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.
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]:

\[
\begin{align*}
  x &:= \text{sc} 1 \\
  a &:= x_{\text{acq}} #1 \\
  c &:= y_{\text{acq}} #0 \\
  b &:= y_{\text{acq}} #1 \\
  d &:= x_{\text{acq}} #0 \\
  y &:= \text{sc} 1
\end{align*}
\]

(IRIW-acq-sc)

Here and in all other programs in this paper, we write \(a, b, \ldots\) 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. 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).
As an example, in Fig. 1, we depict an execution of IR IW-acq-sc yielding the result 
\( a = b = 1 \land c = d = 0 \).

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, \( E \), are called events and represent the individual memory accesses and fences of the program, while its edges represent various relations among these events:

- The \textit{sequenced-before} (\texttt{sb}) relation, a.k.a. program order, captures the order of events in the program’s control flow.
- The \textit{reads-from} (\texttt{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 \textit{modification order} (\texttt{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 IR IW-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 \texttt{sb}-before all other events.

**Notation 1.** Given a binary relation \( R \), we write \( R^2, R^+ \), and \( R^\ast \) 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 \textit{synchronizes with} (\texttt{sw}) the read.\(^2\) Next, we say that one event \textit{happens before} (\texttt{hb}) another event if they are connected by a sequence of \texttt{sb} or \texttt{sw} edges. Formally, \( \texttt{hb} \triangleq (\texttt{sb} \cup \texttt{sw})^\ast \). 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 \texttt{mo}-before another write \( f \) to the same location, we say that \( e \) \textit{reads-before} (\texttt{rb}) \( f \) (this relation is also called “from-read” [4], but we find \texttt{reads-before} more intuitive). Formally, \( \texttt{rb} \triangleq \texttt{rf}^{-1}; \texttt{mo} \setminus [E] \). The “\( \setminus [E] \)” 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 \( (m, p) \in \texttt{rb} \) and \( (o, k) \in \texttt{rb} \).

Consistent C11 executions require that \texttt{hb} is irreflexive (equivalently, \( \texttt{sb} \cup \texttt{sw} \) 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 \texttt{mo} does not contradict \texttt{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 \texttt{hb}-related reads from the same location cannot read from two writes in reversed \texttt{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, \( S \), 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 \( E^{sc} \) denotes the set of all SC events in \( E \):

- \((S1)\) \( S \) must include \texttt{hb} restricted to SC events (formally: \( [E^{sc}]; \texttt{hb}; [E^{sc}] \subseteq S \));
- \((S2)\) \( S \) must include \texttt{mo} restricted to SC events (formally: \( [E^{sc}]; \texttt{mo}; [E^{sc}] \subseteq S \));
- \((S3)\) \( S \) must include \texttt{rb} restricted to SC events (formally: \( [E^{sc}]; \texttt{rb}; [E^{sc}] \subseteq S \));
- \((S4-7)\) \( S \) must obey a few more conditions having to do with SC fences.

**Remark 1.** The \( S \) condition above, due to Batty \textit{et al.} [5], is slightly simpler and stronger than the one imposed by the official C11. Crucially, however, \textbf{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 \textit{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 \( S \) corresponds to the order in which SC events are executed, it should agree with the other global orders of events: \texttt{hb}, \texttt{mo}, and \texttt{rb}. However, as we will see shortly, condition \((S1)\) is too strong. Before we get there, let us first

---

\(^2\) The actual definition of \texttt{sw} contains further cases, which are not relevant for the current discussion. These are included in our formal model in §3.
look at a few examples to illustrate how the conditions on $S$ interact to enforce sequential consistency.

Consider the classic “store buffering” litmus test:

\[
\begin{align*}
x &:= \text{ac} 1 \\
y &:= \text{ac} 1 \\
a &:= y_{\text{ac}} // 0 \\
b &:= x_{\text{ac}} // 0
\end{align*}
\] (SB)

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

\[
\begin{align*}
x &:= \text{ac} 1 \\
y &:= \text{ac} 1 \\
a &:= y_{\text{ac}} // 1 \\
b &:= x_{\text{ac}} // 1
\end{align*}
\] (2+2W)

To see this, consider the second execution graph in Fig. 2, which has the outcome $a = b = 1$: the rf and mo edges are forced because of the values read and the coherenceWW. Now, $S(k, l)$ and $S(n, o)$ hold because of condition S1; while $S(l, n)$ and $S(o, k)$ hold because of condition S2. Again, this entails a cycle in 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 $S(p, k)$ in Fig. 1. We also know that $S(k, m)$ because of happens-before via $l$ (S1). Thus, by transitivity, $S(p, m)$. However, if the second thread reads $y = 0$, then $m$ reads-before $p$, in which case $S(m, p)$ (S3), and 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:

\[
\begin{align*}
x &:= \text{ac} 1 \\
y &:= \text{rel} 1 \\
b &:= \text{FAI}(y)_{\text{sc}} // l \\
c &:= y_{\text{aix}} // 3 \\
y &:= \text{ac} 3 \\
a &:= x_{\text{ac}} // 0
\end{align*}
\] (Z6.U)

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 rf and mo edges are forced: even if all accesses in the program were relaxed atomic, they would have to go this way. $S(k, m)$ holds because of condition S1 ($k$ happens-before $l$, which happens-before $m$); $S(m, o)$ holds because of condition S2 ($m$ precedes $o$ in modification order); $S(o, p)$ 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, $S(p, k)$, thus forming a cycle in 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 $a := x_{\text{ac}}$ load in the last thread, is equivalent to an $\text{aixsync}$ 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 $\text{hb}$-related accesses to different locations, there should be a Power sync fence occurring somewhere in the $\text{hb}$-path (the sequence of $\text{sb}$ and $\text{sw}$ 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 $\text{hb}$-related SC accesses. Specifically, if we follow the trailing sync convention, the $\text{hb}$-path (in Fig. 1) from $k$ to $m$ starting with an $\text{sw}$ edge avoids the sync fence placed after $k$.

Conversely, if we follow the leading sync convention, the $\text{hb}$-path (in Fig. 3) from $k$ to $m$ ending with an $\text{sw}$ edge avoids
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.

To see the unsoundness of eliminating an overwritten SC write, consider the following program. The annotated execution contains an SC before which occurs twice.

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.

### 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 S, they require a single acyclicity condition, namely that \([E^sc]; (hb \cup mo \cup rb); [E^sc]\) be acyclic. (In general, acyclicity of \(\cup R_i\) is equivalent to the existence of a total order that contains \(R_1, R_2, \ldots\)).

We propose to correct the condition by replacing \(hb\) with \(sb\cup sb; hb; sb\cup hb|loc\). Accordingly, we require that

\[
[E^sc]; (sb \cup sb; hb; sb \cup hb|loc \cup mo \cup rb); [E^sc]
\]

is acyclic. Note that this condition still ensures SC semantics for programs that have only SC accesses. Indeed, since \([E^sc]; rf; [E^sc] \subseteq [E^sc]; sw; [E^sc] \subseteq [E^sc]; hb|loc; [E^sc]\), our condition implies acyclicity of \([E^sc]; (sb \cup rf \cup mo \cup rb); [E^sc]\).

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

---

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.

### 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 S, they require a single acyclicity condition, namely that \([E^sc]; (hb \cup mo \cup rb); [E^sc]\) be acyclic. (In general, acyclicity of \(\cup R_i\) is equivalent to the existence of a total order that contains \(R_1, R_2, \ldots\)).

We propose to correct the condition by replacing \(hb\) with \(sb\cup sb; hb; sb\cup hb|loc\). Accordingly, we require that

\[
[E^sc]; (sb \cup sb; hb; sb \cup hb|loc \cup mo \cup rb); [E^sc]
\]

is acyclic. Note that this condition still ensures SC semantics for programs that have only SC accesses. Indeed, since \([E^sc]; rf; [E^sc] \subseteq [E^sc]; sw; [E^sc] \subseteq [E^sc]; hb|loc; [E^sc]\), our condition implies acyclicity of \([E^sc]; (sb \cup rf \cup mo \cup rb); [E^sc]\).

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.

To see the unsoundness of eliminating an overwritten SC write, consider the following program. The annotated execution contains an SC before which occurs twice.

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.

### 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 S, they require a single acyclicity condition, namely that \([E^sc]; (hb \cup mo \cup rb); [E^sc]\) be acyclic. (In general, acyclicity of \(\cup R_i\) is equivalent to the existence of a total order that contains \(R_1, R_2, \ldots\)).

We propose to correct the condition by replacing \(hb\) with \(sb\cup sb; hb; sb\cup hb|loc\). Accordingly, we require that

\[
[E^sc]; (sb \cup sb; hb; sb \cup hb|loc \cup mo \cup rb); [E^sc]
\]

is acyclic. Note that this condition still ensures SC semantics for programs that have only SC accesses. Indeed, since \([E^sc]; rf; [E^sc] \subseteq [E^sc]; sw; [E^sc] \subseteq [E^sc]; hb|loc; [E^sc]\), our condition implies acyclicity of \([E^sc]; (sb \cup rf \cup mo \cup rb); [E^sc]\).

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.

To see the unsoundness of eliminating an overwritten SC write, consider the following program. The annotated execution contains an SC before which occurs twice.

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.

### 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 S, they require a single acyclicity condition, namely that \([E^sc]; (hb \cup mo \cup rb); [E^sc]\) be acyclic. (In general, acyclicity of \(\cup R_i\) is equivalent to the existence of a total order that contains \(R_1, R_2, \ldots\)).

We propose to correct the condition by replacing \(hb\) with \(sb\cup sb; hb; sb\cup hb|loc\). Accordingly, we require that

\[
[E^sc]; (sb \cup sb; hb; sb \cup hb|loc \cup mo \cup rb); [E^sc]
\]

is acyclic. Note that this condition still ensures SC semantics for programs that have only SC accesses. Indeed, since \([E^sc]; rf; [E^sc] \subseteq [E^sc]; sw; [E^sc] \subseteq [E^sc]; hb|loc; [E^sc]\), our condition implies acyclicity of \([E^sc]; (sb \cup rf \cup mo \cup rb); [E^sc]\).

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.

To see the unsoundness of eliminating an overwritten SC write, consider the following program. The annotated execution contains an SC before which occurs twice.

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.

### 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 S, they require a single acyclicity condition, namely that \([E^sc]; (hb \cup mo \cup rb); [E^sc]\) be acyclic. (In general, acyclicity of \(\cup R_i\) is equivalent to the existence of a total order that contains \(R_1, R_2, \ldots\)).

We propose to correct the condition by replacing \(hb\) with \(sb\cup sb; hb; sb\cup hb|loc\). Accordingly, we require that

\[
[E^sc]; (sb \cup sb; hb; sb \cup hb|loc \cup mo \cup rb); [E^sc]
\]

is acyclic. Note that this condition still ensures SC semantics for programs that have only SC accesses. Indeed, since \([E^sc]; rf; [E^sc] \subseteq [E^sc]; sw; [E^sc] \subseteq [E^sc]; hb|loc; [E^sc]\), our condition implies acyclicity of \([E^sc]; (sb \cup rf \cup mo \cup rb); [E^sc]\).

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.

To see the unsoundness of eliminating an overwritten SC write, consider the following program. The annotated execution contains an SC before which occurs twice.

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.
would be a problem if C11 relaxed accesses were compiled
where in C11 does not restore sequential consistency, even if
There is, however, another way in which putting fences every-
This holds for hardware memory models, such as x86-TSO,
rt from

§

IRlw even without fences [14,

Fixing the Semantics of SC Fences

Fixing the Semantics of SC Fences  Analyzing the execution of RWC+syncs, we note that there is a sb; rb; rf; sb path from f2 to f1, but this path does not contribute to psc1.

Although both rb and rf edges contribute to psc1, their composition rb; rf does not.

To repair the model, we define the extended coherence order, eco \( \triangleq (rf \cup mo \cup rb)^\dagger \). This order includes the reads-from relation, rf, the modification order, mo, the reads-before relation, 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 psc1 \( \cup [F^\dagger]; sb; eco; sb; [F^\dagger] \) is acyclic.

This stronger condition rules out the weak behavior of RWC+syncs because there are sb; eco; 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 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-
atomical accesses, placing an SC fence between every two
accesses guarantees SC. Indeed, by the definition of SC, it
suffices to show that eco \( \cup sb \) is acyclic. Consider a eco \( \cup sb \)
cycle. Since eco and sb are irreflexive and transitive, the
cycle necessarily has the form \((eco; sb)^\dagger\). Thus, between
every two eco steps, there must be an SC fence. So in effect,
we have a cycle in eco; sb; [F^\dagger]; sb, which can be regrouped
to a cycle in [F^\dagger]; sb; eco; sb; [F^\dagger], which is forbidden by
our model.

Finally, one might further consider strengthening the model by including eco in sb (which is used to define psc1), 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.

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

In general, one might expect that inserting an SC fence be-
tween 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 ac-
cesses (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 every-
where in C11 does not restore sequential consistency, even if
all the accesses are atomic. Consider the following program:

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 psc1 edge is from f1 to f2. 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 stan-
dard 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].
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 \( sb; rb; sb \) path from \( f_1 \) to \( f_2 \), the only path from \( f_2 \) back to \( f_1 \) is \( sb; rb; sb; sw; sb \) (or, more generally, \( hb; rb; hb \)), and so the execution is allowed.

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

\[
psc_{base} \triangleq ([E^{sc}] \cup [F^{sc}]; hb) \cup scb \cup ([F^{sc}]; [hb])
\]

\[
psc_F \triangleq [F^{sc}]; (hb \cup hb; eco; hb); [F^{sc}]
\]

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

### 3.2 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 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 comes 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 \( sb \cup rf \) 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.

\[
a := x_{rlx} \# 1 \quad b := y_{rlx} \# 1 \quad x := x_{rlx} 1
\]

(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 \( sb \cup rf \) 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 \( sb \cup rf \) 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 \( R, \text{dom}(R) \) and \( \text{codom}(R) \) denote its domain and codomain. Given a function \( f, =_f \) denotes the set of \( f \)-equivalent pairs \( (=_f \triangleq \{ (a, b) \mid f(a) = f(b) \}) \), 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, \( R|_{\text{imm}} \) denotes the set of all immediate \( R \) edges, i.e., pairs \( (a, b) \in R \) such that for every \( c, (c, b) \in R \) implies \( (c, a) \in R \), and \( (a, c) \in R \) implies \( (b, c) \in R \).

We assume finite sets \( \text{Loc} \) and \( \text{Val} \) 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 \( \sqsubseteq \) as follows:

\[
na \rightarrow rlx \rightarrow rel \rightarrow acqrel \rightarrow sc
\]
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 \( E \subseteq \mathbb{N} \) containing a distinguished set \( E_0 = \{ a_0^x \mid x \in \text{Loc} \} \) of initialization events. We use \( a, b, \ldots \) as metavariables for events.

2. a function \( \text{lab} \) assigning a label to every event in \( E \). Labels are of one of the following forms:
   - \( R^c(x, v) \) where \( o \in \{ \text{na}, \text{rlx}, \text{acq}, \text{sc} \} \).
   - \( W^o(x, v) \) where \( o \in \{ \text{na}, \text{rlx}, \text{rel}, \text{sc} \} \).
   - \( F^x \) where \( o \in \{ \text{acq}, \text{rel}, \text{acqrel}, \text{sc} \} \).

   We assume that \( \text{lab}(a_0^x) = W^{\text{na}}(x, 0) \) for every \( a_0^x \in E_0 \).

   \( \text{lab} \) naturally induces the functions \( \text{typ}, \text{mod}, \text{loc}, \text{val}_r, \text{val}_w \), and \( \text{val}_x \) that return (when applicable) the type (\( R, W \) or \( F \)), mode, location, and read/written value of an event.

For \( T \subseteq \{ R, W, F \} \), \( T \) denotes the set \( \{ e \in E \mid \text{typ}(e) \subseteq T \} \). We also concatenate the event sets notations, use subscripts to denote the accessed location, and superscripts for modes (e.g., \( R_W = R \cup W \) and \( W^{\text{acq}} \) denotes all events \( a \in W \) with \( \text{loc}(a) = x \) and \( \text{mod}(a) \supseteq \text{rel} \)).

3. a strict partial order \( \text{sb} \subseteq E \times E \), called \textit{sequenced-before}, which orders the initialization events before all other events, i.e., \( E_0 \times (E \setminus E_0) \subseteq \text{sb} \).

4. a binary relation \( \text{rmw} \subseteq [R]; \text{sb}\{\text{imm}\} = \text{loc} \); \( [W] \), called \textit{read-modify-write pairs}, such that for every \( (a, b) \in \text{rmw} \), \( \langle \text{mod}(a), \text{mod}(b) \rangle \) is one of the following:
   - \( \langle \text{rlx}, \text{rlx} \rangle \) (\( \text{RMW}^{\text{rlx}} \))
   - \( \langle \text{acq}, \text{rlx} \rangle \) (\( \text{RMW}^{\text{acqrlx}} \))
   - \( \langle \text{rlx}, \text{rel} \rangle \) (\( \text{RMW}^{\text{rlxrel}} \))

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

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 \( \text{rmw} \) relation, whereas in C11 they are represented by single 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 \( \text{rf} \subseteq [W]; \text{sb}\{\text{imm}\} = \text{loc} \); \( [R] \), called \textit{reads-from}, satisfying (i) \( \text{val}_r(a) = \text{val}_w(b) \) for every \( (a, b) \in \text{rf} \), and (ii) \( a_1 = a_2 \) whenever \( (a_1, b), (a_2, b) \in \text{rf} \).

6. a strict partial order \( \text{mo} \) on \( W \), called \textit{modification order}, which is a disjoint union of relations \( \{ \text{mo}_x \}_x \in \text{Loc} \), such that each \( \text{mo}_x \) is a strict total order on \( W_x \).

In what follows, to resolve ambiguities, we may include a prefix “\( G \)” to refer to the components of an execution \( G \).

Exections 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 \( \| i \in \text{Tid} \| c_i \), where \( \text{Tid} \) 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 \( \text{rf} \) and \( \text{mo} \) are arbitrary). Note that the set of executions of a program \( P \) is taken to be \textit{prefix-closed}: an \( \text{sb} \)-prefix of an execution of \( P \) (which includes at least the initialization events) is also considered to be an execution of \( P \). By \textit{full} executions of \( P \), we refer to executions that represent traces generated by the whole program \( P \).

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, \( R \subseteq \text{dom}(\text{rf}) \)). We refer to such executions as \textit{complete}.

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

\[
\begin{align*}
\text{rb} & \triangleq \text{rf}^{-1}; \text{mo} & \quad & \text{(reads-before)} \hfill \\
\text{eco} & \triangleq \text{rf} \cup \text{mo} \cup \text{rb}^+ & \quad & \text{(extended coherence order)} \hfill \\
\text{rs} & \triangleq [W]; \text{sb}\{\text{loc}\}; [W^{\text{rlx}}]; (\text{rf}; \text{rmw})^* & \quad & \text{(release sequence)} \hfill \\
\text{sw} & \triangleq [E^{\text{rlx}}]; ([F]; \text{sb})^+; \text{rs}; \text{rf} & \quad & \text{(synchronizes with)} \hfill \\
\text{hb} & \triangleq (\text{sb} \cup \text{sw})^+ & \quad & \text{(happens-before)} \hfill \\
\end{align*}
\]

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

The other three relations, \( \text{rs}, \text{sw} \) and \( \text{hb} \), are taken from [30]. Intuitively, \( \text{hb} \) records when an event is globally perceived as occurring before another one. It is defined in terms of two more basic relations. First, the \textit{release sequence} (\( \text{rs} \)) 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 (aw) an acquire event b, whenever b (or, in
case b is a fence, some sb-prior read) reads from the release
sequence of a (or in case a is a fence, of some sb-later write).
Then, we say that an event a happens-before (hb) an event b
if there is a path from a to b consisting of sb and sw edges.
Finally, we define the SC-before relation, scb, and the
partial SC relations, pscb, and pscf, as follows:

\[ \text{psc}_{\text{base}} \triangleq (\text{F}^{\text{sc}} \cup \text{F}^{\text{rc}}; \text{hb}) \cup \text{scb}; (\text{F}^{\text{sc}} \cup \text{hb}, \text{sc}) \]

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

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

- \( \text{hb; eco} \) is irreflexive.
- \( \text{rmw} \cap (\text{rb; mo}) = \emptyset. \) (ATOMICITY)
- \( \text{psc} \) is irreflexive.
- \( \text{rb; mo} \) is irreflexive.

**COHERENCE** ensures that programs with one only shared
location are sequentially consistent, as at least two locations
are needed for a cycle in \( \text{sb; eco}. \) **ATOMICITY** ensures that
the read and the write comprising a RMW are adjacent in
\( \text{eco}; \) 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 \( \S 5. \)

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 ∈ E, w ∈ \{typ(a), typ(b)\}, a ≠ b, and
loc(a) = loc(b). A pair \( \langle a, b \rangle \) is called race in G (denoted
\( \langle a, b \rangle \in \text{race} \)) if a and b are conflicting events in G, and
\( \langle a, b \rangle \notin \text{hb; hb} \).

**Definition 3.** An execution G is called racy if there is some
\( \langle a, b \rangle \in \text{race} \) with \( \text{na} \in \{\text{mod}(a), \text{mod}(b)\}. \) 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 no-
maximal event in \( W. \) We say that O : Loc → 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 rf ⊆ 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, \S 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 atoms.

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

- hb is irreflexive.
- rf; hb is irreflexive.
- mo; rf; hb is irreflexive.
- mo; hb is irreflexive.
- mo; hb; rf is irreflexive.

**Proposition 2.** The SC condition is equivalent to requiring the existence of a total strict order \( S \) on \( E^{\text{sc}} \) such that \( S; \text{psc} \)
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 ∈ E \( \setminus \text{En}, \) if \( \text{mod}(a) = \text{sc} \) and \( \text{loc}(a) = \text{loc}(b) \) then \( \text{mod}(b) = \text{sc} \) then \( \text{E}^{\text{sc}}; \text{hb}; \text{E}^{\text{sc}} \subseteq \text{psc}^{+}. \))

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
Then, given a program \( P \) that is implemented in the \( \text{GCC} \) TSO, which uses barriers after SC writes (see Fig. 8), any execution of this program that yields the annotated behavior has a cycle in \( E^{sc} \). This entails that the original C11 model, as well as Batty et al.’s \( \text{RC11} \) model, are correctly compiled to x86-TSO. Additionally, the proof only assumes the existence of an \( \text{MFENCE} \) instruction 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 \( \text{MFENCE} \) after each store that originated from an SC write. An alternative correct compilation scheme may place \( \text{MFENCE} \) 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 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. \( \text{hb} \) is irreflexive.
2. \( \text{mo} ; \text{hb} \) is irreflexive.
3. \( \text{rb} ; \text{hb} \) is irreflexive.
4. \( \text{rb} ; \text{mo} \) is irreflexive.
5. \( \text{rb} ; \text{mo} ; \text{rf} ; \text{sb} \) is irreflexive (where \( \text{rf} = \text{rf} \setminus \text{sb} \)).
6. \( \text{rb} ; \text{mo} ; [\text{RMW} \cup \text{F}] ; \text{sb} \) is irreflexive.

Unlike \( \text{RC11} \), well-formed TSO programs do not require undefined behavior. Thus, a function \( O : \text{Loc} \rightarrow \text{Val} \) 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 \( \text{GCC} \) 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 \( \text{RC11} \).

**Theorem 1.** For a program \( P \), denote by \( \langle P \rangle \) the TSO program obtained by compiling \( P \) using the scheme in Fig. 8. Then, given a program \( P \), every outcome of \( \langle P \rangle \) under TSO is an outcome of \( P \) under \( \text{RC11} \).

**Proof (Outline).** We consider the compilation as 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 a TSO-consistent TSO execution \( G_t \) of \( \langle P \rangle \) (where \( P \) has no non-SC fences), there exists an \( \text{RC11} \)-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 \( \text{RC11} \) obtained by replacing the \( \text{scb} \) relation by \( \text{scb}' \triangleq \text{hb} \cup \text{mo} \cup \text{rb} \). 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 \( \text{MFENCE} \) 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 \( \text{MFENCE} \) after each store that originated from an SC write. An alternative correct compilation scheme may place \( \text{MFENCE} \) 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 \( \text{MFENCE} \) before SC reads can be shown to be sound even for a very strong SC condition that requires acyclicity of:

\[
\text{psc}_{\text{strong}} = ([E^{sc}] \cup [F^{ec}] ; \text{hb}^2) ; (\text{hb} \cup \text{eco}) ; ([E^{sc}] \cup \text{hb}^2) ; [F^{ec}]).
\]

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 \( \text{psc}_{\text{strong}} \), or even if one requires acyclicity of \( [E^{sc}] ; (\text{sb} \cup \text{eco}) ; [E^{sc}] \). To see this, consider the following variant of SB:

\[
\begin{align*}
x & := \text{rel} 1 \\
a & := x_{\text{sc}} #/1 \\
b & := y_{\text{sc}} #/0 \\
c & := y_{\text{sc}} #/1 \\
d & := x_{\text{sc}} #/0
\end{align*}
\]

(SB-rfis)

Any execution of this program that yields the annotated behavior has a cycle in \( [E^{sc}] ; (\text{sb} \cup \text{eco}) ; [E^{sc}] \) (we have \( \text{rb} ; \text{rf} \) both from \( R^{sc}(x, 0) \) to \( R^{sc}(x, 1) \), and from \( R^{sc}(y, 0) \) to \( R^{sc}(y, 1) \)). 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 ppo, which is a subset of sb guaranteed to be preserved. The exact definition of ppo is quite intricate, and is included in [3].
- Read/write labels have the form R(x, v) and 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 F_{lsync} and F_{sync} the set of all lightweight fence and full fence events in a Power execution. Power’s “instruction fence” (isync) is used to derive ppo but is not recorded in executions.

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

- sync ≜ [RW]; sb; [F_{sync}]; sb; [RW]  
- lwsync ≜ [RW]; sb; [F_{lsync}]; sb; [RW] \ (W \times R)  
- fence ≜ sync ∪ lwsync  
- hb ≜ ppo ∪ fence; rfe  

(Power’s happens-before)

- prop₁ ≜ [W]; rfe; fence; hb⁺; [W]  
- prop₂ ≜ (moe ∪ rbe)²; rfe; (fence; hb⁺)²; sync; hb⁺  

(propagation relation)

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

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

1. sb|loc ∪ rf ∪ rb ∪ mo is acyclic. (SC-PER-LOC)
2. rbe; prop; hb⁺ is irreflexive. (OBSERVATION)
3. mo ∪ prop is acyclic. (PROPAGATION)
4. rmw \ (rbe; moe) = ∅. (POWER-ATOMICITY)
5. hb is acyclic. (POWER-NO-THIN-AIR)

Remark 5. The model in [4] contains an additional constraint: mo \ [At]; sb; [At] should be acyclic (recall that At = dom(rmw) ∪ codom(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 : Loc → 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).

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 sync fence before each SC access. The alternative scheme follows the trailing sync convention, and places a 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 \{P\} 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 \{P\} 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 sync: Each R^{sc}/W^{sc}/RMW^{sc} in P is replaced by F^{sc} followed by R^{eq}/W^{rel}/RMW^{acqrel}.
2. Trailing sync: Each R^{sc}/W^{sc}/RMW^{sc} in P is replaced by R^{eq}/W^{rel}/RMW^{acqrel} followed by F^{sc}.
3. The mappings in Fig. 9 are applied.

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].
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 sb ∪ 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 rf edge that closes an sb ∪ rf cycle, and replace it by a different rf edge that avoids the cycle. This is highly non-trivial because it is unclear how to define a “first” rf edge when sb ∪ rf is cyclic. To solve this problem, we came up with a different ordering of events, which does not include all 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 \( \llbracket P \rrbracket_{\text{weak}} \) 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 \( \llbracket P \rrbracket_{\text{weak}} \) under Power is an outcome of \( P \) under weakRC11.

Finally, we note that it is also possible to use a lightweight fence (lsync) 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 (lsync). Thus, on ARMv7 we have \( F_{lsync} = 0 \) and so fence = sync.

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

### 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 \( P_{src} \leadsto P_{tgt} \), 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 also 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 sb. Essentially sequentialization just adds edges to the sb relation. Its soundness trivially follows from the monotonicity of RC11-consistency with respect to sb.

#### Deordering

Table 1 defines the deorderable pairs, for which we proved the soundness of the transformation \( X; Y \leadsto X \parallel Y \) in RC11. (Note that deordering is obtained by applying deordering and sequentialization.) Generally speaking, RC11 supports all reorderings that are intended...
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 \(X; Y \rightarrow Z\), 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., \(R_{x}; R_{acq} \Delta R_{lx}\) can be first strengthened to \(R_{acq} \Delta R_{x}\) 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 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 \(\text{sb} \cup \text{rf} \cup \text{mo} \cup \text{rb}\) is acyclic [28].

Theorem 4. If in all SC-consistent executions of a program \(P\), every race \((a, b)\) has \(\text{md}(a) = \text{md}(b) = \text{sc}\), 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 \((a, b) \notin \text{sb} \cup \text{sb}^{-1}\) for some distinct events \(a, b \in E_x\).

Theorem 5. Let \(G\) be an RC11-consistent execution. Suppose that for every two distinct shared locations \(x\) and \(y\), \([E_x]; \text{sb}; [E_y] \subseteq \text{sb}; [F^\text{rc}]; \text{sb}\). 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 \([F^\text{rc}]; (\text{sb} \cup \text{rf} \cup \text{mo} \cup \text{rb}); [E_x]\) to be acyclic; and for Thm. 5 it suffices for \([F^\text{rc}]; \text{sb}; [E^\text{rc}]; \text{sb}\) 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 x86-64-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 \(\text{sb} \cup \text{rf}\) 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 num-
We have further proved without non-atomic accesses, weak behaviors can always be avoided by placing SC fences. It would be very useful to avoid the performance cost for the “out-of-thin-air” problem. 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 before 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.

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 always be 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 \( [E^{sc}] : [sb \cup rf \cup mo \cup rb]; [E^{sc}] \cup [F^{rs}] ; sb; eco; sb; [F^{sc}] \). At the other extreme, one can require the acyclicity of \( psc_{strong} = ([E^{sc}] \cup [F^{rs}] : hb); (hb \cup eco); ([E^{rs}] \cup hb; [F^{ec}]) \), 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 \( psc \) 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 \( psc_{strong} \) 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 before 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


