}$ with $\mathtt{loc}(a) = x$ and $\mathtt{mod}(a) \sqsupseteq \mathtt{rel}$).
3. a strict partial order $\mathtt{sb} \subseteq \mathtt{E} \times \mathtt{E}$, called *sequenced-before,* which orders the initialization events before all other events, *i.e.,* $\mathtt{E}_0 \times (\mathtt{E} \setminus \mathtt{E}_0) \subseteq \mathtt{sb}$.
4. a binary relation $\mathtt{rmw} \subseteq [\mathtt{R}]; (\mathtt{sb}|_{\mathrm{imm}} \cap =_{\mathtt{loc}}); [\mathtt{W}]$, called *read-modify-write pairs*, such that for every $\langle a, b\rangle \in \mathtt{rmw}$, $\langle \mathtt{mod}(a), \mathtt{mod}(b)\rangle$ is one of the following:
   - $\langle \mathtt{rlx}, \mathtt{rlx}\rangle$ ($\mathtt{RMW}^{\mathtt{rlx}}$)
   - $\langle \mathtt{acq}, \mathtt{rlx}\rangle$ ($\mathtt{RMW}^{\mathtt{acq}}$)
   - $\langle \mathtt{rlx}, \mathtt{rel}\rangle$ ($\mathtt{RMW}^{\mathtt{rel}}$)
   - $\langle \mathtt{acq}, \mathtt{rel}\rangle$ ($\mathtt{RMW}^{\mathtt{acqrel}}$)
   - $\langle \mathtt{sc}, \mathtt{sc}\rangle$ ($\mathtt{RMW}^{\mathtt{sc}}$)

   We denote by $\mathtt{At}$ the set of all events in $\mathtt{E}$ that are a part of an $\mathtt{rmw}$ edge (that is, $\mathtt{At} = \mathit{dom}(\mathtt{rmw}) \cup \mathit{codom}(\mathtt{rmw})$).

   Note that our executions represent RMWs differently from C11 executions. Here each RMW is represented as two events, a read and a write, related by the $\mathtt{rmw}$ relation, whereas in C11 they are represented by single $\mathtt{RMW}$ events, which act as both the read and the write of the RMW. Our choice is in line with the Power and ARM memory models, and simplifies the formal development (*e.g.,* the definition of receptiveness).
5. a binary relation $\mathtt{rf} \subseteq [\mathtt{W}]; =_{\mathtt{loc}}; [\mathtt{R}]$, called *reads-from*, satisfying (i) $\mathtt{val}_\mathtt{w}(a) = \mathtt{val}_\mathtt{r}(b)$ for every $\langle a, b\rangle \in \mathtt{rf}$; and (ii) $a_1 = a_2$ whenever $\langle a_1, b\rangle, \langle a_2, b\rangle \in \mathtt{rf}$.
6. a strict partial order $\mathtt{mo}$ on $\mathtt{W}$, called *modification order*, which is a disjoint union of relations $\{\mathtt{mo}_x\}_{x \in \mathsf{Loc}}$, such that each $\mathtt{mo}_x$ is a strict total order on $\mathtt{W}_x$.

In what follows, to resolve ambiguities, we may include a prefix "G." to refer to the components of an execution G.

Executions of a given program represent prefixes of traces of shared memory accesses and fences that are generated by the program. In this paper, we only consider "partitioned" programs of the form $\|_{i \in \mathsf{Tid}}\ c_i$, where Tid is a finite set of thread identifiers, $\|$ denotes parallel composition, and each c_i is a sequential program. Then, the set of executions associated with a given program is defined by induction over the structure of sequential programs. We do not define formally this construction (it depends on the particular syntax and features of the source programming language). In this initial stage the read values are not restricted whatsoever (and $\mathtt{rf}$ and $\mathtt{mo}$ are arbitrary). Note that the set of executions of a program P is taken to be *prefix-closed*: an $\mathtt{sb}$-prefix of an execution of P (which includes at least the initialization events) is also considered to be an execution of P. By *full* executions of P, we refer to executions that represent traces generated by the whole program P.

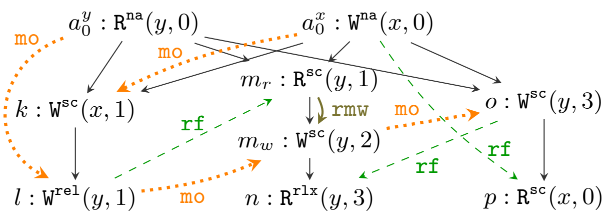

$a_0^y : \mathtt{R}^{\mathtt{na}}(y, 0)$
$a_0^x : \mathtt{W}^{\mathtt{na}}(x, 0)$
mo
mo
$m_r : \mathtt{R}^{\mathtt{sc}}(y, 1)$
$k : \mathtt{W}^{\mathtt{sc}}(x, 1)$
rmw
mo
$o : \mathtt{W}^{\mathtt{sc}}(y, 3)$
rf
$m_w : \mathtt{W}^{\mathtt{sc}}(y, 2)$
rf
rf
$l : \mathtt{W}^{\mathtt{rel}}(y, 1)$
mo
$n : \mathtt{R}^{\mathtt{rlx}}(y, 3)$
$p : \mathtt{R}^{\mathtt{sc}}(x, 0)$

Figure 7. An execution of Z6.U.

We show an example of an execution in Fig. 7. This is a full execution of the Z6.U program, and is essentially the same as the C11 execution shown in Fig. 3, except for the representation of RMWs (see Item 4 above).

3.2 Consistent Executions

The main part of the memory model is filtering the consistent executions among all executions of the program. The first obvious restriction is that every read should read some written value (formally, $\mathtt{R} \subseteq \mathit{codom}(\mathtt{rf})$). We refer to such executions as *complete*.

To state the other constraints we use a number of derived relations:

$$\mathtt{rb} \triangleq \mathtt{rf}^{-1}; \mathtt{mo} \qquad \textit{(reads-before)}$$

$$\mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^+ \qquad \textit{(extended coherence order)}$$

$$\mathtt{rs} \triangleq [\mathtt{W}]; \mathtt{sb}|_{\mathtt{loc}}^?; [\mathtt{W}^{\sqsupseteq \mathtt{rlx}}]; (\mathtt{rf}; \mathtt{rmw})^* \qquad \textit{(release sequence)}$$

$$\begin{aligned}\mathtt{sw} \triangleq\ & [\mathtt{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathtt{F}]; \mathtt{sb})^?; \mathtt{rs}; \mathtt{rf}; \\ & [\mathtt{R}^{\sqsupseteq \mathtt{rlx}}]; (\mathtt{sb}; [\mathtt{F}])^?; [\mathtt{E}^{\sqsupseteq \mathtt{acq}}]\end{aligned} \qquad \textit{(synchronizes with)}$$

$$\mathtt{hb} \triangleq (\mathtt{sb} \cup \mathtt{sw})^+ \qquad \textit{(happens-before)}$$

The first two, $\mathtt{rb}$ and $\mathtt{eco}$, are as described previously. Note that since the modification order, $\mathtt{mo}$, is transitive, we have $\mathtt{eco} = \mathtt{rf} \cup (\mathtt{mo} \cup \mathtt{rb}); \mathtt{rf}^?$ in every execution.

The other three relations, $\mathtt{rs}$, $\mathtt{sw}$ and $\mathtt{hb}$, are taken from [30]. Intuitively, $\mathtt{hb}$ records when an event is globally perceived as occurring before another one. It is defined in terms of two more basic relations. First, the *release sequence* ($\mathtt{rs}$) of a write contains the write itself and all later atomic writes to the same location in the same thread, as well as all RMWs that recursively read from such writes. Next, a release event a *synchronizes with* ($\mathtt{sw}$) an acquire event b, whenever b (or, in case b is a fence, some $\mathtt{sb}$-prior read) reads from the release sequence of a (or in case a is a fence, of some $\mathtt{sb}$-later write). Then, we say that an event a *happens-before* ($\mathtt{hb}$) an event b if there is a path from a to b consisting of $\mathtt{sb}$ and $\mathtt{sw}$ edges.

Finally, we define the *SC-before* relation, $\mathtt{scb}$, and the *partial SC* relations, $\mathtt{psc}_{\text{base}}$ and $\mathtt{psc}_{\mathtt{F}}$, as follows:

$$
\begin{aligned}
\mathtt{sb}|_{\neq\mathtt{loc}} &\triangleq \mathtt{sb} \setminus \mathtt{sb}|_{\mathtt{loc}} \\
\mathtt{scb} &\triangleq \mathtt{sb} \cup \mathtt{sb}|_{\neq\mathtt{loc}}; \mathtt{hb}; \mathtt{sb}|_{\neq\mathtt{loc}} \cup \mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb} \\
\mathtt{psc}_{\text{base}} &\triangleq \left([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{hb}^{?}\right); \mathtt{scb}; \left([\mathtt{E}^{\mathtt{sc}}] \cup \mathtt{hb}^{?}; [\mathtt{F}^{\mathtt{sc}}]\right) \\
\mathtt{psc}_{\mathtt{F}} &\triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathtt{sc}}] \\
\mathtt{psc} &\triangleq \mathtt{psc}_{\text{base}} \cup \mathtt{psc}_{\mathtt{F}}
\end{aligned}
$$

Using these derived relations, RC11 imposes four constraints on executions:

Definition 1. An execution G is called RC11-*consistent* if it is complete and the following hold:

- $\mathtt{hb}; \mathtt{eco}^{?}$ is irreflexive. (COHERENCE)
- $\mathtt{rmw} \cap (\mathtt{rb}; \mathtt{mo}) = \emptyset$. (ATOMICITY)
- $\mathtt{psc}$ is acyclic. (SC)
- $\mathtt{sb} \cup \mathtt{rf}$ is acyclic. (NO-THIN-AIR)

COHERENCE ensures that programs with only one shared location are sequentially consistent, as at least two locations are needed for a cycle in $\mathtt{sb} \cup \mathtt{eco}$. ATOMICITY ensures that the read and the write comprising a RMW are adjacent in $\mathtt{eco}$: there is no write event in between. The SC condition is the main novelty of RC11 and is used to give semantics to SC accesses and fences. Finally, NO-THIN-AIR rules out thin-air behaviors, albeit at a performance cost, as we will see in §5.

3.3 Program Outcomes

Finally, in order to allow the compilation of non-atomic read and writes to plain machine load and store instructions (as well as the compiler to reorder such accesses), RC11 follows the *"catch-fire"* approach: races on non-atomic accesses result in undefined behavior, that is, any outcome is allowed. Formally, it is defined as follows.

Definition 2. Two events a and b are called *conflicting* in an execution G if $a, b \in \mathtt{E}$, $\mathtt{W} \in \{\mathtt{typ}(a), \mathtt{typ}(b)\}$, $a \neq b$, and $\mathtt{loc}(a) = \mathtt{loc}(b)$. A pair $\langle a, b\rangle$ is called a *race* in G (denoted $\langle a, b\rangle \in \mathtt{race}$) if a and b are conflicting events in G, and $\langle a, b\rangle \notin \mathtt{hb} \cup \mathtt{hb}^{-1}$.

Definition 3. An execution G is called *racy* if there is some $\langle a, b\rangle \in \mathtt{race}$ with $\mathtt{na} \in \{\mathtt{mod}(a), \mathtt{mod}(b)\}$. A program P has *undefined behavior* under RC11 if it has some racy RC11-consistent execution.

Definition 4. The *outcome* of an execution G is the function assigning to every location x the value written by the $\mathtt{mo}$-maximal event in $\mathtt{W}_x$. We say that $O : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome of a program P under* RC11 if either O is an outcome of some RC11-consistent full execution of P, or P has undefined behavior under RC11.

3.4 Comparison with C11

Besides the new SC and NO-THIN-AIR conditions, RC11 differs in a few other ways from C11.

- It does not support *consume* accesses, a premature feature of C11 that is not implemented by major compilers, nor *locks*, as they can be straightforwardly implemented with release-acquire accesses.
- For simplicity, it assumes all locations are initialized.
- It incorporates the fixes proposed by Vafeiadis *et al.* [30], namely (i) the strengthening of the release sequences definition, (ii) the removal of restrictions about different threads in the definition of synchronization, and (iii) the lack of distinction between atomic and non-atomic locations (and accordingly omitting the problematic $\mathtt{rf} \subseteq \mathtt{hb}$ condition for non-atomic locations). The third fix avoids "out-of-thin-air" problems that arise when performing non-atomic accesses to atomic location [6, §5].
- It does not consider "unsequenced races" between atomic accesses to have undefined behavior. Our results are not affected by such undefined behavior.

We have also made three presentational changes: (1) we have a much more concise axiomatization of coherence; (2) we model RMWs using two events; and (3) we do not have a total order over SC atomics.

Proposition 1. *RC11's* COHERENCE *condition is equivalent to the conjunction of the following constraints of C11:*

- $\mathtt{hb}$ *is irreflexive.* (IRREFLEXIVE-HB)
- $\mathtt{rf}; \mathtt{hb}$ *is irreflexive.* (NO-FUTURE-READ)
- $\mathtt{mo}; \mathtt{rf}; \mathtt{hb}$ *is irreflexive.* (COHERENCE-RW)
- $\mathtt{mo}; \mathtt{hb}$ *is irreflexive.* (COHERENCE-WW)
- $\mathtt{mo}; \mathtt{hb}; \mathtt{rf}^{-1}$ *is irreflexive.* (COHERENCE-WR)
- $\mathtt{mo}; \mathtt{rf}; \mathtt{hb}; \mathtt{rf}^{-1}$ *is irreflexive.* (COHERENCE-RR)

Proposition 2. *The* SC *condition is equivalent to requiring the existence of a total strict order* $\mathtt{S}$ *on* $\mathtt{E}^{\mathtt{sc}}$ *such that* $\mathtt{S}; \mathtt{psc}$ *is irreflexive.*

Finally, the next proposition ensures that without mixing SC and non-SC accesses to the same location, RC11 supplies the stronger guarantee of C11. As a consequence, programmers that never mix such accesses may completely ignore the difference between RC11 and C11 regarding SC accesses.

Proposition 3. *If SC accesses are to distinguished locations (for every* $a, b \in \mathtt{E} \setminus \mathtt{E}_0$*, if* $\mathtt{mod}(a) = \mathtt{sc}$ *and* $\mathtt{loc}(a) = \mathtt{loc}(b)$ *then* $\mathtt{mod}(b) = \mathtt{sc}$*) then* $[\mathtt{E}^{\mathtt{sc}}]; \mathtt{hb}; [\mathtt{E}^{\mathtt{sc}}] \subseteq \mathtt{psc}^{+}$.

4. Compilation to x86-TSO

In this section, we present the x86-TSO memory model, and show that its intended compilation scheme is sound. We use a

$$\begin{array}{llll}
\llparenthesis \mathtt{R} \rrparenthesis & \triangleq \mathtt{MOV}\ \text{(from memory)} & \llparenthesis \mathtt{W}^{\sqsubseteq \mathtt{rel}} \rrparenthesis & \triangleq \mathtt{MOV}\ \text{(to memory)} \\
\llparenthesis \mathtt{W}^{\mathtt{sc}} \rrparenthesis & \triangleq \mathtt{MOV;MFENCE} & \llparenthesis \mathtt{RMW} \rrparenthesis & \triangleq \mathtt{CMPXCHG} \\
\llparenthesis \mathtt{F}^{\neq \mathtt{sc}} \rrparenthesis & \triangleq \text{No operation} & \llparenthesis \mathtt{F}^{\mathtt{sc}} \rrparenthesis & \triangleq \mathtt{MFENCE}
\end{array}$$

Figure 8. Compilation to TSO.

declarative model of x86-TSO from [17], that we denote by TSO. By [25, Theorem 3] and [17, Theorem 5], this definition is equivalent to the better known operational one. TSO executions are similar to the ones defined above, with the following exceptions:

- Read/write/fence labels have the form $\mathtt{R}(x, v)$, $\mathtt{W}(x, v)$, and $\mathtt{F}$ (they do not include a "mode"). In addition, labels may also be $\mathtt{RMW}(x, v_r, v_w)$, and executions do not include an $\mathtt{rmw}$ component (*i.e.,* RMWs are represented with a single event). We use $\mathtt{RMW}$ to denote the set of all events $a \in \mathtt{E}$ with $\mathtt{typ}(a) = \mathtt{RMW}$.
- The modification order, $\mathtt{mo}$, is a strict total order on $\mathtt{W} \cup \mathtt{RMW} \cup \mathtt{F}$ (rather than a union of total order on writes to the same location).
- Happens-before is given by $\mathtt{hb} \triangleq (\mathtt{sb} \cup \mathtt{rf})^+$.
- Reads-before is given by $\mathtt{rb} \triangleq \mathtt{rf}^{-1}; \mathtt{mo}|_{\mathtt{loc}} \setminus [\mathtt{E}]$.

Remark 3. Lahav *et al.* [17] treat fence instructions as syntactic sugar for RMWs of a distinguished location. Here, we have fences as primitive instructions that induce fence events in TSO executions.

Definition 5. A TSO execution G is TSO-*consistent* if it is complete and the following hold:

1. $\mathtt{hb}$ is irreflexive.
2. $\mathtt{mo}; \mathtt{hb}$ is irreflexive.
3. $\mathtt{rb}; \mathtt{hb}$ is irreflexive.
4. $\mathtt{rb}; \mathtt{mo}$ is irreflexive.
5. $\mathtt{rb}; \mathtt{mo}; \mathtt{rfe}; \mathtt{sb}$ is irreflexive (where $\mathtt{rfe} = \mathtt{rf} \setminus \mathtt{sb}$).
6. $\mathtt{rb}; \mathtt{mo}; [\mathtt{RMW} \cup \mathtt{F}]; \mathtt{sb}$ is irreflexive.

Unlike RC11, well-formed TSO programs do not have undefined behavior. Thus, a function $O : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome of a* TSO *program* P if it is an outcome of some TSO-consistent full execution of P (see Def. 4).

Fig. 8 presents the compilation scheme from C11 to x86-TSO that is implemented in the GCC and the LLVM compilers. Since TSO provides strong consistency guarantees, it allows most language primitives to be compiled to plain loads and stores. Barriers are only needed for the compilation of SC writes. Our next theorem says that this compilation scheme is also correct for RC11.

Theorem 1. *For a program P, denote by $\llparenthesis P \rrparenthesis$ the* TSO *program obtained by compiling P using the scheme in Fig. 8. Then, given a program P, every outcome of $\llparenthesis P \rrparenthesis$ under* TSO *is an outcome of P under* RC11.

Proof (Outline). We consider the compilation as if it happens in three steps, and prove the soundness of each step:

1. All non-atomic/relaxed accesses are strengthened to release/acquire ones, and all relaxed/release/acquire RMWs are strengthened to acquire-release ones. It is easy to see that this step does not introduce new outcomes (see §7).
2. All non-SC fences are removed. Due to the previous step, it is easy to show that non-SC fences have no effect.
3. The mappings in Fig. 8 are applied. The correctness of this step, given in [3], is established by showing that given aTSO-consistent TSO execution G_t of $\llparenthesis P \rrparenthesis$ (where P has no non-SC fences), there exists an RC11-consistent execution G of P that has the same outcome as G_t. □

In fact, the proof of Thm. 1 establishes the correctness of compilation even for a strengthening of RC11 obtained by replacing the $\mathtt{scb}$ relation by $\mathtt{scb}' \triangleq \mathtt{hb} \cup \mathtt{mo} \cup \mathtt{rb}$. This entails that the original C11 model, as well as Batty *et al.*'s strengthening [5], are correctly compiled to x86-TSO. Additionally, the proof only assumes the existence of an MFENCE between every store originated from an SC write and load originated from an SC read. The compilation scheme in Fig. 8 achieves this by placing an MFENCE after each store that originated from an SC write. An alternative correct compilation scheme may place MFENCE before SC reads, rather than after SC writes [1]. (Since there are typically more SC reads than SC writes in programs, the latter scheme is less preferred.)

Remark 4. The compilation scheme that places MFENCE before SC reads can be shown to be sound even for a very strong SC condition that requires acyclicity of

$$\mathtt{psc}_{\text{strong}} = ([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{hb}^?); (\mathtt{hb} \cup \mathtt{eco}); ([\mathtt{E}^{\mathtt{sc}}] \cup \mathtt{hb}^?; [\mathtt{F}^{\mathtt{sc}}]).$$

To prove this (see [3]), we are able to follow a simpler approach utilizing the recent result of Lahav and Vafeiadis [19] that provides a characterization of TSO in terms of program transformations (or "compiler optimizations"). This allows one to reduce compilation correctness to soundness of certain transformations. The preferred compilation scheme to x86-TSO, which uses barriers after SC writes (see Fig. 8), is unsound if one requires acyclicity of $\mathtt{psc}_{\text{strong}}$, or even if one requires acyclicity of $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{sb} \cup \mathtt{eco}); [\mathtt{E}^{\mathtt{sc}}]$. To see this, consider the following variant of SB:

$$\begin{array}{l||l}
x :=_{\mathtt{rel}} 1 & y :=_{\mathtt{rel}} 1 \\
a := x_{\mathtt{sc}} \ \text{// 1} & c := y_{\mathtt{sc}} \ \text{// 1} \\
b := y_{\mathtt{sc}} \ \text{// 0} & d := x_{\mathtt{sc}} \ \text{// 0}
\end{array} \qquad \text{(SB+rfis)}$$

Any execution of this program that yields the annotated behavior has a cycle in $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{sb} \cup \mathtt{eco}); [\mathtt{E}^{\mathtt{sc}}]$ (we have $\mathtt{rb}; \mathtt{rf}$ both from $\mathtt{R}^{\mathtt{sc}}(x, 0)$ to $\mathtt{R}^{\mathtt{sc}}(x, 1)$, and from $\mathtt{R}^{\mathtt{sc}}(y, 0)$ to $\mathtt{R}^{\mathtt{sc}}(y, 1)$). However, since the program has no SC writes, following Fig. 8, all accesses are compiled to plain accesses, and x86-TSO clearly allows this behavior.

5. Compilation to Power

In this section, we present the Power model and the mappings of language operations to Power instructions. We then prove the correctness of compilation from RC11 to Power.

As a model of the Power architecture, we use the recent declarative model by Alglave *et al.* [4], which we denote by Power. Its executions are similar to the RC11's execution, with the following exceptions:

- Power executions track syntactic dependencies between events in the same thread, and derive a relation called *preserved program order*, denoted $\mathtt{ppo}$, which is a subset of $\mathtt{sb}$ guaranteed to be preserved. The exact definition of $\mathtt{ppo}$ is quite intricate, and is included in [3].
- Read/write labels have the form $\mathtt{R}(x, v)$ and $\mathtt{W}(x, v)$ (they do not include a "mode"). Power has two types of fence events: a "lightweight fence" and a "full fence". We denote by $\mathtt{F}^{\mathtt{lwsync}}$ and $\mathtt{F}^{\mathtt{sync}}$ the set of all lightweight fence and full fence events in a Power execution. Power's "instruction fence" ($\mathtt{isync}$) is used to derive $\mathtt{ppo}$ but is not recorded in executions.

In addition to $\mathtt{ppo}$, the following additional derived relations are needed to define Power-consistency (see [4] for further explanations and details).

- $\mathtt{sync} \triangleq [\mathtt{RW}]; \mathtt{sb}; [\mathtt{F}^{\mathtt{sync}}]; \mathtt{sb}; [\mathtt{RW}]$
- $\mathtt{lwsync} \triangleq [\mathtt{RW}]; \mathtt{sb}; [\mathtt{F}^{\mathtt{lwsync}}]; \mathtt{sb}; [\mathtt{RW}] \setminus (\mathtt{W} \times \mathtt{R})$
- $\mathtt{fence} \triangleq \mathtt{sync} \cup \mathtt{lwsync}$ *(fence order)*
- $\mathtt{hb} \triangleq \mathtt{ppo} \cup \mathtt{fence} \cup \mathtt{rfe}$ (*Power's happens-before*)
- $\mathtt{prop}_1 \triangleq [\mathtt{W}]; \mathtt{rfe}^?; \mathtt{fence}; \mathtt{hb}^*; [\mathtt{W}]$
- $\mathtt{prop}_2 \triangleq (\mathtt{moe} \cup \mathtt{rbe})^?; \mathtt{rfe}^?; (\mathtt{fence}; \mathtt{hb}^*)^?; \mathtt{sync}; \mathtt{hb}^*$
- $\mathtt{prop} \triangleq \mathtt{prop}_1 \cup \mathtt{prop}_2$ *(propagation relation)*

where for every relation $\mathtt{c}$ (e.g., $\mathtt{rf}$, $\mathtt{mo}$, etc.), we denote by $\mathtt{ce}$ its thread-external restriction. Formally, $\mathtt{ce} = \mathtt{c} \setminus \mathtt{sb}$.

Definition 6. A Power execution G is Power-*consistent* if it is complete and the following hold:

1. $\mathtt{sb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{rb} \cup \mathtt{mo}$ is acyclic. (SC-PER-LOC)
2. $\mathtt{rbe}; \mathtt{prop}; \mathtt{hb}^*$ is irreflexive. (OBSERVATION)
3. $\mathtt{mo} \cup \mathtt{prop}$ is acyclic. (PROPAGATION)
4. $\mathtt{rmw} \cap (\mathtt{rbe}; \mathtt{moe})$ is irreflexive. (POWER-ATOMICITY)
5. $\mathtt{hb}$ is acyclic. (POWER-NO-THIN-AIR)

Remark 5. The model in [4] contains an additional constraint: $\mathtt{mo} \cup [\mathtt{At}]; \mathtt{sb}; [\mathtt{At}]$ should be acyclic (recall that $\mathtt{At} = \mathit{dom}(\mathtt{rmw}) \cup \mathit{codom}(\mathtt{rmw})$). Since none of our proofs requires this property, we excluded it from Def. 6.

Like in the case of TSO, we say that a function $O : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome of a* Power *program* P if it is an outcome of some Power-consistent full execution of P (see Def. 4).

$$\begin{array}{llll}
\llparenthesis \mathtt{R}^{\mathtt{na}} \rrparenthesis & \triangleq \mathtt{ld} & \llparenthesis \mathtt{W}^{\mathtt{na}} \rrparenthesis & \triangleq \mathtt{st} \\
\llparenthesis \mathtt{R}^{\mathtt{rlx}} \rrparenthesis & \triangleq \mathtt{ld;cmp;bc} & \llparenthesis \mathtt{W}^{\mathtt{rlx}} \rrparenthesis & \triangleq \mathtt{st} \\
\llparenthesis \mathtt{R}^{\mathtt{acq}} \rrparenthesis & \triangleq \mathtt{ld;cmp;bc;isync} & \llparenthesis \mathtt{W}^{\mathtt{rel}} \rrparenthesis & \triangleq \mathtt{lwsync;st} \\
\llparenthesis \mathtt{F}^{\neq\mathtt{sc}} \rrparenthesis & \triangleq \mathtt{lwsync} & \llparenthesis \mathtt{F}^{\mathtt{sc}} \rrparenthesis & \triangleq \mathtt{sync} \\
\llparenthesis \mathtt{RMW}^{\mathtt{rlx}} \rrparenthesis & \multicolumn{3}{l}{\triangleq \mathsf{L}{:}\mathtt{lwarx;cmp;bc}\ \mathsf{Le};\mathtt{stwcx.;bc}\ \mathsf{L};\mathsf{Le}{:}} \\
\llparenthesis \mathtt{RMW}^{\mathtt{acq}} \rrparenthesis & \multicolumn{3}{l}{\triangleq \llparenthesis \mathtt{RMW}^{\mathtt{rlx}} \rrparenthesis ;\mathtt{isync}} \\
\llparenthesis \mathtt{RMW}^{\mathtt{rel}} \rrparenthesis & \multicolumn{3}{l}{\triangleq \mathtt{lwsync};\llparenthesis \mathtt{RMW}^{\mathtt{rlx}} \rrparenthesis} \\
\llparenthesis \mathtt{RMW}^{\mathtt{acqrel}} \rrparenthesis & \multicolumn{3}{l}{\triangleq \mathtt{lwsync};\llparenthesis \mathtt{RMW}^{\mathtt{rlx}} \rrparenthesis ;\mathtt{isync}}
\end{array}$$

Figure 9. Compilation of non-SC primitives to Power.

Leading $\mathtt{sync}$		Trailing $\mathtt{sync}$	
$\llparenthesis \mathtt{R}^{\mathtt{sc}} \rrparenthesis$	$\triangleq \mathtt{sync};\llparenthesis \mathtt{R}^{\mathtt{acq}} \rrparenthesis$	$\llparenthesis \mathtt{R}^{\mathtt{sc}} \rrparenthesis$	$\triangleq \mathtt{ld;sync}$
$\llparenthesis \mathtt{W}^{\mathtt{sc}} \rrparenthesis$	$\triangleq \mathtt{sync;st}$	$\llparenthesis \mathtt{W}^{\mathtt{sc}} \rrparenthesis$	$\triangleq \llparenthesis \mathtt{W}^{\mathtt{rel}} \rrparenthesis;\mathtt{sync}$
$\llparenthesis \mathtt{RMW}^{\mathtt{sc}} \rrparenthesis$	$\triangleq \mathtt{sync};\llparenthesis \mathtt{RMW}^{\mathtt{acq}} \rrparenthesis$	$\llparenthesis \mathtt{RMW}^{\mathtt{sc}} \rrparenthesis$	$\triangleq \llparenthesis \mathtt{RMW}^{\mathtt{rel}} \rrparenthesis;\mathtt{sync}$

Figure 10. Compilations of SC accesses to Power.

As already mentioned, the two compilation schemes from C11 to Power that have been proposed in the literature [1] differ only in the mappings used for SC accesses (see Fig. 10). The first scheme follows the *leading sync* convention, and places a $\mathtt{sync}$ fence *before* each SC access. The alternative scheme follows the *trailing sync* convention, and places a $\mathtt{sync}$ fence *after* each SC access. Importantly, the same scheme should be used for all SC accesses in the program, since mixing the schemes is unsound. The mappings for the non-SC accesses and fences are common to both schemes and are shown in Fig. 9. Note that our compilation of relaxed reads is stronger than the one proposed for C11 (see §2.3).

Our main theorem says that the compilation schemes are correct.

Theorem 2. *For a program P, denote by $\llparenthesis P \rrparenthesis$ the* Power *program obtained by compiling P using the scheme in Fig. 9 and either of the schemes in Fig. 10 for SC accesses. Then, given a program P, every outcome of $\llparenthesis P \rrparenthesis$ under* Power *is an outcome of P under* RC11.

Proof (Outline). The main idea is to consider the compilation as if it happens in three steps, and prove the soundness of each step:

1. Leading $\mathtt{sync}$: Each $\mathtt{R}^{\mathtt{sc}}/\mathtt{W}^{\mathtt{sc}}/\mathtt{RMW}^{\mathtt{sc}}$ in P is replaced by $\mathtt{F}^{\mathtt{sc}}$ followed by $\mathtt{R}^{\mathtt{acq}}/\mathtt{W}^{\mathtt{rel}}/\mathtt{RMW}^{\mathtt{acqrel}}$.
   Trailing $\mathtt{sync}$: Each $\mathtt{R}^{\mathtt{sc}}/\mathtt{W}^{\mathtt{sc}}/\mathtt{RMW}^{\mathtt{sc}}$ in P is replaced by $\mathtt{R}^{\mathtt{acq}}/\mathtt{W}^{\mathtt{rel}}/\mathtt{RMW}^{\mathtt{acqrel}}$ followed by $\mathtt{F}^{\mathtt{sc}}$.
2. The mappings in Fig. 9 are applied.
3. Leading $\mathtt{sync}$: Pairs of the form $\mathtt{sync;lwsync}$ that originated from $\mathtt{W}^{\mathtt{sc}}/\mathtt{RMW}^{\mathtt{sc}}$ are reduced to $\mathtt{sync}$ (eliminating the redundant $\mathtt{lwsync}$).
   Trailing $\mathtt{sync}$: Any $\mathtt{cmp;bc;isync;sync}$ sequences originated from $\mathtt{R}^{\mathtt{sc}}/\mathtt{RMW}^{\mathtt{sc}}$ are reduced to $\mathtt{sync}$ (eliminating the redundant $\mathtt{cmp;bc;isync}$).

The resulting Power program is clearly identical to the one obtained by applying the mappings in Figures 9 and 10.

The soundness for each step (that is, none of them introduces additional outcomes) is established in [3]. □

X \ Y	$\mathtt{R}_y^{o_2}$	$\mathtt{W}_y^{o_2}$	$\mathtt{RMW}_y^{o_2}$	$\mathtt{F}^{o_2}$
$\mathtt{R}_x^{o_1}$	$o_1 \sqsubseteq \mathtt{rlx}$	$o_1, o_2 \sqsubseteq \mathtt{rlx} \land (o_1 = \mathtt{na} \lor o_2 = \mathtt{na})$	$o_1 = \mathtt{na} \land o_2 \sqsubseteq \mathtt{acq}$	$o_1 \neq \mathtt{rlx} \land o_2 = \mathtt{acq}$
$\mathtt{W}_x^{o_1}$	$o_1 \neq \mathtt{sc} \lor o_2 \neq \mathtt{sc}$	$o_2 \sqsubseteq \mathtt{rlx}$	$o_2 \sqsubseteq \mathtt{acq}$	$o_2 = \mathtt{acq}$
$\mathtt{RMW}_x^{o_1}$	$o_1 \sqsubseteq \mathtt{rel}$	$o_1 \sqsubseteq \mathtt{rel} \land o_2 = \mathtt{na}$	$-$	$o_1 \sqsupseteq \mathtt{acq} \land o_2 = \mathtt{acq}$
$\mathtt{F}^{o_1}$	$o_1 = \mathtt{rel}$	$o_1 = \mathtt{rel} \land o_2 \neq \mathtt{rlx}$	$o_1 = \mathtt{rel} \land o_2 \sqsupseteq \mathtt{rel}$	$o_1 = \mathtt{rel} \land o_2 = \mathtt{acq}$

Table 1. Deorderable pairs of accesses/fences (x and y are distinct locations).

The main difficulty (and novelty of our proof) lies in proving soundness of the second step, and more specifically in establishing the NO-THIN-AIR condition. Since Power, unlike RC11, does not generally forbid $\mathtt{sb} \cup \mathtt{rf}$ cycles, we have to show that such cycles can be untangled to produce a racy RC11-consistent execution, witnessing the undefined behavior. Here, the idea is, similar to DRF-SC proofs, to detect a first $\mathtt{rf}$ edge that closes an $\mathtt{sb} \cup \mathtt{rf}$ cycle, and replace it by a different $\mathtt{rf}$ edge that avoids the cycle. This is highly non-trivial because it is unclear how to define a "first" $\mathtt{rf}$ edge when $\mathtt{sb} \cup \mathtt{rf}$ is cyclic. To solve this problem, we came up with a different ordering of events, which does not include all $\mathtt{sb}$ edges, and Power ensures to be acyclic (a relation we call *Power-before* in [3]).

For completeness, we also show that the conditional branch after the relaxed read is only necessary if we care about enforcing the NO-THIN-AIR condition. That is, let weakRC11 be the model obtained from RC11 by omitting the NO-THIN-AIR condition, and denote by $(\!|P|\!)_{\mathsf{weak}}$ the Power program obtained by compiling P as above, except that relaxed reads are compiled to plain loads (again, with either leading or trailing syncs for SC accesses). Then, this scheme is correct with respect to the weakRC11 model.

Theorem 3 (Compilation of weakRC11 to Power)**.** *Given a program P, every outcome of $(\!|P|\!)_{\mathsf{weak}}$ under* Power *is an outcome of P under* weakRC11.

Finally, we note that it is also possible to use a lightweight fence (lwsync) instead of a fake control dependency and an instruction fence (isync) in the compilation of (all or some) acquire accesses.

6. Compilation to ARMv7

The ARMv7 model [4] is very similar to the Power model just presented in §5. There are only two differences.

First, while ARMv7 has analogues for Power's strong fence and instruction fence (dmb for sync, and isb for isync), it lacks an analogue for Power's lightweight fence (lwsync). Thus, on ARMv7 we have $\mathtt{F}^{\mathtt{lwsync}} = \emptyset$ and so $\mathtt{fence} = \mathtt{sync}$.

The second difference is that ARMv7 has a somewhat weaker *preserved program order*, ppo, than Power, which in particular does not always include $[\mathtt{R}]; \mathtt{sb}|_{\mathtt{loc}}; [\mathtt{W}]$ (following the model in [4]). In our Power compilation proofs, however, we never rely on this property of Power's ppo (see [3]).

The compilation schemes to ARMv7 are essentially the same as those to Power substituting the corresponding ARMv7 instructions for the Power ones: dmb instead of sync and lwsync, and isb instead of isync. The soundness of compilation to ARMv7 follows directly from Theorems 2 and 3.

We note that neither GCC (version 5.4) nor LLVM (version 3.9) map acquire reads into ld;cmp;bc;isb. Instead, they emit ld;dmb (that corresponds to Power's ld;sync). With this stronger compilation scheme, there is no correctness problem in compilation of C11 to ARMv7. Nevertheless, if one intends to use isb's, the same correctness issue arises (*e.g.,* the one in Fig. 1), and RC11 overcomes this issue.

7. Correctness of Program Transformations

In this section, we list program transformations that are sound in RC11, and prove that this is the case. As in [30], to have a simple presentation, all of our arguments are performed at the *semantic* level, as if the transformations were applied to events in an execution. Thus, to prove soundness of a program transformation $P_{\mathsf{src}} \rightsquigarrow P_{\mathsf{tgt}}$, we are given an arbitrary RC11-consistent execution G_{tgt} of P_{tgt}, and construct a RC11-consistent execution G_{src} of P_{src}, such that either G_{src} and G_{tgt} have the same outcome or G_{src} is racy. In the former case, we show that G_{tgt} is racy only if G_{src} is. Consequently, one obtains that every outcome of P_{tgt} under RC11 is also an outcome of P_{src} under RC11.

The soundness proofs (sketched in [3]) are mostly similar to the proofs in [30], with the main difference concerning the new SC condition.

Strengthening Strengthening transforms the mode o of an event in the source into o' in the target where $o \sqsubseteq o'$. Soundness of this transformation is trivial, because RC11-consistency is monotone with respect to the mode ordering.

Sequentialization Sequentialization merges two program threads into one, by interleaving their events in $\mathtt{sb}$. Essentially sequentialization just adds edges to the $\mathtt{sb}$ relation. Its soundness trivially follows from the monotonicity of RC11-consistency with respect to $\mathtt{sb}$.

Deordering Table 1 defines the *deorderable* pairs, for which we proved the soundness of the transformation $\mathtt{X};\mathtt{Y} \rightsquigarrow \mathtt{X} \parallel \mathtt{Y}$ in RC11. (Note that reordering is obtained by applying deordering and sequentialization.) Generally speaking, RC11 supports all reorderings that are intended

$$\begin{array}{lll@{\qquad\qquad}lll}
\mathtt{R}^{o};\mathtt{R}^{o} & \rightsquigarrow & \mathtt{R}^{o} & \mathtt{W}^{o};\mathtt{W}^{o} & \rightsquigarrow & \mathtt{W}^{o} \\
\mathtt{W}^{\mathtt{sc}};\mathtt{R}^{\mathtt{sc}} & \rightsquigarrow & \mathtt{W}^{\mathtt{sc}} & \mathtt{W}^{o};\mathtt{R}^{\mathtt{acq}} & \rightsquigarrow & \mathtt{W}^{o} \\
\mathtt{RMW}^{o};\mathtt{R}^{o_{\mathrm{r}}} & \rightsquigarrow & \mathtt{RMW}^{o} & \mathtt{RMW}^{o};\mathtt{RMW}^{o} & \rightsquigarrow & \mathtt{RMW}^{o} \\
\mathtt{W}^{o_{\mathrm{w}}};\mathtt{RMW}^{o} & \rightsquigarrow & \mathtt{W}^{o_{\mathrm{w}}} & \mathtt{F}^{o};\mathtt{F}^{o} & \rightsquigarrow & \mathtt{F}^{o}
\end{array}$$

Figure 11. Mergeable pairs (assuming both accesses are to the same location). o_{r} denotes the maximal mode in $\{\mathtt{rlx}, \mathtt{acq}, \mathtt{sc}\}$ satisfying $o_{\mathrm{r}} \sqsubseteq o$; and o_{w} denotes the maximal mode in $\{\mathtt{rlx}, \mathtt{rel}, \mathtt{sc}\}$ satisfying $o_{\mathrm{w}} \sqsubseteq o$.

to be sound in C11 [30], except for load-store reorderings of relaxed accesses, which are unsound in RC11 due to the conservative NO-THIN-AIR condition (if one omits this condition, these reorderings are sound). Importantly, load-store reorderings of *non-atomic* accesses are sound due to the "catch-fire" semantics. The soundness of these reorderings (in the presence of NO-THIN-AIR) was left open in [30], and requires a non-trivial argument of the same nature as the one used to show NO-THIN-AIR in the compilation correctness proof.

Merging Merges are transformations of the form $\mathtt{X};\mathtt{Y} \rightsquigarrow \mathtt{Z}$, eliminating one memory access or fence. Fig. 11 defines the set of *mergeable* pairs. Note that using strengthening, the modes mentioned in Fig. 11 are upper bounds (*e.g.,* $\mathtt{R}_x^{\mathtt{acq}};\mathtt{R}_x^{\mathtt{rlx}}$ can be first strengthened to $\mathtt{R}_x^{\mathtt{acq}};\mathtt{R}_x^{\mathtt{acq}}$ and then merged). Generally speaking, RC11 supports all mergings that are intended to be mergeable in C11 [30].

Remark 6. The elimination of redundant read-after-write allows the write to be non-atomic. Nevertheless, an SC read cannot be eliminated in this case, unless it follows an SC write. Indeed, eliminating an SC read after a non-SC write is unsound in RC11. We note that the effectiveness of this optimization seems to be low, and, in fact, it is already unsound for the model in [5] (see [3] for a counterexample). Note also that read-after-RMW elimination does not allow the read to be an acquire read unless the update includes an acquire read (unlike read-after-write). This is due to release sequences: eliminating an acquire read after a relaxed update may remove the synchronization due to a release sequence ending in this update.

Register Promotion Finally, "register promotion" is sound in RC11. This global program transformation replaces all the accesses to a memory location by those to a register, provided that the location is used by only one thread. At the execution level, all accesses to a particular location are removed from the execution, provided that they are all $\mathtt{sb}$-related.

8. Programming Guarantees

In this section, we demonstrate that our semantics for SC atomics (*i.e.,* the SC condition in Def. 1) is not overly weak. We do so by proving theorems stating that programmers who follow certain defensive programming patterns can be assured that their programs exhibit no weak behaviors. The first such theorem is DRF-SC, which says that if a program has no races on non-SC accesses under SC semantics, then its outcomes under RC11 coincide with those under SC.

In our proofs we use the standard declarative definition of SC: an execution is SC-consistent if it is complete, satisfies ATOMICITY, and $\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic [28].

Theorem 4. *If in all SC-consistent executions of a program P, every race $\langle a, b\rangle$ has $\mathtt{mod}(a) = \mathtt{mod}(b) = \mathtt{sc}$, then the outcomes of P under* RC11 *coincide with those under SC.*

Note that the NO-THIN-AIR condition is essential for the correctness of Thm. 4 (recall the LB+deps example).

Next, we show that adding a fence instruction between every two accesses to *shared* locations restores SC, or there remains a race in the program, in which case the program has undefined behavior.

Definition 7. A location x is *shared* in an execution G if $\langle a, b\rangle \notin \mathtt{sb} \cup \mathtt{sb}^{-1}$ for some distinct events $a, b \in \mathtt{E}_x$.

Theorem 5. *Let G be an* RC11*-consistent execution. Suppose that for every two distinct shared locations x and y, $[\mathtt{E}_x];\mathtt{sb};[\mathtt{E}_y] \subseteq \mathtt{sb};[\mathtt{F}^{\mathtt{sc}}];\mathtt{sb}$. Then, G is SC-consistent.*

We remark that for the proofs of Theorems 4 and 5, we do not need the full SC condition: for Thm. 4 it suffices for $[\mathtt{E}^{\mathtt{sc}}];(\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb});[\mathtt{E}^{\mathtt{sc}}]$ to be acyclic; and for Thm. 5 it suffices for $[\mathtt{F}^{\mathtt{sc}}];\mathtt{sb};\mathtt{eco};\mathtt{sb};[\mathtt{F}^{\mathtt{sc}}]$ to be acyclic.

9. Related Work

The C11 memory model was designed by the C++ standards committee based on a paper by Boehm and Adve [10]. During the standardization process, Batty *et al.* [8] formalized the C11 model and proved soundness of its compilation to x86-TSO. They also proposed a number of key technical improvements to the model (such as some coherence axioms), which were incorporated into the standard.

Since then, however, a number of problems have been found with the C11 model. In 2012, Batty *et al.* [7] and Sarkar *et al.* [27] studied the compilation of C11 to Power, and incorrectly proved the correctness of two compilation schemes. In their proofs, from a consistent Power execution, they constructed a corresponding C11 execution, which they tried to prove consistent, but in doing so they forgot to check the overly strong condition S1. The examples shown in §1 and in §2.1 are counterexamples to their theorems.

Quite early on, a number of papers [12, 31, 24, 11] noticed the disastrous effects of thin-air behaviors allowed by the C11 model, and proposed strengthening the definition of consistency by disallowing $\mathtt{sb} \cup \mathtt{rf}$ cycles. Boehm and Demsky [11] further discussed how the compilation schemes of relaxed accesses to Power and ARM would be affected by the change, but did not formally prove the correctness of their proposed schemes.

Next, Vafeiadis *et al.* [30] noticed a number of other problems with the C11 memory model, which invalidated a number of source-to-source program transformations that were assumed to hold. They proposed local fixes to those problems, and showed that these fixes enabled proving correctness of a number of local transformations. We have incorporated their fixes in the RC11-consistency definition.

Then, in 2016, Batty *et al.* [5] proposed a more concise semantics for SC atomics, whose presentation we have followed in our proposed RC11 model. As their semantics is stronger than C11, it cannot be compiled efficiently to Power, contradicting the claim of that paper. Moreover, as already discussed, SC fences are still too weak according to their model: in particular, putting them between every two accesses in a program with only atomic accesses does not guarantee SC.

Recently, Manerkar *et al.* [22] discovered the problem with trailing-sync compilation to Power (in particular, they observed the IRIW-acq-sc counterexample), and identified the mistake in the existing proof. Independently, we discovered the same problem, as well as the problem with leading-sync compilation. Moreover, in this paper, we have proposed a fix for both problems, and proven that it works.

A number of previous papers [31, 29, 18, 17] have studied only small fragments of the C11 model—typically the release/acquire fragment. Among these, Lahav *et al.* [17] proposed strengthening the semantics of SC fences in a different way from the way we do here, by treating them as read-modify-writes to a distinguished location. That strengthening, however, was considered in the restricted setting of only release/acquire accesses, and does not directly scale to the full set of C11 access modes. In fact, for the fragment containing only SC fences and release/acquire accesses, RC11-consistency is equivalent to RA-consistency that treats SC fences as RMWs to a distinguished location [17].

Finally, several solutions to the "out-of-thin-air" problem were recently suggested, *e.g.,* [26, 15, 16]. These solutions aim to avoid the performance cost of disallowing $\mathtt{sb} \cup \mathtt{rf}$ cycles, but none of them follows the declarative framework of C11. The conservative approach of disallowing $\mathtt{sb} \cup \mathtt{rf}$ cycles allows us to formulate our model in the style of C11.

10. Conclusion

In this paper, we have introduced the RC11 memory model, which corrects all the known problems of the C11 model (albeit at a performance cost for the "out-of-thin-air" problem). We have further proved (i) the correctness of compilation from RC11 to x86-TSO [25], Power and ARMv7 [4]; (ii) the soundness of various program transformations; (iii) a DRF-SC theorem; and (iv) a theorem showing that for programs without non-atomic accesses, weak behaviors can be always avoided by placing SC fences. It would be very useful to mechanize the proofs of this paper in a theorem prover; we leave this for future work.

A certain degree of freedom exists in the design of the SC condition. A very weak version, which maintains the two formal programming guarantees of this paper, would require acyclicity of $[\mathtt{E}^{\mathtt{sc}}]; (\mathtt{sb} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}); [\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{sb}; \mathtt{eco}; \mathtt{sb}; [\mathtt{F}^{\mathtt{sc}}]$. At the other extreme, one can require the acyclicity of $\mathtt{psc}_{\text{strong}} = ([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{hb}^?); (\mathtt{hb} \cup \mathtt{eco}); ([\mathtt{E}^{\mathtt{sc}}] \cup \mathtt{hb}^?; [\mathtt{F}^{\mathtt{sc}}])$, and either disallow mixing SC and non-SC accesses to the same location, or have rather expensive compilation schemes (for Power/ARMv7: compile release-acquire atomics exactly as the SC ones; for TSO: place a barrier before every SC read). Our choice of $\mathtt{psc}$ achieves the following: (i) it allows free mixing of different access modes to the same location in the spirit of C11; (ii) it ensures the correctness of the existing compilation schemes; and (iii) it coincides with $\mathtt{psc}_{\text{strong}}$ in the absence of mixing of SC and non-SC accesses to the same location.

Regarding the infamous "out-of-thin-air" problem, we employed in RC11 a conservative solution at the cost of including a fake control dependency after every relaxed read. While this was already considered a valid solution before, we are the first to prove the correctness of this compilation scheme, as well as the soundness of reordering of independent non-atomic accesses under this model. Correctness of an alternative scheme that places a lightweight fence after every relaxed write is left for future work. It would be interesting to evaluate the practical performance costs of each scheme. On the one hand, relaxed writes (which are not followed by a fence) are perhaps rare in real programs, compared to relaxed reads. On the other hand, a control dependency is cheaper than a lightweight fence, and relaxed reads are often anyway followed by a control dependency.

Another important future direction would be to combine our SC constraint with our recent operational model in [16], which prevents "out-of-thin-air" values (and avoids undefined behaviors altogether), while still allowing the compilation of relaxed reads and writes to plain loads and stores. This is, in particular, crucial for adopting a model like RC11 in a type-safe language, like Java, which cannot allow undefined behaviors. Integrating our SC condition in that model, however, is non-trivial because the model is defined in a very different style from C11, and thus we will have to find an equivalent operational way to check our SC condition.

Finally, extending RC11 with additional features of C11 (see §3.4) and establishing the correctness of compilation of RC11 to ARMv8 [13] are important future goals as well.

Acknowledgments

We thank Hans Boehm, Soham Chakraborty, Doug Lea, Peter Sewell and the PLDI'17 reviewers for their helpful feedback. This research was supported in part by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-07, and in part by an ERC Consolidator Grant for the project "RustBelt" (grant agreement no. 683289). The third author has been supported by a Korea Foundation for Advanced Studies Scholarship.

References

[1] C/C++11 mappings to processors, available at `http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`. [Online; accessed 27-September-2016].

[2] Crossbeam: support for concurrent and parallel programming, available at `https://github.com/aturon/crossbeam`. [Online; accessed 24-October-2016].

[3] Supplementary material for this paper, available at `http://plv.mpi-sws.org/scfix/`.

[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] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL 2016*, pages 634–648. ACM, 2016.

[6] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *ESOP 2015*, pages 283–307. Springer, 2015.

[7] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *POPL 2012*, pages 509–520. ACM, 2012.

[8] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL 2011*, pages 55–66. ACM, 2011.

[9] H.-J. Boehm. Can seqlocks get along with programming language memory models? In *MSPC 2012*, pages 12–20. ACM, 2012.

[10] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI 2008*, pages 68–78. ACM, 2008.

[11] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC 2014*, pages 7:1–7:6. ACM, 2014.

[12] M. Dodds, M. Batty, and A. Gotsman. C/C++ causal cycles confound compositionality. *TinyToCS*, 2, 2013.

[13] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL 2016*, pages 608–621. ACM, 2016.

[14] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. `http://download.intel.com/design/Itanium/Downloads/25142901.pdf`. [Online; accessed 14-November-2016].

[15] A. Jeffrey and J. Riely. On thin air reads: Towards an event structures model of relaxed memory. In *LICS 2016*, pages 759–767. ACM, 2016.

[16] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL 2017*, pages 175–189. ACM, 2017.

[17] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *POPL 2016*, pages 649–662. ACM, 2016.

[18] O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In *ICALP 2015*, pages 311–323. Springer, 2015.

[19] O. Lahav and V. Vafeiadis. Explaining relaxed memory models with program transformations. In *FM 2016*, pages 479–495. Springer, 2016.

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

[21] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *PPoPP 2013*, pages 69–80. ACM, 2013.

[22] Y. A. Manerkar, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *arXiv preprint arXiv:1611.01507*, 2016.

[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] B. Norris and B. Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA 2013*, pages 131–150. ACM, 2013.

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

[26] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, pages 622–633. ACM, 2016.

[27] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI 2012*, pages 311–322. ACM, 2012.

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

[29] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA 2014*, pages 691–707. ACM, 2014.

[30] 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 *POPL 2015*, pages 209–220. ACM, 2015.

[31] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA 2013*, pages 867–884. ACM, 2013.

[32] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In *POPL 2017*, pages 190–204. ACM, 2017.