A Critique of Sequential Consistency in C11

DUC THAN NGUYEN, University of Illinois at Chicago, USA

ABSTRACT

The C++ Standards Committee designed the C11 memory model. During the process of standardization, Batty et al. [2011] formalized the C11 model, demonstrated to x86-TSO that its compilation was sound, and proposed numerous significant technical improvements to the model, all of which were incorporated into the standard. Since then, nevertheless, several issues have been found with the semantics of sequentially consistent accesses to the C11 model. In this report, we discuss and evaluate a number of proposed solutions to the known problems with the sequential consistency in the C11 model. These approaches include both fixes and refinements.

Contents

PhD Qualifier Examination 2022. Duc Than Nguyen, University of Illinois at Chicago, dnguye96@uic.edu.

1 INTRODUCTION

A compiler's responsibility is to make the code as optimized as possible, while a programmer's goal is to fully comprehend what they have written. Simple compiler optimizations can generate unexpected behavior in a concurrent setting. It is the goal of the memory model, which lies at the core of the concurrent semantics of shared memory systems, to alleviate the tension, as mentioned earlier. The memory model specifies the set of permitted outputs of a program's read and write operations and constrains an implementation to have only such allowed executions. The C11 memory model defines the semantics of concurrent accesses in the C programming language and offers atomic variables, fences, and a set of memory ordering for atomic accesses and fences. In addition, C11 specifies a set of memory consistency constraints for the shared memory accesses and fences. Batty et al. [2011] first formalized the concurrency model in the C++ standard. Since its original formalization, the model has evolved through a variety of distinct representations and revisions (e.g., Vafeiadis et al. [2015], Batty et al. [2016], Lahav et al. [2017]). Several issues have been discovered with the semantics of sequentially consistent C11 model accesses. This study investigates known issues with the sequential consistency of the C11 model, as well as proposed fixes and revisions.

1.1 Primary papers.

This report's analysis of the addressed field must be based on at least three papers. The following is a list of the papers and the contributors' contributions.

(1) Vafeiadis et al. [2015] discovered a number of counterexamples to illustrate that the C11 memory model prohibits a number of common transformations. Then, they offered local C11 modifications. Their contributions are detailed in Section 3.
(2) Batty et al. [2016] introduced more concise semantics for SC atomics. Their work also helped lay down more straightforward foundations for reasoning about C11, which were used to guide our critique in Section 4.
(3) Lahav et al. [2017] addressed issues regarding SC semantics and proposed repairing C11 semantics (called RC11). They demonstrate that RC11 compiles on Power and ARMv7, which is valid. The suggested semantics also ensure that interleaving behavior is restored when SC fences are placed between every pair of shared memory accesses. We provide a critique of their proposal in Section 5.

2 BACKGROUND

This report assumes that the reader is familiar with programming languages and compiler design. We go over a few more ideas in the section below.

2.1 Sequential consistency

A simple model of behavior in concurrent settings is the model of sequential consistency (SC). This model presents a simple interleaving semantics of memory access, in which each thread in the system takes one step at a time, modifying a single global shared memory. Lamport [1979] describes a machine as sequentially consistent if "*... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*" Unfortunately, SC does not catch all the complications when executing concurrent programs on real-world hardware. To see the problem, consider the following example (known as store-buffering). In this program, x and y are shared variables, and they are set to 0, whereas a and b are local variables. The question is whether both loads can get the value 0, as suggested by the annotated behavior. In other words,

```
                    x := 0;   y := 0;
x := 1;                     ||  y := 1;
a := *y; //load 0           ||  b := *x; //load 0
```

at the end of the program's execution, can both thread-local variables a and b have the value 0? There are three possible program behaviors under SC, as follows: (1) $a = 0 \wedge b = 0$, (2) $a = 0 \wedge b = 1$, and (3) $a = 1 \wedge b = 0$. Nevertheless, the state $a = 0 \wedge b = 0$ cannot happen, as no interleaving of the threads of the programs yields that output. However, executing the program mentioned earlier on any modern system will occasionally provide a result in which both a and b have the value 0. The architectures follow non-sequentially consistent models that are based on the underlying organizations of the many systems in order to make it possible for such behaviors to occur. These models are referred to as "relaxed/weak" memory models. Note that the term "weak" in this context refers to the state of being weaker than SC.

2.2 C11 memory model

The recent memory for the C and C++ languages (named C11) is built on data-race-free, which ensures SC for programs without data races. According to the C11, racy programs exhibit undefined behavior. The C11 does not consider conflicting atomic accesses as races. The C11 memory model provides many modes as low-level atomics of memory accesses. These modes are allowed to be racy and are used to establish synchronization among non-atomic accesses. These models range from sequentially consistent (**sc**), which enforces a total ordering semantics, to weaker ones, such as release (**rel**) and acquire (**acq**), which can be used to implement message passing efficiently, and relaxed (**rlx**). In addition, it contains non-atomic (**na**) accesses, which are normal data accesses; therefore, the programmer is responsible for ensuring that they are correctly synchronized through other means.

Memory access operations in C11 include *load* (R), *store* (W), and *update* (U), which includes fetch-and-add and compare-and-swap. C11 also has *fence* (F) as a shared memory operation. *Load* and *store* operators can be either non-atomic or atomic, whereas *fence* and *update* operations are atomic. The memory access modes for loads can be **na**, **rlx**, **acq**, or **sc**, whereas the memory access modes for stores can be **na**, **rlx**, **rel**, or **sc**. The memory access mode for updates can be **rlx**, **acq**, **rel**, **relacq** (release-acquire), or **sc**. A fence can be composed of **rel**, **acq**, **relacq**, or **sc**. In order of increasing strength, these are: $\mathbf{na} \sqsubset \mathbf{rlx} \sqsubset \{\mathbf{acq}, \mathbf{rel}\} \sqsubset \mathbf{relacq} \sqsubset \mathbf{sc}$.

Notation for relations. To comprehend the semantics of C11, we will provide the notation used to explain binary relations. We write R to represent a binary relation. $R^?$, R^+, and R^* indicate reflexive, transitive, and reflexive-transitive closures, respectively. The notation $R_1; R_2$ denotes the left composition of two relations R_1 and R_2, i.e., $R_1; R_2 = \{(x, z) \mid \exists y.(x, y) \in R_1 \wedge (y, z) \in R_2\}$. The inverse relation of R is $R^{-1} = \{(x, y) \mid (y, x) \in R\}$. The notation $[A]$ stands for the identity relation on the set A, i.e., $[A] = \{(x, x) \mid x \in A\}$. Accordingly, $[A]; R$ is filtering R on the left with A, while $R; [B]$ is filtering R on the right with B. As a result, $[A]; R; [B] = R \cap (A \times B)$. If $R \cap [A] = \emptyset$, the relation R is irreflexive. The notations $R|_{=\mathrm{loc}}$ and $R|_{\neq \mathrm{loc}}$ indicate the relation R restricted to the same and different locations, respectively.

Semantics. The C11 introduces a set of relations between the shared memory accesses. Sequenced-before (sb) relation (commonly referred to as program order) captures the syntactic order of the intra-thread shared memory access. Reads-from (rf) relation connects two write and read accesses. When read access reads from write access, it forms a reads-from (rf) relation between the pair

of write and read accesses. Modification order (mo) specifies the order of the per-location write operations. The mo relation imposes a total order on all write operations to the same location during a given C execution. All threads are required to observe writes to a given location in this mo-order.

A few more additional relational types can be used to specify whether behaviors ought to be permitted formally. All of these relations can be derived from the three fundamental relations listed above (sequenced-before (sb), reads-from (rf), and modification order (mo) relations).

In particular, based on these three basic relations, C11 defines a variety of derived relations, including synchronization-with (sw) relation, happens-before (hb), and reads-before (rb) (Alglave et al. [2014]). When an acquire read or SC read reads from a release write or SC write, it establishes synchronization with (sw). The happens-before relation is the result of combining the synchronization-with and sequenced-before relations. It is worth noting that the happens-before is the transitive closure. Formally, this expression can be written as $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^{+}$. The reads-before (rb) relation relates read access and write access, where read access reads from a write that is mo-before. Formally, $\mathsf{rb} \triangleq \mathsf{rf}^{-1}; \mathsf{mo} \setminus [\mathtt{E}]$, where $\setminus[\mathtt{E}]$ is to exclude the case of an update event (in the context of a read-modify-write) from itself. Besides, we introduce other relations, if necessary, in each paragraph. We utilize filled arrows, dotted arrows, and dashed arrows with the same colors from sb, mo, and rf edges, respectively. We also have a similar way for other relations.

The following rules must be adhered to by the total order of sequentially consistent operations [Batty et al. 2016]. (1) There must be a total order on SC operations; any two SC operations must be ordered w.r.t. each other; (2) it must be consistent with hb and mo restricted to SC atomics; and (3) SC reads must either read from the most recent SC write before them in the SC order or from a non-SC write that does not happen before the most recent SC write to that location.

Notation for programs and executions. We write $a, b, \ldots$ for local variables (registers). Unless otherwise stated, we assume that all variables are initialized to 0. For a program, we use $\mathtt{a} := {}^{*o}\mathtt{x}$ to represent a read of x to a with access mode $o \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}, \mathtt{sc}\}$, whereas $\mathtt{x} :=_o \mathtt{a}$ for a write of a to location x with access mode $o \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{rel}, \mathtt{sc}\}$. Unless otherwise specified, we presume that $\mathtt{a} := {}^{*}\mathtt{x}$ and $\mathtt{x} := \mathtt{a}$ are normal read and write operations.

In the context of the execution of a program, we refer to the operation of reading v from l with the access mode $o \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}, \mathtt{sc}\}$ as $\mathsf{R}^{o}(l, v)$, while $\mathsf{W}^{o}(l, v)$ denotes the operation of writing v to l with the access mode $o \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{rel}, \mathtt{sc}\}$. Similarly, $\mathsf{U}^{o}(l, v, v')$ represents read-modify-writes operations with the access mode $o \in \{\mathtt{na}, \mathtt{rlx}, \mathtt{relacq}, \mathtt{sc}\}$. When referring to a memory SC fence, we also use the notation $\mathtt{fence}_{\mathtt{sc}}$.

Lastly, to refer to a specific program outcome, we write the values that specify that specific loads are intended to return comments (`//v`).

Axiomatic/Declarative approach. C/C++11 follows a standard approach to relaxed-memory semantics by defining an axiomatic or declarative memory model. One might use the axiom or declarative approach to display behaviors in relaxed memory models to place several constraints (or axioms) on the allowed program executions. Specifically, axiomatic semantics applies a set of constraints to determine if a given execution is valid. The constraints are applied to a set of program events and their relationships. The events represent the shared memory accesses in the execution of relaxed memory concurrency. There are several relations between these events. The events and relations in execution are shown as a graph, where the events and relations are shown as nodes and edges. Let us look at the store-buffering example mentioned in Section 2.1 to comprehend how the execution of a program is depicted as a graph.

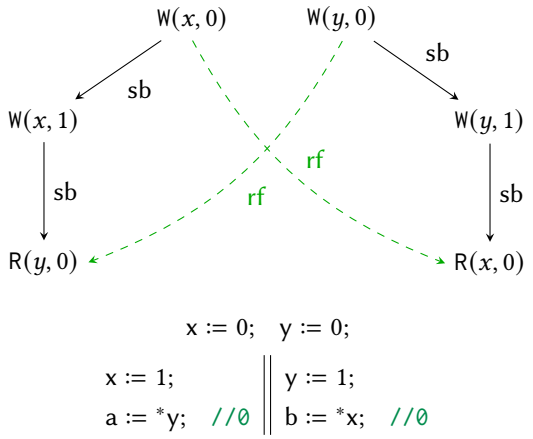


Fig. 1. A program and an execution of SB behavior.

$\mathtt{W}(x,0)$ and $\mathtt{W}(y,0)$ events on the graph of Figure 1 indicate that x and y have been initialized to zero. The $\mathtt{W}(x,1)$ and $\mathtt{W}(y,1)$ events reflect the store accesses in which x and y are both assigned values 1. The $\mathtt{R}(x,0)$ and $\mathtt{R}(y,0)$ events indicate that the reads of x and y both return values of 0. A sequenced-before (sb) relation (or program-order relation) is a relation between two events that captures the order in which the shared memory accesses have taken place. Note that the sb relation must be *irreflexive* for valid execution. This graph also has the read-from (rf) relation, which connects a write to a read event in which the read reads the value written by the write event.

2.3 Hardware memory models

x86. Despite being inferior to the sequentially consistent model, the Intel x86 memory model remains one of the strongest models among modern CPU implementations. The consistency order of the Intel-x86 architecture is based on the total store order (TSO) [Owens 2010], where the store order refers to the consistency order.

The x86 architecture provides each processor with a write buffer. Each thread is associated explicitly with a FIFO (first in, first out) store buffer. When a write operation is performed, the processor enqueues the write in the thread-local write buffer. This write only is visible to reads of the same thread. Writes in the store buffer will be debuffering in FIFO order and propagated to the main memory at non-deterministic points in time, making them visible to all other threads. For a read operation, the processor reads from the most recent corresponding write in its thread-local write buffer. If such a write exists, the value from that write is read. If not, the processor retrieves the value from the main memory.

Power and ARM. The ARM and POWER architectures are two well-known designs that are used today. Both of these architectures have extensively more relaxed memory models, allowing for a wider variety of hardware optimizations. In contrast to TSO models, these architectures' subsequent behaviors are possible. First, hardware threads can execute reads and writes out of order. Second, the memory system does not make sure that all hardware threads can see a write at the same time. This is called "write non-atomicity."

3 COMMON COMPILER OPTIMIZATIONS ARE INVALID

A compiler converts a program to an underlying architecture while preserving its semantics. Therefore, a compiler must be aware of the memory consistency models of both the programming language and the architecture. In contrast, several C11 compilers, such as GCC and LLVM, rarely perform C code compilation in a single step. Specifically, the front end translates the C11 program into the intermediate representations (IR) of the compiler, after which the compiler executes several optimizing transformations on the IR and generates the target code for the specific architecture. The IR enables compiler optimizations and mappings to architectures. When a transformation is correct, the semantics of the source program are preserved in the target program. The compiler performs two types of transformations: mapping shared memory primitives from one language to another and optimization-driven transformations.

In this section, only optimization transformations are presented. During the optimization transformation, source and target programs maintain the same consistency model. Common optimization transformations for concurrent programs include reordering independent memory accesses and removing redundant memory accesses. To optimize a C11 program in which x and y are independent shared variables, a compiler might execute reordering $x :=_{\texttt{rel}} 1; y :=_{\texttt{na}} 1; \rightsquigarrow y :=_{\texttt{na}} 1; x :=_{\texttt{rel}} 1;$. A compiler could also eliminate redundant shared memory access. $x :=_{\texttt{na}} 1; x :=_{\texttt{na}} 2; \rightsquigarrow x :=_{\texttt{na}} 2;$ is an example.

Vafeiadis et al. [2015] discovered counterexamples to demonstrate that the C11 memory model did not permit a number of common transformations and proposed a number of local fixes for the C11. In particular, they found that the C11 model formalized by [Batty et al. 2011] does not validate several source-to-source transformations expected to be performed by compilers and intended to be correct.

We first review the transformational approach, one of the various styles of defining concurrency models to demonstrate behaviors in relaxed memory models. In other words, the transformational approach explains certain relaxed memory behaviors via program transformation. We examine the possible interleaving executions of a given program to determine whether the desired result is possible by applying a set of allowed transformations. In the case of the load-buffering program depicted in Figure 2, the possible outcome that can be justified by reordering is a = b = 1.

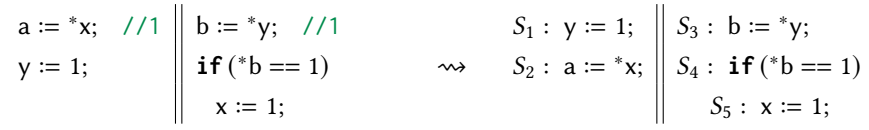


Fig. 2. Transformation of the load-buffering program.

Consider an interleaving S_1, S_3, S_4, S_5, and S_2, in which the load of y in the second thread reads from the store of S_1 while the load of x in the first thread reads from the store of S_5. Therefore, the result is a = b = 1. However, this approach is highly dependent on dependency analysis and allows transformations. In the ARMv7 architecture, for instance, a = 1 is a possible outcome, as seen in Figure 3.

Its behavior cannot be explained by any valid reordering followed by interleaving execution. However, if we look at a transformation that adds synchronization by sequentializing two concurrent accesses, such as $C_1||C_2 \rightsquigarrow C_1; C_2$, we might find a transformation sequence that results in an execution where a = 1 is possible. The transformation is depicted in Figure 4.

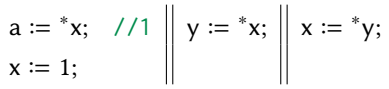


Fig. 3. Program with the annotated behavior a = 1.

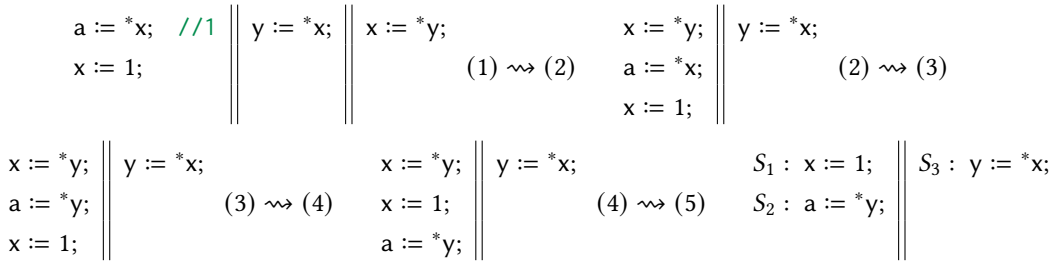


Fig. 4. Transformation of program in Figure 3.

By executing the third and first threads sequentially, the (1) ⇝ (2) transformation can be observed. Read-after-write elimination then results in (2) ⇝ (3). Then, (3) ⇝ (4) is determined by reordering x := 1 and a := *y. Finally, we observe that (4) ⇝ (5) overwrote write elimination and dead code elimination to eliminate x := *y. We can have an interleaving execution S_1, S_3, and S_2 where y and x are read from concurrent writes of x and y, respectively, resulting in a = 1.

Vafeiadis et al. [2015] discovered problems arising from program transformations, invalidating many expected source-to-source program transformations. These are presented below.

3.1 Causality Cycles and the ConsRFna Axiom

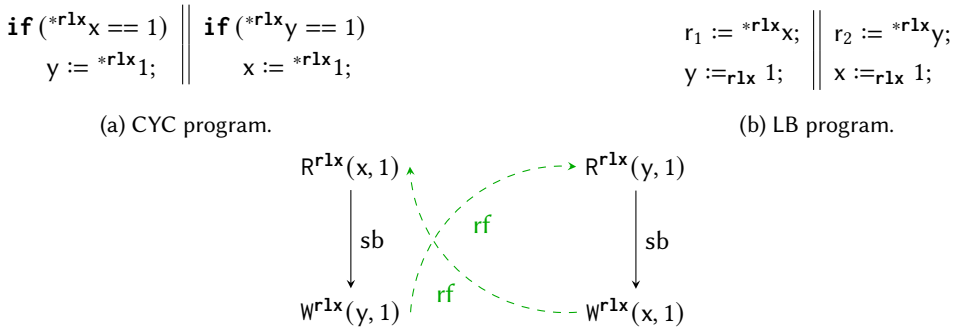
Traditionally, axiomatic models have been required to specify a set of constraints in order to guarantee the correctness of a single execution. However, there is no guarantee that the constraints will be able to distinguish between two underlying programs. The execution in Figure 5, for instance, permits a = b = 1 in LB and CYC programs. Such execution results in the CYC program's behavior described as *out-of-thin-air*. Out-of-thin-air behavior cannot take place in the actual execution of the CYC program.

Consider the SEQ example located on the left-hand side of Figure 6. No execution results in reading a happens. Assume there is an execution of reading a that occurs. Because the store a := 1 does not take place before a load of a in the second thread, the only value that can be returned by a load of a is the value of 0.

The ConsRFna axiom of the C11 model states that "If a read reads from a write and either the read or the write are non-atomic, then the write must have happened before the read." In other words, non-atomic loads must return the most recent write that happened before their execution.

Since a has a value of 0, y cannot be loaded, and y thus cannot return 1. It leads to the conclusion that x cannot take place. As a result, the load of x cannot return 1, and thus the load of a cannot occur. The only possible outcome is $a = 1 \wedge x = y = 0$.

On the other hand, if we apply $C_1 \,||\, C_2 \rightsquigarrow C_1; C_2$ transformation by sequentializing the first thread with the second thread of SEQ example, we get the new program on the right-hand side of Figure 6. The load of a returns the value 1 because the store to a happens before the load of a. The final result is a = x = y = 1. It would appear that the ConsRFna axiom is responsible for this.



(a) CYC program.

(b) LB program.

(c) Execution of CYC and LB programs.

Fig. 5. Programs and an execution of CYC and LB programs.

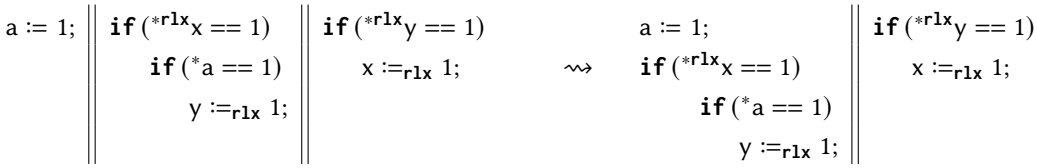


Fig. 6. Transformation of SEQ example.

3.2 Semantics of SC accesses

An SC read reads only from an SC write w that is immediately preceding in SC order to the same location or from a non-SC write that does not happen before w, according to C11 semantics (this is known as the SCReads axiom).

However, Vafeiadis et al. [2015] note that the restriction mentioned above is so strong that strengthening atomic access into a sequentially consistent one is unsound. Consider the behavior illustrated in Fig 7, in which the behavior is $r = s_1 = t_1 = 1 \wedge s_2 = t_2 = 2 \wedge s_3 = t_3 = 3$.

If the execution of the described above behavior holds (coherence of the relaxed loads in the final thread necessitates the mo-ordering), then $x :=_{\mathtt{sc}} 2$ in the first thread is the immediate SC preceding write w.r.t. read $r := {}^{*\mathtt{sc}}x$ in the third thread. In a consistent execution, however, read $r := {}^{*\mathtt{sc}}x$ *cannot* be read from the first store to x and return $r = 1$ since it always happens before $x :=_{\mathtt{sc}} 2$ (via sequenced-before). This execution is inconsistent as a result.

However, strengthening the $x :=_{\mathtt{rlx}} 3$ in the second thread into $x :=_{\mathtt{sc}} 3$, it establishes SC order from $y :=_{\mathtt{sc}} 1$ to $x :=_{\mathtt{sc}} 3$. Hence, $x :=_{\mathtt{sc}} 3$ is the immediately SC-preceding store to x. In other words, it is the immediate SC order successor of read $r := {}^{*\mathtt{sc}}x$ on location x. Currently reading 1 from $r := {}^{*\mathtt{sc}}x$ is valid due to the SC constraint that says $x :=_{\mathtt{rlx}} 1$ does *not* happen before $x :=_{\mathtt{sc}} 3$ (because $x :=_{\mathtt{rlx}} 1$ is in a different thread that has not been synchronized with).

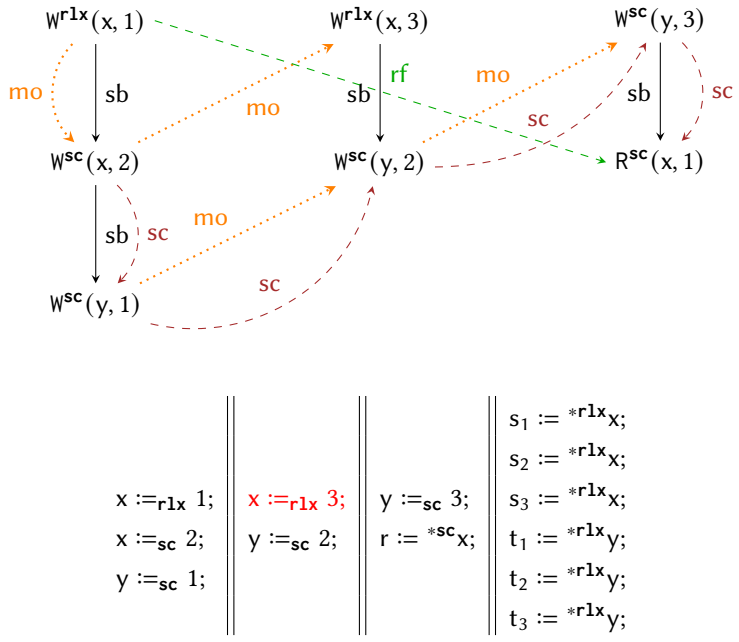


Fig. 7. A program and an execution of with annotated behavior: $r = s_1 = t_1 = 1 \wedge s_2 = t_2 = 2 \wedge s_3 = t_3 = 3$.

3.3 Non atomic reads

Roach motel reorderings are unsound. Moving a non-atomic store prior to a release write in the C11 is unsound. According to the C11 specification, a non-atomic read reads *only* from a happens before write. Nevertheless, Vafeiadis et al. [2015] show that this constraint is overly restrictive for desired compiler transformations such as roach motel reorderings. In the context of C11, Roach motel reorderings would allow moving non-atomic accesses before a release write or after an acquire read. Consider the case in Figure 8 by Vafeiadis et al. [2015].

```
z :=rel 1;  ||  if (*rlx x == 1)  // 0   ||  if (*rlx y == 1)  // 0
a :=na 1;   ||    if (*acq z == 1)  // 0 ||    x :=rlx 1;
            ||      if (*na a == 1)  // 0 ||
            ||        y :=rlx 1;         ||
```

Fig. 8. A program of roaching motel reorderings.

The only possible outcome in the source program is $z = a = 1 \wedge x = y = 0$. After the roach motel reordering of two writes in the first thread, we obtain a consistent execution, which could lead to the result $x = y = z = a = 1$. Consequently, roach motel reordering is *unsound* in C11 when a non-atomic read reads *only* from a happens-before write.

3.4 Proposed Fixes

3.4.1 ***Resolving Causality Cycles and the*** ConsRFna ***Axiom***. Vafeiadis et al. [2015] offered a number of local fixes for the C11 model that include the interaction between causality cycles

and the ConsRFna axiom. They specifically attempt to address causality cycles and the ConsRFna axiom. Below are some proposed fixes.

Ruling out ($\mathit{hb} \cup \mathit{rf}$) ***cycles.*** As stated in Section 3.1, the offending execution of the CYC program is exactly the same as the execution of the LB program. Intuitively, one way could be to attempt to prohibit causality cycles. They offered a way to avoid causality cycles by incorporating the axiom $\mathsf{acyclic}(\mathsf{hb} \cup \{(a,b) \mid \mathsf{rf}(b) = a\})$. Additionally, they also eliminate the ConsRFna axiom.

Ruling out only non-atomic cycles. Vafeiadis et al. [2015] also proposed another approach that replaces the ConsRFna axiom with a new model (Arfna) below.

$$\mathsf{acyclic}(\mathsf{hb} \cup \{(\mathsf{rf}(a), b) \mid \mathtt{isNA}(b) \vee \mathtt{isNA}(\mathsf{rf}(b))\}), \text{ where } \mathsf{isNA}(a) \triangleq \mathsf{mode}(a) = \mathbf{na} \quad \text{(Arfna)}$$

According to the Arfna, a non-atomic load may read from a concurrent write so long as there is no cycle. This new model (Arfna) permits all C11-allowed behaviors. It implies that any complication from C11 to hardware models (x86-TSO and Power) is retained in the new model. This model is not significantly weaker than C11.

3.4.2 ***Correcting the*** SCReads ***Axiom.*** We discussed a counterexample of weird behavior when strengthening $\mathsf{x} :=_{\mathtt{rlx}} 3$ into $\mathsf{x} :=_{\mathtt{sc}} 3$, introducing new behavior in Section 3.2. The axiom of SCReads is responsible for this counterexample. First, the SCReads axiom is written as follows:

$$\forall a, b.\ \mathsf{rf}(b) = a \wedge \mathsf{isSC}(b) \Rightarrow \mathsf{imm}(\mathsf{scr}, a, b) \vee \neg\mathsf{isSC}(a) \wedge \nexists x.\ \mathsf{hb}(a, x) \wedge \mathsf{imm}(\mathsf{scr}, x, b)$$

❶ $\mathsf{isSC}(a) \quad \triangleq \mathsf{mode}(a) = \mathbf{sc}$
❷ $\mathsf{imm}(R, a, b) \triangleq R(a, b) \wedge \nexists c.\ R(a, c) \wedge R(c, b)$
❸ $\mathsf{scr}(a, b) \quad \triangleq \mathsf{sc}(a, b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a)$, where $\mathsf{loc}(b)$ denotes for location accessed
❹ $\mathsf{iswrite}_l(a) \quad \triangleq \exists v.(\exists X, v'.\ \mathsf{lab}(a) \in \{\mathtt{W}^X(l, v), \mathtt{U}^X(l, v', v)\})$
where lab is a function that relates action identifier to actions,
and X is the memory access mode.

Fig. 9. Formal definition of the SCReads axiom.

The fact that the SCReads axiom disallows a non-SC write that happens before another SC-write that is immediately preceding in SC order to the same location is the leading cause of the issue caused by the SCReads axiom. Vafeiadis et al. [2015] proposed to strengthen the SCReads axiom by requiring there not to be a happen-before edge between a write event $\mathsf{rf}(b)$ and any same location write sc-prior to the read. They change $\mathsf{imm}(\mathsf{scr}, x, b)$ to $\mathsf{scr}(\mathsf{x}, \mathsf{b})$ and rewrite it as follows:

$$\forall a, b.\ \mathsf{rf}(b) = a \wedge \mathsf{isSC}(b) \Rightarrow \mathsf{imm}(\mathsf{scr}, a, b) \vee \neg\mathsf{isSC}(a) \wedge \nexists x.\mathsf{hb}(a, x) \wedge \mathsf{scr}(\mathsf{x}, \mathsf{b})$$

This new axiom rules out that $\mathsf{r} := {}^{*\mathtt{sc}}\mathsf{x}$ reading from $\mathsf{x} :=_{\mathtt{rlx}} 1$ is no longer valid, and it guarantees that the target program does not have questionable behavior. Thus, the modified SC read constraint preserves the correctness of the access strengthening transformation, i.e., there is no new behavior introduced.

3.5 Critique

Vafeiadis et al. [2015] indicated that the current rules for managing SC atomics violate several desirable memory model properties. They proposed a strengthening of the model to be repaired. It is worth noting that they offered a solution to out-of-thin-air reads that restricts the (hb ∪ rf) cycle in which either the source or destination of an rf edge is non-atomic. This solution has a number of nice properties, including the efficient mapping to ARM/Power in which a shared load and store pair do not require any intermediate fence. However, this solution disallows the reordering of non-atomic load and store operations.

The new model (Arfna) is proposed by Vafeiadis et al. [2015], which permits all C11-allowed behaviors. Any complication from C11 to the hardware models (x86-TSO and Power) is still maintained in the new model. Also, this model is not much weaker than C11. However, this approach is flawed when a non-atomic load is reordered past an adjacent non-atomic store. Consider the example in Figure 10 to illustrate how its behavior is prohibited by this model.

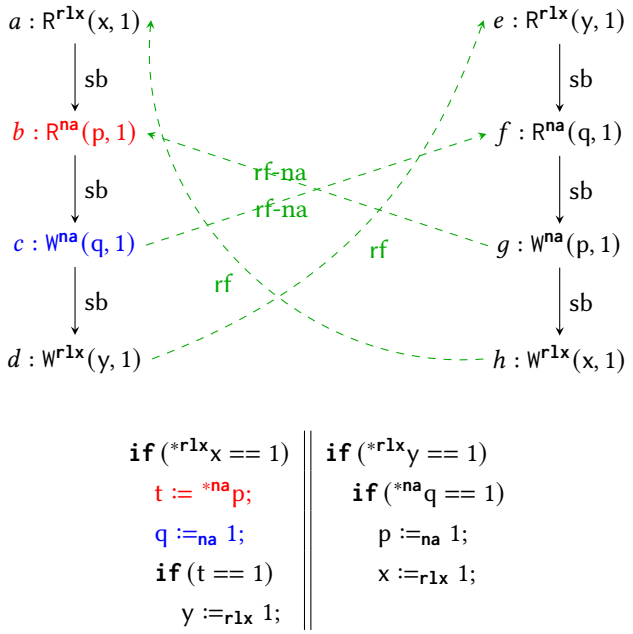


Fig. 10. A program and an execution of Arfna.

We can see that the program with the behavior p = q = 0 is the only possible outcome. The racy execution yielding p = q = 1 is not allowed because it contains a cycle that the Arfna axiom rules out. We observe the cycle $b \to c \to f \to g \to b$ in particular. However, when reordering the two adjacent non-atomic accesses of the first thread as

$$\mathsf{t} := {}^{*\mathsf{na}}\mathsf{p}; \mathsf{q} :=_{\mathsf{na}} 1; \rightsquigarrow \mathsf{q} :=_{\mathsf{na}} 1; \mathsf{t} := {}^{*\mathsf{na}}\mathsf{p};$$

Consequently, the racy execution, p = q = 1, is consistent. Thus, the event $c : \mathsf{W}^{\mathsf{na}}(\mathsf{q}, 1)$ is prior to the event $b : \mathsf{R}^{\mathsf{na}}(\mathsf{p}, 1)$, and there exists a path $c \to f \to g \to b$ that does not produce a cycle via the via rf-na and sb relations. The transformation, therefore, introduces new behavior and is invalid. So, compilers are not allowed to make such orders.

4 OVERHAULING SC ATOMICS IN C11

In this section, we describe several aspects of Batty et al. [2016], in which they provide more succinct semantics for SC atomics. Specifically, they provide more straightforward foundations for reasoning about C11. Their new model constructs a partial order for SC operations instead of requiring a total order. In addition, they provide proof that the C11 compilation for Power and x86 machines remains valid with their strengthening models.

We begin by examining a number of derived sets and relations in Figure 11 that Batty et al. [2016] formalized as C11 axioms by referencing the C11 standard. Note that S is a relation, which stands for *sequential consistency order*.

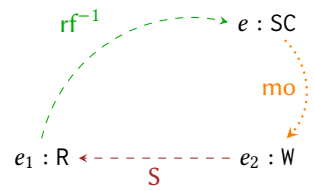
❶	$\mathsf{irr}(\mathsf{S}; r_1)$	where $r_1 = \mathsf{hb}$
❷	$\mathsf{irr}(\mathsf{S}; r_2)$	where $r_2 = ([\mathsf{F}^{\mathsf{sc}}]; \mathsf{sb})^?; \mathsf{mo}; (\mathsf{sb}; [\mathsf{F}^{\mathsf{sc}}])^?$
❸	$\mathsf{irr}(\mathsf{S}; r_3)$	where $r_3 = \mathsf{rf}^{-1}; [\mathsf{E}^{\mathsf{sc}}]; \mathsf{mo}$
❹	$\mathsf{irr}((\mathsf{S} \setminus (\mathsf{mo}; \mathsf{S})); r_4)$	where $r_4 = \mathsf{rf}^{-1}; \mathsf{hbl}; [\mathsf{E}^{\mathsf{W}}]$, and hbl is hb to events on the same location
❺	$\mathsf{irr}(\mathsf{S}; r_5)$	where $r_5 = ([\mathsf{F}^{\mathsf{sc}}]; \mathsf{sb}); \mathsf{rb}$
❻	$\mathsf{irr}(\mathsf{S}; r_6)$	where $r_6 = \mathsf{rb}; (\mathsf{sb}; [\mathsf{F}^{\mathsf{sc}}])$
❼	$\mathsf{irr}(\mathsf{S}; r_7)$	where $r_7 = ([\mathsf{F}^{\mathsf{sc}}]; \mathsf{sb}); \mathsf{rb}; (\mathsf{sb}; [\mathsf{F}^{\mathsf{sc}}])$

Fig. 11. Definition of further derived sets and relations.

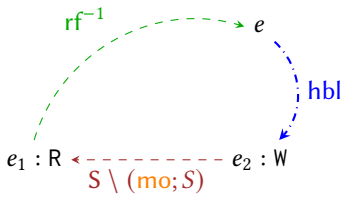
The axiom ❶ states that S must be in agreement with hb. The relationship between S and mo is governed by the axiom ❷, which states that if the write e_1 is mo-before the write e_2 (and any fences sequenced after e_2), then e_2 cannot come before e_1 (or any fences sequenced before e_1) in S.

The axioms ❸ and ❹ provide a constraint that if there are any SC writes to l preceding an SC read e_1 in S, then must e_1 must read either (1) from the most recent of these in S (call it e_2), or (2) from a non-SC write that does not happen before e_2. This condition is also mentioned in Section 3.2, which is referred to as SCReads.

First, the desired condition for (1) might forbid the read from an SC write that is not the most recent in S. As seen in Figure 12a, the constraint that is delivered by the axiom ❸ prohibits cycles of the form $(\mathsf{S}; r_3)$. In this particular instance, there is a path that goes as follows: $e_1 \to e \to e_2 \to e_1$, and we need to make sure that this path is not cyclic. Second, for (2), the desired condition might block e_1 from reading from a write e that happens before e_2, i.e., we want to disallow cycles (the axiom ❹) of the path $e_1 \to e \to e_2 \to e_1$, which is illustrated in Figure 12b.



(a) Shape depicited of axiom ❸.



(b) Shape depicited of axiom ❹.

Fig. 12. Diagrams depicted of axioms ❸ and ❹.

The axioms ❺, ❻ and ❼ rule SC fences. In particular, according to C11 semantics, if a read event e_1 of a location l is sequenced after an SC fence, then e_1 must not read from a write event e_2 to

location l, in which e_2 is mo-before another last write e that precedes the fence in S. Therefore, in order to rule out the existence of this property, $(\mathsf{S}; r_5)$ must not contain any cycles (the axiom ❺).

Also, if a write event e_2 to location l is sequenced before an SC fence, then any SC read of location l that follows the fence in S must not read from a write event to location l that is mo-before e_2. As a result, we have the axiom ❻, which states that $(\mathsf{S}; r_6)$ cannot include any cycles. Eventually, if a read event e_1 of location l is sequenced after an SC fence, and a write event e_2 to location l is sequenced before another SC fence that precedes the first fence in S, then e_1 must not read from a write event that is mo-before e_2. As a consequence, we get the axiom ❼, which states that $\mathsf{irr}(\mathsf{S}; r_7)$.

4.1 Constructing a partial order on SC operations

Batty et al. [2016] realized that the rules for SC axioms in C11 are intricate and challenging to comprehend. Then, they made some revisions. One of their changes was substituting the total order over SC with a partial order.

Regarding Definition 11, they remarked that all, with the exception of the axiom ❹ , can be expressed as $\mathsf{irr}(\mathsf{S}; r)$, where r is a relational expression. The form of the axiom ❹ that is distinct from all others is $(\mathsf{S} \setminus (\mathsf{mo}; \mathsf{S}))$. They sought to replace $\mathsf{S} \setminus (\mathsf{mo}; \mathsf{S})$ with only S in order to obtain the new axiom ❹✓, i.e., $\mathsf{irr}(\mathsf{S}; r_4)$. Then, we have the same form $\mathsf{irr}(\mathsf{S}; r_i)$, where $i = 1, \ldots, 7$ for seven axioms.

The new axiom ❹✓ coincides completely with the revised model offered by Vafeiadis et al. [2015], which is described in "Correcting the SCReads Axiom" in Section 3.4. Specifically, the new axiom ❹✓ states that it disallows an SC read to see any write that happens before any SC write in S. Meanwhile, the former axiom ❹ only forbids an SC read to observe any write that happens before the most recent SC in S. Batty et al. [2016] gives an attempt to replace the seven axioms with only the acyclicity axiom. Note that the axiom ❹✓ has been substituted for the axiom ❹.

A strict total order on SC events satisfies all seven axioms if and only if the union of all the constraints on S, when restricted to unequal SC events, is acyclic.

Just call $\mathsf{S}_{\mathsf{partial}} = \mathsf{acyclic}([\mathsf{E}^{\mathsf{sc}}]; (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7); [\mathsf{E}^{\mathsf{sc}}])$, then they prove

$$\mathsf{S}_{\mathsf{partial}} = \exists \mathsf{S}.\ \mathsf{WfS} \wedge ❶ \wedge ❷ \wedge ❸ \wedge ❹✓ \wedge ❺ \wedge ❻ \wedge ❼$$

It is important to remember that the condition WfS says that (1) S is a strict total order, and (2) S relation relates all and only the SC events that occur during execution. We also state it formally as $\mathsf{acyclic}(\mathsf{S}) \wedge (\mathsf{S} \cup \mathsf{S}^{-1}) = (([\mathsf{E}^{\mathsf{sc}}] \times [\mathsf{E}^{\mathsf{sc}}]) \setminus \mathsf{id})$, where id is the identity relation.

As a consequence, instead of needing the S relation, there is a new axiom, $\mathsf{S}_{\mathsf{partial}}$, that does not require the S relation anymore. The following gives the axiom $\mathsf{S}_{\mathsf{partial}}$ formally.

$$\mathsf{acyclic}([\mathsf{E}^{\mathsf{sc}}]; (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7); [\mathsf{E}^{\mathsf{sc}}]) \qquad (\mathsf{S}_{\mathsf{partial}})$$

4.2 A stronger and simpler SC axiom

After we have established the axiom $\mathsf{S}_{\mathsf{partial}}$, the question that arises is whether or not it is possible for it to strengthen the SC semantics without requiring changes to compilation schemes of any of the C11 target architectures (x86 and Power).

We note that both hb and mo relations constrain the S order between any combination of SC fences and atomics. We observe that the rb edges that begin or finish at a fence, the axioms ❺, ❻, and ❼ guarantee that the SC order is restricted to match. Meanwhile, the axiom ❹✓ contains rb edges covered by the hbl edges. Therefore, hbl is hb to events that take place at the same location, and when connecting with rf^{-1}, we also have $\mathsf{rf}^{-1}; \mathsf{mo}$ in the relation r_4 (see Figure 12b for further details). There is a slight difference with rb edges in the axiom ❸; the intermediate access creates

rb as an SC atomic. Then, naturally, all that has to be done is remove $[E^{sc}]$ from r_3 to get a new model with the name ❸✓ that has the form irr(S; rb).

Batty et al. [2016] proposed to enhance the axiom $S_{partial}$ by saying that the union of all the constraints $r_1, r_2, r_4, r_5, r_6, r_7$ and rb (instead r_3) on S, when restricted to unequal SC events, is acyclic. Formally, this expression is written as follows:

$$S_{simp} = \text{acyclic}([E^{sc}]; (r_1 \cup r_2 \cup \text{rb} \cup r_4 \cup r_5 \cup r_6 \cup r_7); [E^{sc}])$$

Then, they demonstrate that it is equivalent to

$$\text{acyclic}([E^{sc}]; (([F^{sc}]; \text{sb})^{?}; (\text{hb} \cup \text{rb} \cup \text{mo}); (\text{sb}; [F^{sc}])^{?})[E^{sc}]) \qquad (S_{simp})$$

4.3 Critique

In the original C11 model, the SC order was a primitive component. Nevertheless, Batty et al. [2016] exhibited that SC order is not a primitive component; it is possible to derive SC ordering among the SC accesses utilizing other relations.

However, such a new constraint proposed for SC accesses contains subtle weaknesses. Manerkar et al. [2016] provided counterexamples to demonstrate that the proposed constraints by [Batty et al. 2016] are too strong to preserve compiler correctness from C11 to Power and ARMv7 architectures. Let us consider the following example of the well-known Independent Reads Independent Writes (IRIW) in Figure 13, where initially x = y = 0.

$$x :=_{\mathbf{sc}} 1; \;\Big\|\; \begin{array}{l} a := {}^{*\mathbf{acq}}x; \quad //1 \\ c := {}^{*\mathbf{sc}}y; \quad //0 \end{array} \;\Big\|\; \begin{array}{l} b := {}^{*\mathbf{acq}}y; \quad //1 \\ d := {}^{*\mathbf{sc}}x; \quad //0 \end{array} \;\Big\|\; y :=_{\mathbf{sc}} 1;$$

Fig. 13. A program of IRIW-acq-sc.

The result $a = b = 1 \wedge c = d = 0$ is restricted by the C11 semantics presented by Batty et al. [2016]; nevertheless, it allows the output when mapped to Power instructions. Also, the semantics do not guarantee interleaving semantics when there are SC fences between every shared memory access pair.

Lahav et al. [2016] also pointed out that Batty et al. [2016]'s semantics of SC fences is stronger than C11's semantics. However, it is weaker than theirs and still allows weak behaviors for release-acquire programs even when fences are placed between every two commands.

Next, Lahav et al. [2017] addressed the above problems of SC semantics, and they proposed RC11 (Repaired C11) semantics to demonstrate the correctness of RC11 to Power and ARMv7 compilations. Their semantics guarantee that putting SC fences between every pair of shared memory accesses restores interleaving behavior.

5 REPAIRING SEQUENTIAL CONSISTENCY IN C11

Following Section 3 and Section 4, this part delivers Lahav et al. [2017]'s work that addresses these problems of SC semantics and proposes RC11 semantics. In particular, they are concerned with the semantics of SC atomics, including SC accesses and SC fences. The main concern they address is mixing SC and non-SC accesses at the *same location*. Several examples of code mixing SC accesses with release/acquire or relaxed accesses (non-SC) to the same location are seqlocks (Boehm [2012]) and Rust's crossbeam library (Turon [2016]). The axiomatic model of RC11 incorporated the proposed fixes of Vafeiadis et al. [2015] and addressed the unsound compilation strategies. Lahav et al. [2017] prove the correctness of RC11 for Power and ARMv7 compilations. The proposed semantics also ensure that placing SC fences between every pair of shared memory accesses restores interleaving behavior.

Before diving into specifics, we review the semantics of SC atomics in C11. Remember that, for consistency in C11 execution, hb must be irreflexive and represent an acyclic(sb ∪ sw). It also involves the coherence (SC-per-location) and atomicity of RMWs (read-modify-write). The C11 coherence axioms are demonstrated by several forbidden (non-consistent) behaviors by Batty et al. [2011], presented as follows:

- **COHERENCE-WW** requires that mo and hb may not disagree.
- **NO-FUTURE-READ** says that a read may not happen before the write it reads from.
- **COHERENCE-RW** requires that a read may not happen before a write that mo-before the write it reads from.
- **COHERENCE-WR** requires that a read may not read from a write that is already hidden by another write that happens before it.
- **COHERENCE-RR** requires that two reads connected by hb may not be read from write with the inverse order in mo.

There should be a strict total order S overall SC events corresponding to the order in which these events are executed. This order is needed to fulfill several conditions (mentioned in Figure 11) below.

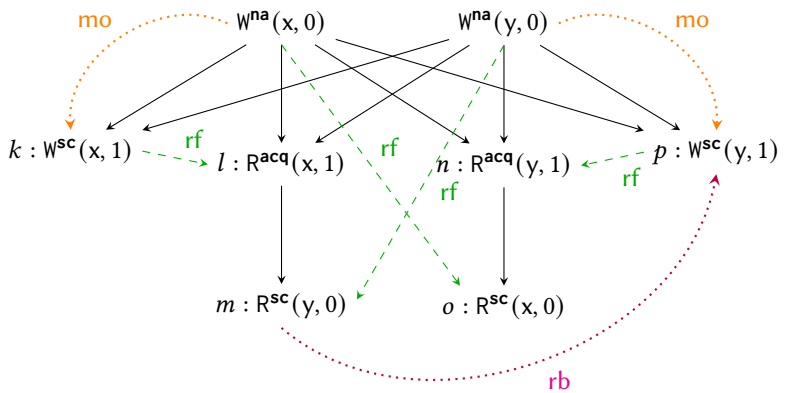
❶ $[E^{sc}]; hb; [E^{sc}] \subseteq S$
❷ $[E^{sc}]; mo; [E^{sc}] \subseteq S$
❸ $[E^{sc}]; rb; [E^{sc}] \subseteq S$
❹ ❺ ❻ ❼ indicate that S is required to comply with a few more conditions about SC fences.

Informally, the condition ❶ says that S must include hb, which is restricted to SC events, whereas the conditions ❷ and ❸ include mo and rb, which are also restricted to SC events.

5.1 Compilation to Power is Broken

Consider a program and an execution of IRIW-acq-sc in Figure 14, whose annotated behavior is forbidden by C11. Notice that S needs to have a **strict total order**. Since these are SC accesses, we have $S(p, k)$. In addition, there is a strict total order $S(k, m)$ since there is a hb via l (which satisfies ❶). Then, based on transitivity, $S(p, m)$. On the other hand, we also have $S(m, p)$ since $(m, p) \in rb$ (fulfilling ❸). Therefore, S contains a cycle. It is banned by C11. Unfortunately, its compilation into Power permits the behavior of the above example.

To maintain coherence, hardware memory models provide strong ordering guarantees on accesses to the *same location*. It is not difficult to ensure that the compilation preserves these conditions (even for Power and ARM). Especially, ❷ and ❸ only force ordering between accesses to the same location. However, for ❶, ensuring a strong ordering between accesses of *different* locations, compiling for weaker hardware requires the insertion of fence instructions.



$$x :=_{sc} 1; \;\Big\|\; a := {}^{*acq}x; \;\; //1 \;\; c := {}^{*sc}y; \;\; //0 \;\Big\|\; b := {}^{*acq}y; \;\; //1 \;\; d := {}^{*sc}x; \;\; //0 \;\Big\|\; y :=_{sc} 1;$$

Fig. 14. A program and an execution of IRIW-acq-sc.

As previously stated, ❶ is *too* strong if the relation hb must be included in S. There are two observations here. First, if two events, a and b, access the same location, the hardware will maintain the order. Second, if the hb path from a to b starts and ends with an sb edge, there should be a sync fence between SC accesses a and b.

As shown below, Lahav et al. [2017] proposed replacing condition ❶ with a weaker condition, ❶✓. The condition S must include hb, which is restricted to any SC event, as long as the hb path between the two events begins and ends with sb edges or accesses to the same location. Specifically, they suggest replacing hb with $(\mathsf{sb} \cup \mathsf{sb}; \mathsf{hb}; \mathsf{sb} \cup \mathsf{hb}|_{=\mathsf{loc}})$.

5.1.1 ***Fixing the model.*** Instead of expressing ❶, ❷, and ❸ as different conditions on the total order S, they only need one acyclicity condition, namely, $\mathsf{acyclic}([\mathsf{E}^{\mathsf{sc}}]; (\mathsf{hb} \cup \mathsf{mo} \cup \mathsf{rb}); [\mathsf{E}^{\mathsf{sc}}])$. As a result, the new condition is acyclic (❶✓) as follows.

$$\mathsf{acyclic}([\mathsf{E}^{\mathsf{sc}}]; (\mathsf{sb} \cup \mathsf{sb}; \mathsf{hb}; \mathsf{sb} \cup \mathsf{hb}|_{=\mathsf{loc}} \cup \mathsf{mo} \cup \mathsf{rb}); [\mathsf{E}^{\mathsf{sc}}]) \qquad (❶✓)$$

5.1.2 ***Enabling Elimination of SC Accesses.*** The condition mentioned above forbids the elimination of an SC write immediately followed by another SC write to the same location and an SC read immediately followed by an SC read from the same location.

Consider the WWmerge example in Figure 15, in which the source program disallows a cycle from $m \to l \to o \to p \to m$ with the annotated behavior shown as WWmerge. Nonetheless, it will be permitted after eliminating $x :=_{sc} 1$, i.e., event $m : W^{sc}(x, 1)$. It is a cycle created from $n \to k \to l \to o \to p \to n$. Therefore, it violates the condition that ❶ ✓ must be acyclic.

Naturally, weakening the condition by replacing sb; hb; sb with $\mathsf{sb}|_{\neq\mathsf{loc}}; \mathsf{hb}; \mathsf{sb}|_{\neq\mathsf{loc}}$ where $\mathsf{sb}|_{\neq\mathsf{loc}}$ denotes sb edges that are not between accesses to the same location.

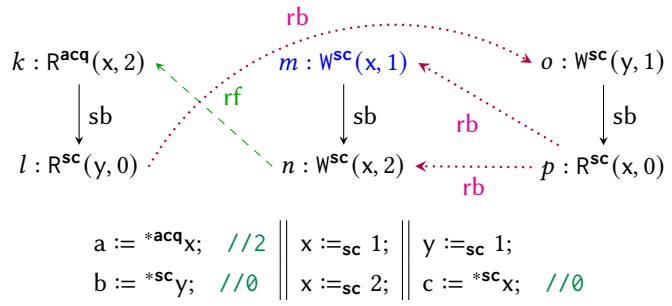


Fig. 15. A program and an execution of WWmerge.

Then, the new condition (named SC-before) is constructed by requiring the acyclicity of $[\mathsf{E}^{\mathsf{sc}}]; \mathsf{scb}; [\mathsf{E}^{\mathsf{sc}}]$ where

$$\mathsf{scb} = \mathsf{sb} \cup \mathsf{sb}|_{\neq \mathsf{loc}}; \mathsf{hb}; \mathsf{sb}|_{\neq \mathsf{loc}} \cup \mathsf{hb}|_{=\mathsf{loc}} \cup \mathsf{mo} \cup \mathsf{rb}$$

5.2 SC Fences are Too Weak

According to the discussion in Section 4.3, one might want to guarantee that adding SC fences between every pair of shared memory accesses will restore interleaving behavior. However, it is not the case with the original C11 model or its strengthening (Batty et al. [2016]).

Adapted from the condition of Batty et al. [2016], that is,

$$\mathsf{acyclic}(([\mathsf{E}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathsf{sb}^?); (\mathsf{hb} \cup \mathsf{mo} \cup \mathsf{rb}); ([\mathsf{E}^{\mathsf{sc}}] \cup \mathsf{sb}^?; [\mathsf{F}^{\mathsf{sc}}]))$$

they also expand the model to encompass SC fences by substituting scb for $\mathsf{hb} \cup \mathsf{mo} \cup \mathsf{rb}$. Therefore, the new model is

$$\mathsf{psc}_1 = ([\mathsf{E}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathsf{sb}^?); \mathsf{scb}; ([\mathsf{E}^{\mathsf{sc}}] \cup \mathsf{sb}^?; [\mathsf{F}^{\mathsf{sc}}])$$

and there are no cycles in psc_1.

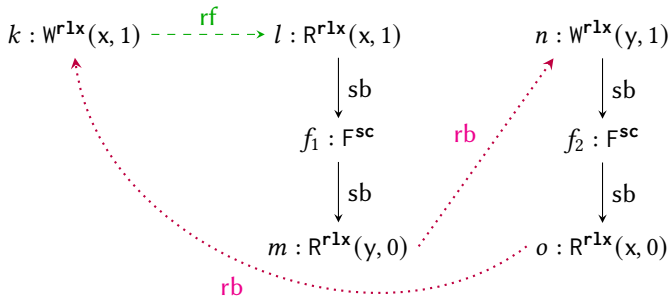


Fig. 16. A program and an execution of RWC+syncs.

Consider the example in Figure 16 with annotated behavior ($a = 1 \wedge b = 0 \wedge c = 0$) to show it is allowed according to the model of Batty et al. [2016]. There is a certain path from f_1 to f_2 via $sb \rightarrow rb \rightarrow sb$. We also observe that there is a way from f_2 to f_1 via $sb \rightarrow rb \rightarrow rf \rightarrow sb$; however, this path contributes neither to the condition of Batty et al. [2016] nor psc_1. Nevertheless, the above behavior is prohibited in all implementations of C11.

5.2.1 ***Fixing the model.*** Lahav et al. [2017] presented an extension of coherence order (called extended-coherence-order relation), which states that two events are loosely ordered before each other. This extension was written as $eco \triangleq (rf \cup mo \cup rb)^+$. Its order combines three distinct notions: the reads-from rf relation, the modification order mo, and the reads-before rb relations. As a result, the new condition associated with eco is an acyclic representation, as seen below.

$$\text{acyclic}(psc_1 \cup [F^{sc}]; sb; eco; sb; [F^{sc}])$$

Such condition disallows the weak behavior of RWC+syncs; specifically, it is impossible for there to be a cycle in either psc_1 or $[F^{sc}]; sb; eco; sb; [F^{sc}]$.

5.3 Out-of-Thin-Air Reads

The annotated behavior $a = b = 1$ is permitted with the formalized C11 model by Batty et al. [2011] in this program, despite the absence of the value 1 in the program LB+deps shown in Figure 17. The initial C11 model of Batty et al. [2011] is too relaxed to provide a DRF-SC (Sequentially consistent execution for data race free) guarantee[1] and eliminate OOTA execution. In contrast, LB+deps exhibits a violation of DRF-SC.

```
a := *rlx x;   //1   ||  b := *rlx y;   //1
if (a == 1)          ||  if (b == 1)
    y :=rlx a;       ||      x :=rlx b;
```

Fig. 17. A program and an execution of LB+deps.

The C11 memory model also allows for Load-Buffering (LB) behavior when $a = b = 1$. The issue with LB is that the same execution in Figure 18 justifies an unwanted behavior of the program LB+deps in Figure 17, where the reads read the value of 1 despite the fact that the value of 1 does not appear in the program.

Lahav et al. [2017] proposed an obvious solution: they prohibit all LB behaviors by requiring $(sb \cup rf)$ to be acyclic. It forbids the weak behavior of LB+deps and LB behavior, which the Power and AR architectures allow.

[1]Programs written in the fragment with only nonatomic and SC accesses and locks and free of races do not exhibit relaxed behavior.

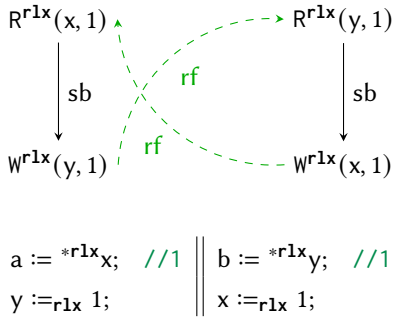


Fig. 18. A program and an execution of LB behavior.

5.4 Critique

Chakraborty and Vafeiadis [2019] suggested an alternative approach to the challenge of preventing out-of-thin-air behaviors using the promising semantics presented by [Kang et al. 2017]. Their memory model is built on event structures and has two variants: the weaker version, WEAKEST, and the stronger version, WEAKERSTMO. While WEAKEST replicates promising semantics, WEAKERSTMO overcomes the out-of-thin-air problem, permits desirable optimizations, and can be mapped to the x86, PowerPC, and ARMv7 architectures. WEAKERSTMO model is weaker than RC11, then Chakraborty and Vafeiadis [2019] use the RC11 mapping to obtain correct mappings to PowerPC and ARM. Even though these mappings are correct, they are sub-optimal in that they insert a fake conditional branch after each relaxed load.

While RC11 a provides full-featured weak memory model, a repaired version of the C/C++11 specification that fixes certain issues involving sequentially consistent accesses, it does not support reasoning about the liveness of the simplest shared-memory concurrent programs under sequential consistency, which typically require some fairness assumptions about the scheduler. Lahav et al. [2021] proposed a fairness condition that maintains the validity of local transformations and the compilation scheme from RC11 to x86-TSO. Additionally, it offers formal proofs of termination of mutual exclusion lock implementations under declarative weak memory models.

Compilation techniques from high-level concurrent programming languages with weak semantics to common multi-core platforms such as x86-TSO, ARM, and Power are validated by RC11. These compilation approaches directly map each high-level primitive to a sequence of machine instructions without code optimizations (such as reordering and eliminating reads or writes). It seems challenging to apply their methods to verify optimization passes.

REFERENCES

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 1–74.

Mark Batty, Alastair F Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 634–648.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL*. 55–66.

Hans-J Boehm. 2012. Can seqlocks get along with programming language memory models?. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. 12–20.

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–28.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *POPL* (Paris, France). ACM, 175–189.

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. *ACM SIGPLAN Notices* 51, 1 (2016), 649–662.

Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making weak memory models fair. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*.

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

Yatin A Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *arXiv preprint arXiv:1611.01507* (2016).

Scott Owens. 2010. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *European Conference on Object-Oriented Programming*. Springer, 478–503.

Aaron Turon. 2016. Crossbeam: support for concurrent and parallel programming.

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