Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it

Viktor Vafeiadis
MPI-SWS

Thibaut Balabonski
INRIA

Soham Chakraborty
MPI-SWS

Robin Morisset
INRIA

Francesco Zappa Nardelli
INRIA

Abstract

We show that the weak memory model introduced by the 2011 C and C++ standards does not permit many common source-to-source program transformations (such as expression linearisation and "roach motel" reorderings) that modern compilers perform and that are deemed to be correct. As such it cannot be used to define the semantics of intermediate languages of compilers, as, for instance, LLVM aimed to. We consider a number of possible local fixes, some strengthening and some weakening the model. We evaluate the proposed fixes by determining which program transformations are valid with respect to each of the patched models. We provide formal Coq proofs of their correctness or counterexamples as appropriate.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Concurrency; Weak memory models; C/C++; Compilers; Program transformations

1. Introduction

Programmers want to understand the code they write, compilers (and hardware) try hard to optimise it. Alas, in concurrent systems even simple compiler optimisations like constant propagation can introduce unexpected behaviours! The *memory models* of programming languages are designed to resolve this tension, by governing which values can be returned when the system reads from shared memory. However, designing memory models is hard: it requires finding a compromise between providing an understandable and portable execution model for concurrent programs to programmers, while allowing common compiler optimisations.

It is well-known that only racy programs (that is, programs in which two threads can access the same resource concurrently in conflicting ways) can observe normal compiler and hardware optimisations. A common approach for a programming language is thus to require that race-free code must exhibit only sequentially-consistent (that is, interleaving) behaviours, while racy code is undefined and has no semantics. This approach, usually referred to as *DRF (data race freedom)*, is appealing to the programmer because under the hypothesis that the shared state is properly protected by locks he has to reason only about interleaving of memory accesses. It is also appealing to the compiler because it can optimise code freely provided that it respects synchronisations. A study by Ševčík [18] shows that it is indeed the case that in an idealised DRF model common compiler optimisations are correct. These include elimination and reorderings of non-synchronising memory accesses, and the so-called *"roach motel" reorderings* [10]: moving a memory access after a lock or before an unlock instruction. Intuitively, the latter amounts to enlarging a critical section, which should be obviously correct.

Although the idealised DRF design is appealing, integrating it into a complete language design is not straightforward because additional complexity has to be taken into account. For instance, Java relies on unforgeability of pointers to enforce its security model, and the Java memory model (JSR-133) [10] must impose additional restrictions to ensure that all programs (including racy programs) enjoy some basic memory safety guarantees. The resulting model is intricate, and fails to allow some optimisations implemented in the HotSpot reference compiler [17]. Despite ongoing efforts, no satisfactory fix to JSR-133 has been proposed yet.

The recent memory model for the C and C++ languages [8, 7], from now on referred to as C11, is also based on the DRF model. Since these languages are not type safe, the Java restrictions are unnecessary and both languages simply state that racy programs have undefined behaviour. However, requiring all programs to be well-synchronised via a locking mechanism is unacceptable when it comes to writing low-level high-performance code, for which C and C++ are often the languages of choice. An escape mechanism called *low-level atomics* was built into the model. The idea is to not consider conflicting atomic accesses as races, and to specify their semantics by attributes annotated on each memory access. These range from *sequentially consistent* (SC), which imposes a total ordering semantics, to weaker ones as *release* (REL) and *acquire* (ACQ), which can be used to efficiently implement message passing, and *relaxed* (RLX), whose purpose is to allow performing single hardware loads and stores without the overhead of memory barrier instructions. As a result, RLX accesses do not synchronise with one another and provide extremely weak ordering guarantees.

A common belief is that the C11 memory model enables all common compiler optimisations, and indeed Morisset et al. [11] proved that Ševčík's correctness theorem for eliminations and reorderings of non-atomic accesses holds in the C11 memory model.

POPL'15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676995

The authors however did not consider transformations involving low-level atomic memory accesses.

Nowadays mainstream compilers are becoming aggressive in performing optimisations that involve atomic accesses; for instance in gcc 4.10, reorderings of SC atomic loads with non-atomic loads can be observed as a side effect of partial-redundancy elimination, while clang 3.5 routinely reorders non-atomic and RLX accesses. A complete understanding of the validity of compiler optimisations in the C11 memory model is now a necessity to guide not only the future standard evolution but also current compiler development.

In this paper we set out to perform an in-depth study of optimisations in the C11 memory model. In particular we build on, and extend, the results of [11] by considering optimisations that involve atomic accesses. Unexpected surprises lurked behind the corner.

Standard Source-to-Source Transformations are Invalid in C11. Surprisingly, and contradicting the common belief, we discovered that the C11 model, as defined in the C/C++ standards and formalised by Batty et al. [4], does not validate a number of source-to-source transformations that are routinely performed by compilers and are intended to be correct. As an appetiser, in what follows we show that *sequentialisation*, a simple transformation that adds synchronisation by sequentialising two concurrent accesses:

$$C_1 \| C_2 \rightsquigarrow C_1; C_2$$

is unsound even when C_1 consists of a single non-atomic variable access. Most of our counterexamples exploit the counterintuitive *causality cycles* allowed by the C11 semantics. To understand these, first consider the following code:

$$\begin{array}{l} r_1 = x.\mathsf{load}(\text{RLX}); \\ y.\mathsf{store}(1, \text{RLX}); \end{array} \;\Big\|\; \begin{array}{l} r_2 = y.\mathsf{load}(\text{RLX}); \\ x.\mathsf{store}(1, \text{RLX}); \end{array} \qquad \text{(LB)}$$

(in all our examples all variables are initialised to 0 before the parallel composition, unless specified otherwise). Since relaxed atomic accesses by design do not race and do not synchronise, it is perfectly reasonable to get $r_1 = r_2 = 1$ at the end of some execution: the memory accesses in each thread are independent and the compiler or the hardware might have reordered them.

The C11 standard keeps track of relative ordering between the memory accesses performed during program execution via the *happens-before* relation (shortened hb), defined as the transitive closure of the program order and of the synchronisations between actions of different threads.[1] Non-atomic loads must return the last write in the hb relation: this is unique in race-free programs and guarantees a sequentially consistent semantics for race-free programs with only non-atomic accesses. For relaxed atomic accesses the intentions of the standard are more liberal and basically state that a relaxed load can see any other write which does not happen after it (according to hb), and which is not shadowed by another write, effectively allowing the outcome $r_1 = r_2 = 1$ above.

Unfortunately, the definition above enables some controversial behaviours. For instance, the program below can terminate with $x = y = 1$ as well:

$$\begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{RLX})) \\ \quad y.\mathsf{store}(1, \text{RLX}); \end{array} \;\Big\|\; \begin{array}{l} \mathsf{if}\ (y.\mathsf{load}(\text{RLX})) \\ \quad x.\mathsf{store}(1, \text{RLX}); \end{array} \qquad \text{(CYC)}$$

Again there are no synchronisations, and relaxed loads can see arbitrary stores. However, justifying this in terms of compiler or hardware optimisations is harder: the first thread might speculate that x has value 1 tentatively executing the store to y, while the second thread speculates that the value of y is 1 tentatively executing the store to x. The two threads then check if the speculation was correct, seeing each other's tentative stores that justify the speculation.

[1] For simplicity, we assume there are no consume atomic accesses.

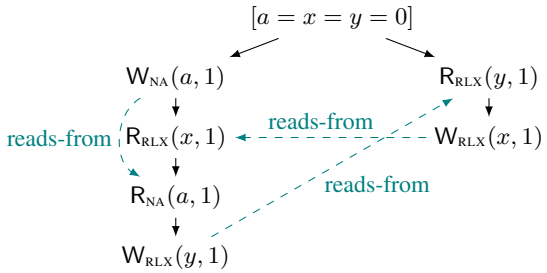


Figure 1. Execution resulting in $a = x = y = 1$.

Several authors have observed that causality cycles make code verification infeasible [2, 12, 16]. We show that the situation is even worse than that, because we can exploit them to show that standard program transformations are unsound. Consider:

$$a = 1; \;\Big\|\; \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{RLX})) \\ \quad \mathsf{if}\ (a) \\ \quad\quad y.\mathsf{store}(1, \text{RLX}); \end{array} \;\Big\|\; \begin{array}{l} \mathsf{if}\ (y.\mathsf{load}(\text{RLX})) \\ \quad x.\mathsf{store}(1, \text{RLX}); \end{array} \qquad \text{(SEQ)}$$

First, notice that there is no execution (*consistent execution* in the terminology of Section 2) in which the load of a occurs. We show this by contradiction. Suppose that there is an execution in which a load of a occurs. In such an execution the load of a can only return 0 (the initial value of a) because the store $a = 1$ does not happen before it (because it is in a different thread that has not been synchronised with) and non-atomic loads must return the latest write that happens before them. Therefore, in this execution the store to y does not happen, which in turn means that the load of y cannot return 1 and the store to x also does not happen. Then, x cannot read 1, and thus the load of a does not occur. As a consequence this program is *not racy*: since the load of a does not occur in any execution, there are no executions with conflicting accesses on the same non-atomic variable. We conclude that the only possible final state is $a = 1 \wedge x = y = 0$.

Now, imagine we apply sequentialisation, collapsing the first two threads and moving the assignment to the start:

$$\begin{array}{l} a = 1; \\ \mathsf{if}\ (x.\mathsf{load}(\text{RLX})) \\ \quad \mathsf{if}\ (a) \\ \quad\quad y.\mathsf{store}(1, \text{RLX}); \end{array} \;\Big\|\; \begin{array}{l} \mathsf{if}\ (y.\mathsf{load}(\text{RLX})) \\ \quad x.\mathsf{store}(1, \text{RLX}); \end{array}$$

Running the resulting code can lead to an execution, formally depicted in Figure 1, in which the load of a actually returns the value 1 since the store to a now happens before (via program-order) the load. This results in the final state $a = x = y = 1$, which is not possible for the initial program.

Consequences. The example above is an instance of *source-to-source* program transformation: the semantics of both the source and target code are defined by the C11 memory model. It might be argued that the main purpose of compiler is not to perform a source-to-source translation but rather compile C11 programs to x86/ARM/Power assembler (to cite three widespread architectures), and a correctness statement for a compiler should relate the C11 semantics of the source program to the x86/ARM/Power semantics of the generated assembler. Indeed if we compile the transformed code above using the standard mapping for low-level atomics for x86 [4] or ARM/Power [13], then the problematic new behaviour does not arise in practice. To the best of our knowledge no modern relaxed architecture allows the causality cycle (or the other idiosyncrasies of the C11 model we exploit) built by the program labelled by (CYC) to terminate with $x = y = 1$. This implies

that our counterexamples do not break C11-to-assembly compiler correctness statements, in contrast to what happens in Java [17].

However compilers rarely compile C code into assembly code in just one pass. Our counterexamples imply that the C11 memory model cannot be used to give semantics to the intermediate languages used internally by a compiler, as for instance the Clang/LLVM compiler aimed to. They also imply that reasoning about the correctness of program transformations cannot be done at the C11 level but must take into account the actual mapping of atomic accesses to a particular architecture, forbidding architecture independent reasoning and preventing compositional reasoning about compiler passes.

The design of a memory model that forbids causality cycles while enabling common compiler optimisation is currently a Holy Grail quest. Our counterexamples exploit a precise form of causality cycles (involving control dependencies) and not the most general form [6]; unfortunately it turns out that there is no simple local fix to the C11 model that makes all these transformations valid.

Contributions and Outline.

- We show that several source-to-source transformations intended to be correct, can introduce new behaviours in the C11 memory model. The transformations we consider include sequentialisation, strengthening, and roach motel reorderings; we present them and demonstrate that C11 forbids them in Section 3.
- We explore a number of possible local fixes to the C11 model, some strengthening and some weakening the model. These involve replacing one C11 consistency axiom by another; we formalise them in Section 4 and study their basic metatheory in Section 5. These include the acyclicity condition advocated by Boehm and Demsky [6] as well as weaker conditions.

  For each patched model, in Sections 6 and 7, we conduct an in-depth study of the soundness of a wide class of program transformations, involving reordering and eliminations of both non-atomic and atomic variables. For each we either provide a proof of its correctness formalised in the Coq proof assistant (with one exception), or a counterexample. For the condition in [6], under an additional condition on sequentially consistent accesses, all the intended transformations are valid. The weaker conditions either disallow some transformations or do not satisfy the DRF theorem. Additionally we show that the side conditions on the memory attributes of the operations involved in each sound optimisation are locally maximal, in that we have counterexamples for any weakening of them.
- We show that "Write-after-Read" elimination of atomic accesses is unsound in the C11 memory model, both in the current formulation and in the patched models (Section 7.1).
- Our investigation also highlighted some corner cases of the C11 model which break important metatheory properties. We discuss them, together with possible fixes, in Sections 4.3 and 5.4.

To make the paper self-contained we recall the presentation of the C11 memory model and the setup to reason about program transformations in Section 2. We finally discuss related work in Section 8. The Coq proof scripts and our appendix with the counterexamples are available at the following URL:

`http://plv.mpi-sws.org/c11comp/`

2. Abstract Optimisations in C11

In this paper, we are not looking at the actual algorithms used to implement compiler optimisations. Rather, we are concerned by the *effects* of compiler optimisations on *program executions*. We thus build on the representation of *abstract optimisations* introduced by Ševčík [18] and adapted to the C11 memory model in Morisset et al. [11], which we recall below. The subsection headings refer to the relevant files in our Coq development.

2.1 Representation of Programs [actions.v, opsemsets.v]

To abstract from the syntax complexity of the C language, we identify a source program with a set of descriptions of what *actions* it can perform when executed in an arbitrary context.

More precisely, in a source program each thread consists of a sequence of *instructions*. We assume that, for each thread, a thread-local semantics associates to each *instruction instance* zero, one, or more shared memory accesses, which we call *actions*. The actions we consider, ranged over by act, are of the form:

$$\begin{aligned}\Phi ::= {} & \mathsf{skip} \mid \mathsf{W}_{(\text{SC}|\text{REL}|\text{RLX}|\text{NA})}(\ell, v) \mid \mathsf{R}_{(\text{SC}|\text{ACQ}|\text{RLX}|\text{NA})}(\ell, v) \\ & \mid \mathsf{C}_{(\text{SC}|\text{REL-ACQ}|\text{ACQ}|\text{REL}|\text{RLX})}(\ell, v, v') \mid \mathsf{F}_{(\text{ACQ}|\text{REL})} \mid \mathsf{A}(\ell) \\ act ::= {} & tid : \Phi\end{aligned}$$

where ℓ ranges over memory locations, v over values and $tid \in \{1..n\}$ over thread identifiers. We consider atomic and non-atomic loads from (denoted R) and stores to (W) memory, fences (F), read-modify-writes (C), and allocations (A) of memory locations. To simplify the statement of some theorems, we also include a no-op (skip) action. Each action specifies its thread identifier tid, the location ℓ it affects, the value read or written v (when applicable), and the memory-order (written as a subscript, when applicable).[2] We assume a labelling function, lab, that associates action identifiers (ranged over by $a, b, r, w, \ldots$) to actions. In the drawings we usually omit thread and action identifiers.

We introduce some terminology regarding actions. A *read action* is a load or a read-modify-write (RMW); a *write* is a load or an RMW; a *memory access* is a load, store or RMW. Where applicable, we write $\mathsf{mode}(a)$ for the memory order of an action, $\mathsf{tid}(a)$ for its thread identifier, and $\mathsf{loc}(a)$ for the location accessed. We say an action is *non-atomic* iff its memory-order is NA, and *SC-atomic* iff it is SC. An *acquire* action has memory-order ACQ or stronger, while a *release* has REL or stronger. The *is stronger* relation, written $\sqsupseteq$: $\mathcal{P}(MO \times MO)$, is defined to be the least reflexive and transitive relation containing SC $\sqsupseteq$ REL-ACQ $\sqsupseteq$ REL $\sqsupseteq$ RLX, and REL-ACQ $\sqsupseteq$ ACQ $\sqsupseteq$ RLX.

The thread local semantics captures control flow dependencies via the *sequenced-before* (sb) relation, which relates action identifiers of the same thread that follow one another in control flow. We have $sb(a, b)$ if a and b belong to the same thread and a precedes b in the thread's control flow. Even among actions of the same thread, the sequenced-before relation is not necessarily total because the order of evaluation of the arguments of functions, or of the operands of most operators, is underspecified in C and C++. The thread local semantics also captures thread creation via the *additional-synchronised-with* (asw) relation, that orders all the action identifiers of a thread after the corresponding thread fork (which can be represented by a skip action).

Summarising, the thread local semantics identifies each program execution a triple $O = (lab, sb, asw)$, called an *opsem*. As an example, Figure 4 depicts one opsem for the program on the left and one for the program on the right. Both opsems correspond to the executions obtained from an initial state where y holds 3, and

[2] We omit *consume* atomics and sequentially consistent fences. The semantics of the former is intricate and at the time of writing no major compiler profits from their weaker semantics, treating consume as acquire; the semantics of the latter is unclear. Despite their name, SC fences do not guarantee sequential consistency even when placed between every two instructions. This is because while they rule out the relaxed behaviour of the SB (store buffering) example, they permit those of the IRIW (independent reads of independent writes) example.

$$
\begin{array}{rcl|rcl|rcl}
\mathsf{isread}_{\ell,v}(a) & \stackrel{\text{def}}{=} & \exists X, v'.\ lab(a) \in \{\mathsf{R}_X(\ell, v), \mathsf{C}_X(\ell, v, v')\} & \mathsf{isread}_{\ell}(a) & \stackrel{\text{def}}{=} & \exists v.\ \mathsf{isread}_{\ell,v}(a) & \mathsf{isread}(a) & \stackrel{\text{def}}{=} & \exists \ell.\ \mathsf{isread}_{\ell}(a) \\
\mathsf{iswrite}_{\ell,v}(a) & \stackrel{\text{def}}{=} & \exists X, v'.\ lab(a) \in \{\mathsf{W}_X(\ell, v), \mathsf{C}_X(\ell, v', v)\} & \mathsf{iswrite}_{\ell}(a) & \stackrel{\text{def}}{=} & \exists v.\ \mathsf{iswrite}_{\ell,v}(a) & \mathsf{iswrite}(a) & \stackrel{\text{def}}{=} & \exists \ell.\ \mathsf{iswrite}_{\ell}(a) \\
\mathsf{isfence}(a) & \stackrel{\text{def}}{=} & lab(a) \in \{\mathsf{F}_{\text{ACQ}}, \mathsf{F}_{\text{REL}}\} & \mathsf{isaccess}(a) & \stackrel{\text{def}}{=} & \mathsf{isread}(a) \vee \mathsf{iswrite}(a) & \mathsf{isNA}(a) & \stackrel{\text{def}}{=} & \mathsf{mode}(a) = \text{NA} \\
\mathsf{sameThread}(a, b) & \stackrel{\text{def}}{=} & \mathsf{tid}(a) = \mathsf{tid}(b) & \mathsf{isrmw}(a) & \stackrel{\text{def}}{=} & \mathsf{isread}(a) \wedge \mathsf{iswrite}(a) & \mathsf{isSC}(a) & \stackrel{\text{def}}{=} & \mathsf{mode}(a) = \text{SC} \\
\mathsf{rsElem}(a, b) & \stackrel{\text{def}}{=} & \mathsf{sameThread}(a, b) \vee \mathsf{isrmw}(b) & \mathsf{isAcq}(a) & \stackrel{\text{def}}{=} & \mathsf{mode}(a) \sqsupseteq \text{ACQ} & \mathsf{isRel}(a) & \stackrel{\text{def}}{=} & \mathsf{mode}(a) \sqsupseteq \text{REL}
\end{array}
$$

$$
\mathsf{rseq}(a, b) \stackrel{\text{def}}{=} a = b \vee \mathsf{rsElem}(a, b) \wedge mo(a, b) \wedge (\forall c.\ mo(a, c) \wedge mo(c, b) \Rightarrow \mathsf{rsElem}(a, c))
$$

$$
\mathsf{sw}(a, b) \stackrel{\text{def}}{=} \exists c, d.\ \begin{array}{l} \neg\mathsf{sameThread}(a, b) \wedge \mathsf{isRel}(a) \wedge \mathsf{isAcq}(b) \wedge \mathsf{rseq}(c, rf(d)) \\ \wedge\ (a = c \vee \mathsf{isfence}(a) \wedge sb^+(a, c)) \wedge (d = b \vee \mathsf{isfence}(b) \wedge sb^+(d, b)) \end{array}
$$

$$
\mathsf{hb} \stackrel{\text{def}}{=} (sb \cup \mathsf{sw} \cup asw)^+
$$

$$
\mathsf{Racy} \stackrel{\text{def}}{=} \exists a, b.\ \begin{array}{l} \mathsf{isaccess}(a) \wedge \mathsf{isaccess}(b) \wedge \mathsf{loc}(a) = \mathsf{loc}(b) \wedge a \neq b \\ \wedge (\mathsf{iswrite}(a) \vee \mathsf{iswrite}(b)) \wedge (\mathsf{isNA}(a) \vee \mathsf{isNA}(b)) \wedge \neg(\mathsf{hb}(a, b) \vee \mathsf{hb}(b, a)) \end{array}
$$

$$
\mathsf{Observation} \stackrel{\text{def}}{=} \{(a, b) \mid mo(a, b) \wedge \mathsf{loc}(a) = \mathsf{loc}(b) = \texttt{world}\}
$$

Figure 2. Auxiliary definitions for a C11 execution $(lab, sb, asw, rf, mo, sc)$.

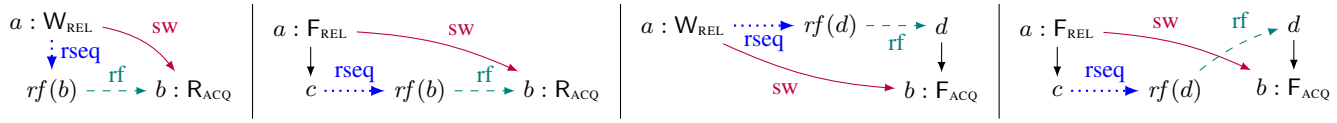


Figure 3. Illustration of the "synchronizes-with" definition: the four cases inducing an sw edge.

the environment does not perform any write to the shared variables (each read returns the last value written).

The set of all the opsems of a program is an *opsemset*, denoted by S. We require opsemsets to be *receptive*: S is receptive if, for every opsem O, for every read action r in the opsem O, for all values v' there is an opsem O' in S which only differs from O because the read r returns v' rather than v, and for the actions that occur after r in $sb \cup asw$. Intuitively an opsemset is receptive if it defines a behaviour for each possible value returned by each read.

We additionally require opsemsets to be prefix-closed, assuming that a program can halt at any time. Formally, we say that an opsem O' is a prefix of an opsem O if there is an injection of the actions of O' into the actions of O that behaves as the identity on actions, preserves sb and asw, and, for each action $x \in O'$, whenever $x \in O$ and $(sb \cup asw)(y, x)$, it holds that $y \in O'$.

Program Transformations. Opsemsets abstract the syntax of programs by identifying each program with the set of actions it can perform in an arbitrary environment. We can then characterise the effect of an arbitrary source code transformation directly on opsemsets. On a given opsem, the effect of any transformation of the source code is to *eliminate*, *reorder*, or *introduce* actions and modifying the sb and asw relations accordingly.

In the example in Figure 4, taken from Morisset et al. [11], the loop on the left is optimised into the code on the right by loop invariant code motion. As we said, the figure shows opsems for the initial state $z = 0$, $y = 3$ assuming that the code is not run in parallel with an interfering context. Observe that the effect of the optimisation on the first opsem is to eliminate the shaded actions, and to reorder the stores to `x`, thus mapping the opsem of the unoptimised code into an opsem of the optimised code.

An opsem captures a possible execution of the program, so by applying a transformation to an opsem we are actually optimising one particular execution. Lifting pointwise this definition of *semantic transformations* to opsemsets enables optimising all the execution paths of a program, one at a time, thus abstracting from actual source program transformation.

Soundness of program transformations can then be formalised by identifying the set of conditions under which eliminating, reordering or introducing actions in the opsems of an opsemset does not introduce new observable behaviours. We must thus define what it means to *execute* an opsemset.

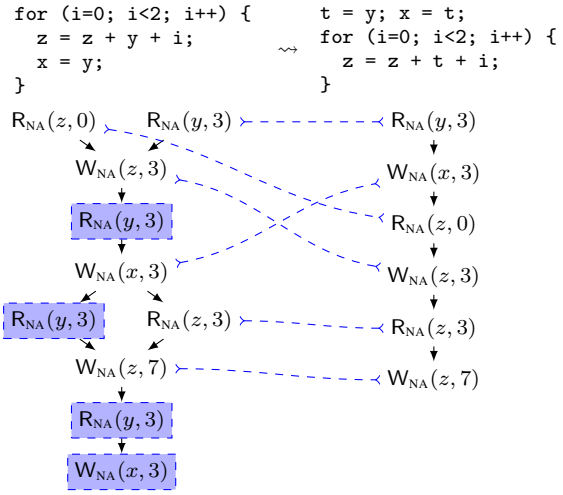


Figure 4. Effect of loop invariant code motion on an opsem.

2.2 Executing Programs [c11.v, lang.v]

The mapping of programs to opsemsets only takes into account the structure of each thread's statements, not the semantics of memory operations. In particular, the values of reads are chosen arbitrarily, without regard for writes that have taken place. (In our Coq development, we present such a mapping from programs to opsemsets for a concurrent WHILE language.)

The C11 memory model then filters inconsistent opsems by constructing additional relations and checking the resulting candidate executions against the axioms of the model. For the subset of C11 we consider, a witness W for an opsem O contains the following additional relations:[3]

[3] The full model includes two additional relations, dd (data dependency) and dob (dependency ordered before), used to define hb for consume reads.

$$\forall a, b.\ sb(a,b) \implies \mathsf{tid}(a) = \mathsf{tid}(b) \qquad \text{(ConsSB)}$$

$$\mathsf{order}(\mathsf{iswrite}, mo) \wedge \forall \ell.\ \mathsf{total}(\mathsf{iswrite}_\ell, mo) \qquad \text{(ConsMO)}$$

$$\mathsf{order}(\mathsf{isSC}, sc) \wedge \mathsf{total}(\mathsf{isSC}, sc) \wedge (\mathsf{hb} \cup mo) \cap (\mathsf{isSC} \times \mathsf{isSC}) \subseteq sc \qquad \text{(ConsSC)}$$

$$\forall b.\ (\exists c.\ rf(b) = c) \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a,b) \qquad \text{(ConsRFdom)}$$

$$\forall a, b.\ rf(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \qquad \text{(ConsRF)}$$

$$\forall a, b.\ rf(b) = a \wedge (\mathsf{isNA}(a) \vee \mathsf{isNA}(b)) \implies \mathsf{hb}(a,b) \qquad \text{(ConsRFna)}$$

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

$$\nexists a.\ \mathsf{hb}(a,a) \qquad \text{(IrrHB)}$$

$$\nexists a, b.\ rf(b) = a \wedge \mathsf{hb}(b,a) \qquad \text{(ConsRFhb)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(b,a) \qquad \text{(CohWW)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(rf(b), rf(a)) \qquad \text{(CohRR)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(rf(b), a) \qquad \text{(CohWR)}$$

$$\nexists a, b.\ \mathsf{hb}(a,b) \wedge mo(b, rf(a)) \qquad \text{(CohRW)}$$

$$\forall a, b.\ \mathsf{isrmw}(a) \wedge rf(a) = b \implies \mathsf{imm}(mo, b, a) \qquad \text{(AtRMW)}$$

$$\forall a, b, \ell.\ lab(a) = lab(b) = \mathsf{A}(\ell) \implies a = b \qquad \text{(ConsAlloc)}$$

where $\mathsf{order}(P,R) \stackrel{\mathsf{def}}{=} (\nexists a.\ R(a,a)) \wedge (R^+ \subseteq R) \wedge (R \subseteq P \times P)$

$\mathsf{total}(P,R) \stackrel{\mathsf{def}}{=} (\forall a, b.\ P(a) \wedge P(b) \implies a = b \vee R(a,b) \vee R(b,a))$

$\mathsf{imm}(R,a,b) \stackrel{\mathsf{def}}{=} R(a,b) \wedge \nexists c.\ R(a,c) \wedge R(c,b)$

$\mathsf{scr}(a,b) \stackrel{\mathsf{def}}{=} sc(a,b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(a)$

Figure 5. Axioms satisfied by consistent C11 executions, $\mathsf{Consistent}(lab, sb, asw, rf, mo, sc)$.

- The *reads-from* map (rf) maps every read action r to the write action w that wrote the value read by r.
- The *modification-order* (mo) relates writes to the same location; for every location, it is a total order among the writes to that location.
- The *sequential-consistency* order (sc) is a total order over all SC-atomic actions. (The standard calls this relation S.)

From these relations, C11 defines a number of derived relations (written in sans-serif font), the most important of which are: the *synchronizes-with* relation and the *happens-before* order.

- *Synchronizes-with* (sw) relates each release write with the acquire reads that read from some write in its release sequence (rseq). This sequence includes the release write and certain subsequent writes in modification order that belong to the same thread or are RMW operations. The sw relation also relates fences under similar conditions. Roughly speaking, a release fence turns succeeding writes in sb into releases and an acquire fence turns preceding reads into acquires. (For details, see the definition in Figure 2 and the illustration in Figure 3.)
- *Happens-before* (hb) is a partial order on actions formalising the intuition that one action was completed before the other. In the C11 subset we consider, $\mathsf{hb} = (sb \cup \mathsf{sw} \cup asw)^+$.

We refer to a pair of an opsem and a witness (O, W) as a *candidate execution*. A candidate execution is said to be *consistent* if it satisfies the axioms of the memory model, which will be presented shortly. The model finally checks if none of the consistent executions contains an *undefined behaviour*, arising from a *race* (two conflicting accesses not related by hb)[4] or a *memory error* (accessing an unallocated location), where two accesses are conflicting if they are to the same address, at least one is a write, and at least one is non-atomic. Programs that exhibit an undefined behaviour in one of their consistent executions are undefined; programs that do not exhibit any undefined behaviour are called *well-defined*, and their semantics is given by the set of their consistent executions.

Consistent Executions. According to the C11 model, a candidate execution $(lab, sb, asw, rf, mo, sc)$ is consistent if all of the properties shown in Figure 5 hold.

(ConsSB) Sequenced-before relates only same-thread actions.

(ConsMO) Writes on the same location are totally ordered by mo.

(ConsSC) The sc relation must be a total order over SC actions and include both hb and mo restricted to SC actions. This in effect means that SC actions are globally synchronised.

(ConsRFdom) The reads-from map, rf, is defined for those read actions for which the execution contains an earlier write to the same location.

(ConsRF) Each entry in the reads-from map, rf, should map a read to a write to the same location and with the same value.

(ConsRFna) 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. Batty et al. [4] additionally require the write to be *visible*: i.e. not to have been overwritten by another write that happened before the read. This extra condition is unnecessary, as it follows from (CohWR).

(SCReads) SC reads are restricted to read only from the immediately preceding SC write to the same location in sc order or from a non-SC write that has not happened before that immediately preceding SC write.

(IrrHB) The happens-before order, hb, must be irreflexive: an action cannot happen before itself.

(ConsRFhb) A read cannot read from a future write.

(CohWW, CohRR, CohWR, CohRW) Next, we have four coherence properties relating mo, hb, and rf on accesses to the same location. These properties require that mo never contradicts hb or the observed read order, and that rf never reads values that have been overwritten by more recent actions that happened before the read.

(AtRMW) Read-modify-write accesses execute atomically: they read from the immediately preceding write in mo.

(ConsAlloc) The same location cannot be allocated twice by different allocation actions. (This axiom is sound because for simplicity we do not model deallocation. The C11 model by Batty et al. [4, 3] does not even model allocation.)

Observable Behaviour. The *observable behaviour* of a candidate execution is the restriction of the mo relation to the distinguished `world` location. If none of the candidate executions of a program exhibit an undefined behaviour, then its observable behaviour is the set of all observable behaviours of its candidate executions. In our counterexamples, we often distinguish executions based on the final values of memory—this is valid because there could be a context program reading those values and writing them to `world`.

[4] The standard distinguishes between races arising from accesses of different threads, which it calls *data races*, and from those of the same thread, which it calls *unsequenced races*. The standard says unsequenced races can occur even between atomic accesses.

$x.\mathsf{store}(1,\text{RLX});$
$x.\mathsf{store}(2,\text{SC});$
$y.\mathsf{store}(1,\text{SC});$
$\|$
$x.\mathsf{store}(3,\text{RLX});$
$y.\mathsf{store}(2,\text{SC});$
$\|$
$y.\mathsf{store}(3,\text{SC});$
$r = x.\mathsf{load}(\text{SC});$
$\|$
$s_1 = x.\mathsf{load}(\text{RLX});$
$s_2 = x.\mathsf{load}(\text{RLX});$
$s_3 = x.\mathsf{load}(\text{RLX});$
$t_1 = y.\mathsf{load}(\text{RLX});$
$t_2 = y.\mathsf{load}(\text{RLX});$
$t_3 = y.\mathsf{load}(\text{RLX});$

Behaviour in question: $r = s_1 = t_1 = 1 \ \wedge \ s_2 = t_2 = 2 \ \wedge \ s_3 = t_3 = 3$

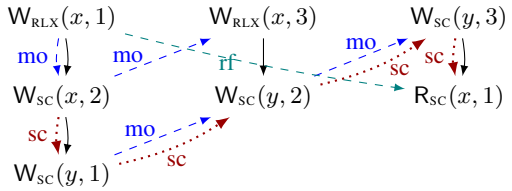


Figure 6. A weird consequence of the SCReads axiom: strengthening the $x.\mathsf{store}(3,\text{RLX})$ into $x.\mathsf{store}(3,\text{SC})$ introduces new behaviour.

3. Invalid Source-to-Source Transformations

In the introduction we discussed how *sequentialisation*, a simple transformation rewriting $C_1 \| C_2 \rightsquigarrow C_1; C_2$ can introduce new behaviours in C11 programs. Here we present other surprising problems that arise from innocent-looking program transformations.

Strengthening is Unsound. A desirable property of a memory model is that adding synchronisation to a program introduces no new behaviour (other than deadlock). The following example shows however that replacing a relaxed atomic store with a release atomic store is unsound in C11. Consider:

$a = 1;$
$z.\mathsf{store}(1,\text{RLX});$
$\|$
$\mathsf{if}\ (x.\mathsf{load}(\text{RLX}))$
$\quad \mathsf{if}\ (z.\mathsf{load}(\text{ACQ}))$
$\quad\quad \mathsf{if}\ (a)$
$\quad\quad\quad y.\mathsf{store}(1,\text{RLX});$
$\|$
$\mathsf{if}\ (y.\mathsf{load}(\text{RLX}))$
$\quad x.\mathsf{store}(1,\text{RLX});$

As in the SEQ program from Section 1, the load of a cannot return 1 because the store to a does not happen before it (and this time we can name the axiom responsible for this: ConsRFna). Therefore, the only final state is $a = z = 1 \wedge x = y = 0$. If, however, we make the store of z a release store, then it synchronises with the acquire load, and it is easy to build a consistent execution with final state $a = z = x = y = 1$. A symmetric counterexample can be constructed for strengthening a relaxed load to an acquire load.

What is more interesting is that *even in the absence of causality cycles*, strengthening an atomic access into a sequentially consistent one is unsound in general. Consider, for example, the program in Figure 6, where coherence of the relaxed loads in the final thread forces the mo-orderings to be as shown in the execution on the right of the figure. Now, the question is whether the SC-load can read from the first store to x and return $r = 1$. In the program as shown, it cannot, because that store happens before the $x.\mathsf{store}(2,\text{SC})$ store, which is the immediate sc-preceding store to x before the load. If, however, we also make the $x.\mathsf{store}(3,\text{RLX})$ be sequentially consistent, then it becomes the immediately sc-preceding store to x, and hence reading $r = 1$ is no longer blocked.

Roach Motel Reorderings are Unsound. Roach motel reorderings are a class of optimisations that let compilers move accesses to memory into synchronised blocks, but not move them out: the intuition is that it is always safe to move more computations (including memory accesses) inside critical sections. In the context of C11, roach motel reorderings would allow moving non-atomic accesses after an acquire read (which behaves as a lock operation) or before a release write (which behaves as an unlock operation).

However the following example program shows that in C11 it is unsound to move a non-atomic store before a release store.

$z.\mathsf{store}(1,\text{REL});$
$a = 1;$
$\|$
$\mathsf{if}\ (x.\mathsf{load}(\text{RLX}))$
$\quad \mathsf{if}\ (z.\mathsf{load}(\text{ACQ}))$
$\quad\quad \mathsf{if}\ (a)$
$\quad\quad\quad y.\mathsf{store}(1,\text{RLX});$
$\|$
$\mathsf{if}\ (y.\mathsf{load}(\text{RLX}))$
$\quad x.\mathsf{store}(1,\text{RLX});$

As before, the only possible final state of this program is $a = z = 1$ and $x = y = 0$. If, however, we reorder the two stores in the first thread, we get a consistent execution leading to the final state

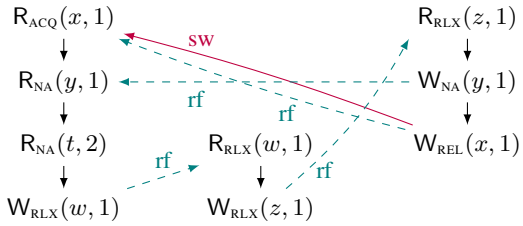


Figure 7. Execution generating new behaviour if the expression evaluation order is linearised.

$a = z = x = y = 1$. Again, we can construct a similar example showing that reordering over an acquire load is also not allowed by C11.

Expression Linearisation is Unsound. A simple variation of sequentialisation is expression evaluation order *linearisation*, a transformation that adds an sb arrow between two actions of the same thread and that every compiler is bound to perform. This transformation is unsound as demonstrated below:

$t = x.\mathsf{load}(\text{ACQ}) + y;$
$\mathsf{if}\ (t == 2)$
$\quad w.\mathsf{store}(1,\text{RLX});$
$\|$
$\mathsf{if}\ (w.\mathsf{load}(\text{RLX}))$
$\quad z.\mathsf{store}(1,\text{RLX});$
$\|$
$\mathsf{if}\ (z.\mathsf{load}(\text{RLX}))$
$\quad y = 1;$
$\quad x.\mathsf{store}(1,\text{REL});$

The only possible final state for this program has all variables, including t, set to zero. Indeed, the store $y = 1;$ does not happen before the load of y, which can then return only 0. However, if the $t = x.\mathsf{load}(\text{ACQ}) + y;$ is linearised into $t = x.\mathsf{load}(\text{ACQ}); t = t + y;$, then a synchronisation on x induces an order on the accesses to y, and the execution shown in Figure 7 is allowed.

4. Further C11 Weaknesses and Proposed Fixes

In this section, we consider possible solutions to the problems identified in the previous section, as well as to two other weaknesses with the C11 model, which however do not manifest themselves as invalid program transformations. (All of the models in this section as well as the relationships among them are formalised in `c11.v`.)

4.1 Resolving Causality Cycles and the ConsRFna Axiom

We first discuss possible solutions for the most important problem with C11, namely the interaction between causality cycles and the ConsRFna axiom.

Naive Fix. A first, rather naive solution is to permit causality cycles, but drop the offending ConsRFna axiom. As we will show in Sections 6 and 7, this solution allows all the optimisations that were intended to be sound on C11. It is, however, of dubious usefulness as it gives extremely weak guarantees to programmers.

The DRF theorem—stating that programs whose sequential consistent executions have no data races, have no additional relaxed behaviours besides the SC ones—does not hold. As a counterexample, take the CYC program from the introduction, replacing the relaxed accesses by non-atomic ones.

Arf: Forbidding $(\mathsf{hb} \cup rf)$ ***Cycles.*** A second, much more reasonable solution is to try to rule out causality cycles. Ruling out causality cycles, while allowing non-causal loops in $\mathsf{hb} \cup rf$ is, however, difficult and cannot be done by stating additional axioms over single executions. This is essentially because the offending execution of the CYC program from the introduction is also an execution of the LB program, also from the introduction.

As an approximation, we can rule out all $(\mathsf{hb} \cup rf)$ cycles, by stating the following axiom:

$$\mathsf{acyclic}(\mathsf{hb} \cup \{(a, b) \mid rf(b) = a\}) \qquad (\mathsf{Arf})$$

This solution has been proposed before by Boehm and Demsky [6] and also by Vafeiadis and Narayan [16]. Here, however, we take a subtly different approach from the aforementioned proposals in that besides adding the Arf axiom, we also drop the problematic ConsRFna axiom.

In Sections 6 and 7 we show that this model allows the same optimisations as the naive one (i.e., all the intended ones), except the reordering of atomic reads over atomic writes.

It is however known to make relaxed accesses more costly on ARM/Power, as there must be either a bogus branch or a lightweight fence between every shared load and shared store [6].

Arfna: Forbidding Only Non-Atomic Cycles. Another approach is to instead make more behaviours consistent, so that the non-atomic accesses in the SEQ example from the introduction can actually occur and race. The simplest way to do this is to replace ConsRFna by

$$\mathsf{acyclic}(\mathsf{hb} \cup \{(rf(b), b) \mid \mathsf{isNA}(b) \vee \mathsf{isNA}(rf(b))\}) \qquad (\mathsf{Arfna})$$

A non-atomic load can read from a concurrent write, as long as it does not cause a causality cycle.

This new model has several nice properties. First, it is weaker than C11 in that it allows all behaviours permitted by C11. This entails that any compilation strategy proved correct from C11 to hardware memory models, such as to x86-TSO and Power, remains correct in the modified model (contrary to the previous fix).

Theorem 1. *If* $\mathsf{Consistent}_{\mathrm{C11}}(X)$*, then* $\mathsf{Consistent}_{\mathsf{Arfna}}(X)$.

Proof. Straightforward, since by the ConsRFna condition,

$$\{(rf(b), b) \mid \mathsf{isNA}(b) \vee \mathsf{isNA}(rf(b))\} \subseteq \mathsf{hb}$$

and hence Arfna follows from IrrHB. □

Second, this model is not much weaker than C11. More precisely, it only allows more racy behaviours.

Theorem 2. *If* $\mathsf{Consistent}_{\mathsf{Arfna}}(X)$ *and not* $\mathsf{Racy}(X)$*, then* $\mathsf{Consistent}_{\mathrm{C11}}(X)$.

Note that the definition of racy executions, $\mathsf{Racy}(X)$, does not depend on the axioms of the model, and is thus the same for all memory models considered here.

Finally, it is possible to reason about this model as most reasoning techniques on C11 remain true. In particular, in the absence of relaxed accesses, this model is equivalent to the Arf model. We are thus able to use the program logics that have been developed for C11 (namely, RSL [16] and GPS [15]) to also reason about programs in the Arfna model.

However, we found that reordering non-atomic loads past non-atomic stores is forbidden in this model, as shown by the following example:

$$\begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\mathrm{RLX}))\ \{ \\ \quad t = a; \\ \quad b = 1; \\ \quad \mathsf{if}\ (t)\ y.\mathsf{store}(1, \mathrm{RLX}); \\ \} \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (y.\mathsf{load}(\mathrm{RLX})) \\ \quad \mathsf{if}\ (b)\ \{ \\ \quad\quad a = 1; \\ \quad\quad x.\mathsf{store}(1, \mathrm{RLX}); \\ \quad \} \end{array}$$

In this program, the causality cycle does not occur, because for it to happen, an $(\mathsf{hb} \cup rf)$-cycle must also occur between the a and b accesses (and that is ruled out by our axiom). However, if we swap the non-atomic load of a and store of b in the first thread, then the causality cycle becomes possible, and the program is racy. Introducing a race is clearly unsound, so compilers are not allowed to do such reorderings (note that these accesses are non-atomic and adjacent). It is not clear whether such a constraint would be acceptable in C/C++ compilers.

4.2 Correcting the SCReads Axiom

As we have seen in the counterexample of Figure 6, the SCReads axiom places an odd restriction on where a sequentially consistent read can read from. The problem arises from the case where the source of the read is a non-SC write. In this case, the axiom forbids that write to happen before the immediately sc-preceding write to the same location. It may, however, happen before an earlier write in the sc order.

We propose to strengthen the SCReads axiom by requiring there not to be a happens before edge between $rf(b)$ and any same-location write sc-prior to the read, as follows:

$$\begin{array}{l} \forall a, b.\ \ rf(b) = a \wedge \mathsf{isSC}(b) \implies \\ \mathsf{imm}(\mathsf{scr}, a, b) \vee \neg\mathsf{isSC}(a) \wedge \nexists x.\ \mathsf{hb}(a, x) \wedge \mathsf{scr}(x, b) \end{array} \qquad (\mathsf{SCread}')$$

Going back to the program in Figure 6, this stronger axiom rules out reading $r = 1$, a guarantee that is provided by the suggested compilations of C11 atomic accesses to x86/Power/ARM.

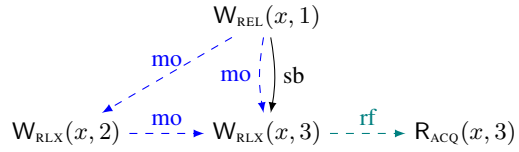
We also considered an even stronger version where instead of hb, the axiom mentions mo, as in the coherence axioms, but have not established its soundness for the suggested compilation of C11 atomic accesses to the Power and ARM architectures.

4.3 Strengthening the Release Sequence Definition

The definition of release sequences in the C11 model is too weak, as shown be the following example.

$$x.\mathsf{store}(2, \mathrm{RLX}); \Bigg\| \begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \mathrm{REL}); \\ x.\mathsf{store}(3, \mathrm{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\mathrm{ACQ}) == 3) \\ \quad \mathsf{print}(y); \end{array}$$

In this program, assuming the test condition holds, the acquire load of x need not synchronise with the release store even though it reads from a store that is sequenced after the release, and hence the program is racy. The reason is that the seemingly irrelevant store of $x.\mathsf{store}(2, \mathrm{RLX})$ can interrupt the release sequence as shown in the following execution snippet.



In the absence, however, of the first thread, the acquire and the release do synchronise and the program is well-defined.

As a fix for the release sequences definition, we propose to replace the definition of release sequences by the least fixed point of the following recursive definition (with respect to $\subseteq$) :

$$\begin{array}{rl} \mathsf{rseq}_{\mathsf{RSnew}}(a, b) \stackrel{\mathsf{def}}{=} & a = b \vee \mathsf{sameThread}(a, b) \wedge mo(a, b) \\ & \vee\ \mathsf{isrmw}(b) \wedge \mathsf{rseq}_{\mathsf{RSnew}}(a, rf(b)) \end{array}$$

Our release sequences are not defined in terms of mo sequences, but rather in terms of rf sequences. Either b should belong to the same thread as a, or there should be a chain of RMW actions reading from one another connecting b to a write in the same thread as a.

In the absence of uninitialised RMW accesses, this change strengthens the semantics. Every consistent execution in the revised model is also consistent in the original model. Despite being a strengthening, it does not affect the compilation results to x86, Power, and ARM. The reason is that release sequences do not play any role on x86, while on Power and ARM the compilation of release writes and fences issues a memory barrier that affects all later writes of the same thread, not just an uninterrupted mo-sequence of such writes.

4.4 Allowing Intra-Thread Synchronisation

A final change is to remove the slightly odd restriction that actions from the same thread cannot synchronise.[5] This change allows us to give meaning to more programs. In the original model, the following program has undefined behaviour:

$$\#\mathsf{define}\ f(x,y)\ (x.\mathsf{CAS}(1,0,\text{ACQ})?(y{+}{+},x.\mathsf{store}(1,\text{REL})):0)$$
$$f(x,y)+f(x,y)$$

That is, although f uses x as a lock to protect the increments of y, and therefore the y accesses could never be adjacent in an interleaving semantics, the model does not treat the x-accesses as synchronising because they belong to the same thread. Thus, the two increments of y are deemed to race with one another.

As we believe that this behaviour is highly suspicious, we have also considered an adaptation of the C11 model, where we set

$$\mathsf{sameThread}_{\mathsf{STnew}}(a,b) \stackrel{\mathsf{def}}{=} sb^{+}(a,b)$$

rather than $\mathsf{tid}(a) = \mathsf{tid}(b)$. We have proved that with the new definition, we can drop the $\neg\mathsf{sameThread}(a,b)$ conjunct from the sw definition without affecting hb.

Since, by the ConsSB axiom, every sb edge has the same thread identifiers, the change also strengthens the model by assigning defined behaviour to more programs.

4.5 Summary of the Models to be Considered

As the four problems are independent and we have proposed fixes to each problem, we consider the product of the fixes:

$$\begin{pmatrix}\mathsf{ConsRFna}\\ \mathsf{Naive}\\ \mathsf{Arfna}\\ \mathsf{Arf}\end{pmatrix} \times \overbrace{\begin{pmatrix}\mathsf{SCorig}\\ \mathsf{SCnew}\end{pmatrix}}^{\S4.2} \times \overbrace{\begin{pmatrix}\mathsf{RSorig}\\ \mathsf{RSnew}\end{pmatrix}}^{\S4.3} \times \overbrace{\begin{pmatrix}\mathsf{STorig}\\ \mathsf{STnew}\end{pmatrix}}^{\S4.4}$$

We use tuple notation to refer to the individual models. For example, we write $(\mathsf{ConsRFna}, \mathsf{SCorig}, \mathsf{RSorig}, \mathsf{STorig})$ for the model corresponding to the 2011 C and C++ standards.

In Sections 5, 6 and 7, we show that the RSnew and STnew components, despite further constraining the set of consistent executions, permit all the transformations allowed by the RSorig and STorig components respectively.

5. Basic Metatheory of the Corrected C11 Models

In this section, we develop basic metatheory of the various corrections to the C11 model, which will assist us in verifying the program transformations in the next sections. The subsection headings mention the Coq source file containing the corresponding proofs.

5.1 Semiconsistent Executions [cmon.v]

We observe that in the monotone models (see Definition 3) the happens-before relation appears negatively in all axioms except for the $\Longleftarrow$ direction of the ConsRFdom axiom. It turns out, however, that this apparent lack of monotonicity with respect to happens-before does not cause problems as it can be circumvented by the following lemma.

[5] This restriction breaks monotonicity in the presence of consume reads.

Definition 1 (Semiconsistent Executions)**.** *An execution is* semiconsistent *with respect to a model M iff it satisfies all the axioms of the model except for the $\Longleftarrow$ direction of the* ConsRFdom *axiom.*

Lemma 1 (Semiconsistency)**.** *Given a semiconsistent execution (O, rf, mo, sc) with respect to M for $M \neq (\mathsf{ConsRFna}, \_, \_, \_)$, there exists $rf' \subseteq rf$ such that (O, rf', mo, sc) is consistent with respect to M.*

Proof. We pick rf' as the greatest fixed point of the functional:

$$F(rf)(x) := \begin{cases} rf(x) & \text{if } \exists y.\ \mathsf{hb}(y,x) \wedge \mathsf{iswrite}_{\mathsf{loc}(x)}(y) \\ \text{undefined} & \text{otherwise} \end{cases}$$

that is smaller than rf (with respect to $\subseteq$). Such a fixed point exists by Tarski's theorem as the function is monotone. By construction, it satisfies the ConsRFdom axiom, while all the other axioms follow easily because they are antimonotone in rf. □

5.2 Monotonicity [cmon.v]

We move on to proving the most fundamental property of the corrected models: *monotonicity*, saying that if we weaken the access modes of some of the actions of a consistent execution and/or remove some sb edges, the execution remains consistent.

Definition 2 (Access type ordering)**.** *Let $\sqsubseteq : \mathcal{P}(MO \times MO)$ be the least reflexive and transitive relation containing* RLX $\sqsubseteq$ REL $\sqsubseteq$ REL-ACQ $\sqsubseteq$ SC*, and* RLX $\sqsubseteq$ ACQ $\sqsubseteq$ REL-ACQ*.*

We lift the access order to memory actions, $\sqsubseteq : \mathcal{P}(EV \times EV)$, by letting $act \sqsubseteq act$, $\mathsf{R}_X(\ell,v) \sqsubseteq \mathsf{R}_{X'}(\ell,v)$, $\mathsf{W}_X(\ell,v) \sqsubseteq \mathsf{W}_{X'}(\ell,v)$, $\mathsf{C}_X(\ell,v,v') \sqsubseteq \mathsf{C}_{X'}(\ell,v,v')$, $\mathsf{F}_X \sqsubseteq \mathsf{F}_{X'}$, and $\mathsf{skip} \sqsubseteq \mathsf{F}_{X'}$, whenever $X \sqsubseteq X'$. We also lift this order to functions pointwise: $lab \sqsubseteq lab'$ iff $\forall a.\ lab(a) \sqsubseteq lab'(a)$.

Monotonicity does not hold for all the models we consider, but only after some necessary fixes have been applied. We call those corrected models monotone.

Definition 3. *We call a memory model, M,* monotone*, iff $M \neq (\mathsf{ConsRFna}, \_, \_, \_)$ and $M \neq (\_, \mathsf{SCorig}, \_, \_)$.*

Theorem 3 (Monotonicity)**.** *For a monotone memory model M, if $\mathsf{Consistent}_M(lab, sb, asw, rf, mo, sc)$ and $lab' \sqsubseteq lab$ and $sb' \subseteq sb$, then there exist $rf' \subseteq rf$ and $sc' \subseteq sc$ such that $\mathsf{Consistent}_M(lab', sb', asw, rf', mo, sc')$.*

Proof sketch. From Lemma 1, it suffices to prove that the execution $(lab', sb', asw, rf, mo, sc')$ is semiconsistent. We can show this by picking:

$$sc'(x,y) \stackrel{\mathsf{def}}{=} sc(x,y) \wedge \mathsf{isSC}(lab'(x)) \wedge \mathsf{isSC}(lab'(y))$$

We can show that $\mathsf{hb}' \subseteq \mathsf{hb}$, and then all the axioms of the model follow straightforwardly. □

From Theorem 3, we can immediately show the soundness of three simple kinds of program transformations:

- *Expression evaluation order linearisation* and *sequentialisation*, because in effect they just add sb edges to the program;
- *Strengthening of the memory access orders*, such as replacing a relaxed load by an acquire load; and
- *Fence insertion*, because this can be seen as replacing a skip node (an empty fence) by a stronger fence.

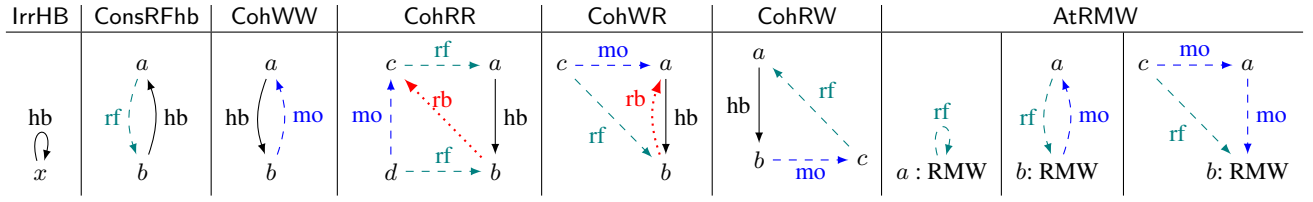


Figure 8. Executions violating the coherence axioms: all contain a cycle in $\{(a,b) \in \mathsf{hb} \mid \mathsf{loc}(a) = \mathsf{loc}(b)\} \cup \mathsf{com}$.

5.3 Alternative Presentation of the Coherence Axioms [coherence.v]

Next, we consider equivalent alternative presentations of the coherence axioms, which can be used to gain better understanding of the models and to simplify some proofs about them.

Since mo is a total order on writes to the same location, and hb is irreflexive, the CohWW axiom is actually equivalent to the following one:

$$\forall a, b, \ell.\ \mathsf{hb}(a,b) \wedge \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b) \implies mo(a,b) \qquad \text{(ConsMOhb)}$$

The equivalence can be derived by a case analysis on how mo orders a and b. (For what it is worth, the C/C++ standards as well as the formal model of Batty et al. [4] include both axioms even though, as we show, one of them is redundant.)

Next, we show that the coherence axioms can be restated in terms of a single acyclicity axiom. To state this axiom, we need some auxiliary definitions. We say that a read, a, *reads before*[6] a different write, b, denoted $\mathsf{rb}(a,b)$, if and only if $a \neq b$ and $mo(rf(a), b)$. (Note that we need the $a \neq b$ condition because RMW actions are simultaneously both reads and writes.) We define the communication order, com, as the union of the modification order, the reads-from map, and the reads-before relation.

$$\mathsf{rb}(a,b) \stackrel{\mathsf{def}}{=} mo(rf(a), b) \wedge a \neq b$$
$$\mathsf{com}(a,b) \stackrel{\mathsf{def}}{=} mo(a,b) \vee rf(b) = a \vee \mathsf{rb}(a,b)$$

In essence, for every location ℓ, com^+ relates the writes to and initialised reads from that location, ℓ. Except for uninitialised reads and loads reading from the same write, com^+ is a total order on all accesses of a given location, ℓ.

We observe that all the violations of the coherence axioms are cyclic in $\{(a,b) \in \mathsf{hb} \mid \mathsf{loc}(a) = \mathsf{loc}(b)\} \cup \mathsf{com}$ (see Figure 8). This is not accidental: from Shasha and Snir [14] we know that any execution acyclic in $\mathsf{hb} \cup \mathsf{com}$ is sequentially consistent, and coherence essentially guarantees sequential consistency on a per-location basis.

Based on this observation, we consider the following axiom stating that the union of hb restricted to relate same-location actions and com is acyclic.

$$\mathsf{acyclic}(\{(a,b) \in \mathsf{hb} \mid \mathsf{loc}(a) = \mathsf{loc}(b)\} \cup \mathsf{com}) \qquad \text{(Coh)}$$

This axiom is equivalent to the conjunction of seven C11 axioms as shown in the following theorem:

Theorem 4. *Assuming* ConsMO *and* ConsRF *hold, then*

$$\mathsf{Coh} \iff \begin{pmatrix}\mathsf{IrrHB} \wedge \mathsf{ConsRFhb} \wedge \mathsf{CohWW} \wedge \\ \mathsf{CohRR} \wedge \mathsf{CohWR} \wedge \mathsf{CohRW} \wedge \mathsf{AtRMW}\end{pmatrix}.$$

Proof (sketch). In the ($\Rightarrow$) direction, it is easy to see that all the coherence axiom violations exhibit cycles (see Fig. 8). In the other direction, careful analysis reveals that these are the only possible cycles—any larger ones can be shortened as mo is a total order. □

[6] Alglave et al. [1] call this relation "from-read."

Although the alternative presentation of the coherence axioms developed here is much more concise than the original one, it is of limited use in verifying the program transformations, because we need to reason about yet another transitive closure (besides hb).

5.4 Prefixes of Consistent Executions [prefixes.v]

Another basic property we would like to hold for a memory model is for any prefix of a consistent execution to also form a consistent execution. Such a property would allow, for instance, to execute programs in a stepwise operational fashion generating the set of consistent executions along the way. It is also very useful in proving the DRF theorem and the validity of certain optimisations by demonstrating an alternative execution prefix of the program that contradicts the assumptions of the statement to be proved (e.g., by containing a race).

One question remains: Under which relation should we be considering execution prefixes? To make the result most widely applicable, we want to make the relation as small as possible, but at the very least we must include (the dependent part of) the program order, sb and asw, in order to preserve the program semantics, as well as the reads from relation, rf, in order to preserve the memory semantics. Moreover, in the case of RSorig models, as shown in the example from Section 4.3, we must also include mo-prefixes.

Definition 4 (Prefix closure)**.** *We say that a relation, R, is* prefix closed *on a set, S, iff* $\forall a, b.\ R(a,b) \wedge b \in S \implies a \in S$.

Definition 5 (Prefix opsem)**.** *An opsem (lab', sb', asw') is a* prefix *of another opsem (lab, sb, asw) iff $lab' \subseteq lab$, $sb' = sb \cap (\mathsf{dom}(lab') \times \mathsf{dom}(lab'))$, $asw' = asw \cap (\mathsf{dom}(lab') \times \mathsf{dom}(lab'))$, and sb and asw are prefix closed on $\mathsf{dom}(lab')$.*

Theorem 5. *Given a model M, opsems O and $O' = (lab', \_, \_)$ and a witness $W = (rf, mo, sc)$, if $\mathsf{Consistent}_M(O, W)$ and O' is a prefix of O and $\{(a,b) \mid rf(b) = a\}$ is prefix-closed on $\mathsf{dom}(lab')$ and either $M = (\_, \_, \mathsf{RSnew}, \_)$ or mo is prefix-closed on $\mathsf{dom}(lab')$, then there exists W' such that $\mathsf{Consistent}_M(O', W')$.*

Proof (sketch). We pick W' to be W restricted to the actions in $\mathsf{dom}(lab')$. Then, we show $\mathsf{hb}' = \mathsf{hb} \cap (\mathsf{dom}(lab') \times \mathsf{dom}(lab'))$ and that each consistency axiom is preserved. □

To be able to use such a theorem in proofs, the relation defining prefixes should be acyclic. This is because we would like there to exist a maximal element in the relation, which we can remove from the execution and have the resulting execution remain consistent. This means that, for example, in the Arf model, we may want to choose $\mathsf{hb} \cup rf$ as our relation. Unfortunately, however, this does not quite work in the RSorig model and requires switching to the RSnew model.

6. Verifying Instruction Reorderings

We proceed to the main technical results of the paper, namely the proofs of validity for the various program transformations. Having already discussed the simple monotonicity-based ones, we now

$\downarrow a \setminus b \rightarrow$	$\mathsf{R}_{\mathrm{NA}\mid\mathrm{RLX}\mid\mathrm{ACQ}}(\ell')$	$\mathsf{R}_{\mathrm{SC}}(\ell')$	$\mathsf{W}_{\mathrm{NA}}(\ell')$	$\mathsf{W}_{\mathrm{RLX}}(\ell')$	$\mathsf{W}_{\mathrm{REL}\mid\mathrm{SC}}(\ell')$	$\mathsf{C}_{\mathrm{RLX}\mid\mathrm{ACQ}}(\ell')$	$\mathsf{C}_{\sqsupseteq\mathrm{REL}}(\ell')$	$\mathsf{F}_{\mathrm{ACQ}}$	$\mathsf{F}_{\mathrm{REL}}$
$\mathsf{R}_{\mathrm{NA}}(\ell)$	✓Thm.6	✓Thm.6	✓Thm.6/?/✗C	✓Thm.6/?/✗C	✗A.1	✓Thm.6/?/✗C	✗A.2	✓Thm.6	✗A.7
$\mathsf{R}_{\mathrm{RLX}}(\ell)$	✓Thm.6	✓Thm.6	✓Thm.6/?/✗C	✓Thm.6/✗B/✗C	✗A.1	✓Thm.6/✗B/✗C	✗A.2	✗A.6	✗A.7
$\mathsf{R}_{\mathrm{ACQ}\mid\mathrm{SC}}(\ell)$	✗A.3	✗A.3	✗A.3	✗A.3	✗A.1	✗A.3	✗A.3	✓Thm.7	✗A.7
$\mathsf{W}_{\mathrm{NA}\mid\mathrm{RLX}\mid\mathrm{REL}}(\ell)$	✓Thm.6	✓Thm.6	✓Thm.6	✓Thm.6	✗A.1	✓Thm.6	✗A.2	✓Thm.6	✗A.7
$\mathsf{W}_{\mathrm{SC}}(\ell)$	✓Thm.6	✗A.4	✓Thm.6	✓Thm.6	✗A.1	✓Thm.6	✗A.2	✓Thm.6	✗A.7
$\mathsf{C}_{\mathrm{RLX}\mid\mathrm{REL}}(\ell)$	✓Thm.6	✓Thm.6	✓Thm.6/?/✗C	✓Thm.6/✗B/✗C	✗A.1	✓Thm.6/✗B/✗C	✗A.2	✗A.6	✗A.7
$\mathsf{C}_{\sqsupseteq\mathrm{ACQ}}(\ell)$	✗A.3	✗A.3	✗A.3	✗A.3	✗A.1	✗A.3	✗A.2	✓Thm.7	✗A.7
$\mathsf{F}_{\mathrm{ACQ}}$	✗A.5	✗A.5	✗A.5	✗A.5	✗A.5	✗A.5	✗A.5	=	✗A.9
$\mathsf{F}_{\mathrm{REL}}$	✓Thm.6	✓Thm.6	✓Thm.6	✗A.8	✓Thm.8	✗A.8	✓Thm.8	✓Thm.6	=

Table 1. Allowed parallelisations $a\,;b \rightsquigarrow a \parallel b$ in monotone models, and therefore reorderings $a\,;b \rightsquigarrow b\,;a$. We assume $\ell \neq \ell'$. Where multiple entries are given, these correspond to Naive/Arf/Arfna. Ticks cite the appropriate theorem, crosses the counterexample. Question marks correspond to unknown cases. (We conjecture these are valid, but need a more elaborate definition of opsem prefixes to prove.)

focus on transformations that reorder adjacent instructions that do not access the same location.

We observe that for monotone models, a reordering can be decomposed into a parallelisation followed by a linearisation:

$$a\,;b \overset{\text{under some conditions}}{\rightsquigarrow} a \parallel b \overset{\text{By Theorem 3}}{\rightsquigarrow} b\,;a$$

We summarise the allowed reorderings/parallelisations in Table 1. There are two types of allowed updates:

(§6.1) "Roach motel" instruction reorderings, and

(§6.2) Fence reorderings against the roach motel semantics.

For the negative cases, we provide counterexamples in the appendix.

6.1 Roach Motel Instruction Reorderings [reorder.v]

The "roach motel" reorderings are the majority among those in Table 1 and are annotated by '✓Thm.6.' This category contains all reorderable pairs of actions, a and b, that are adjacent according to sb and asw. We say that two actions a and b are *adjacent* according to a relation R if (1) every action directly reachable from b is directly reachable from a; (2) every action directly reachable from a, except for b, is also directly reachable by b; (3) every action that reaches a directly can also reach b directly; and (4) every action that reaches b directly, except for a, can also reach a directly. Note that adjacent actions are not necessarily related by R.

Definition 6 (Adjacent actions)**.** *Two actions a and b are* adjacent *in a relation R, written $\mathsf{Adj}(R, a, b)$, if for all c, we have:*
(1) $R(b,c) \Rightarrow R(a,c)$, and (2) $R(a,c) \wedge c \neq b \Rightarrow R(b,c)$, and
(3) $R(c,a) \Rightarrow R(c,b)$, and (4) $R(c,b) \wedge c \neq a \Rightarrow R(c,a)$.

Two actions a and b are *reorderable* if (1) they belong to the same thread; (2) they do not access the same location, (3) a is not an acquire access or fence, (4) b is not a release access or fence, (5) if the model is based on Arfna or Arf and a is a read, then b is not a write, (6) if a is a release fence, then b is not an atomic write, (7) if b is an acquire fence, then a is not an atomic read, and (8) a and b are not both SC actions.

Definition 7 (Reorderable pair)**.** *Two distinct actions a and b are* reorderable *in a memory model M, written $\mathsf{Reord}_M(a,b)$, if*
(1) $\mathsf{tid}(a) = \mathsf{tid}(b)$
and (2) $\mathsf{loc}(a) \neq \mathsf{loc}(b)$
and (3) $\neg\mathsf{isAcq}(a)$
and (4) $\neg\mathsf{isRel}(b)$
and (5i) $\neg(M = (\mathsf{Arfna}, \_, \_, \_) \wedge \mathsf{isread}(a) \wedge \mathsf{iswrite}(b))$
and (5ii) $\neg(M = (\mathsf{Arf}, \_, \_, \_) \wedge \mathsf{isread}(a) \wedge \mathsf{iswrite}(b))$
and (6) $\neg(\mathsf{isFenceRel}(a) \wedge \mathsf{isAtomicWrite}(b))$
and (7) $\neg(\mathsf{isAtomicRead}(a) \wedge \mathsf{isFenceAcq}(b))$
and (8) $\neg(\mathsf{isSC}(a) \wedge \mathsf{isSC}(b))$

Theorem 6. *For a monotone M, if $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(sb, a, b)$, $\mathsf{Adj}(asw, a, b)$, and $\mathsf{Reord}_M(a, b)$, there exists W',*
(i) $\mathsf{Consistent}_M(lab, sb \cup \{(a,b)\}, asw, W')$,
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$, and
(iii) $\mathsf{Racy}_M(lab, sb, W) \Rightarrow \mathsf{Racy}_M(lab, sb \cup \{(a,b)\}, W')$.

Proof (sketch). By Lemma 1, it suffices to show semiconsistency. The main part is then proving that $\mathsf{hb} = \mathsf{hb}' \cup \{(a,b)\}$, where hb (resp. hb') denotes the happens-before relation in $(lab, sb \cup \{(a,b)\}, asw, W)$ (resp. (lab, sb, asw, W)). Hence these transformations do not really affect the behaviour of the program, and the preservation of each axiom is a simple corollary. □

The proof of Theorem 6 (and similarly those of Theorems 7 and 8 in Section 6.2), require only conditions (1) and (3) from the definition of adjacent actions; Conditions (2) and (4) are, however, important for the theorems of Section 7.1, and so, for simplicity, we presented a single definition of when two actions are adjacent.

6.2 Non-RM Reorderings with Fences [fenceopt.v]

The second class is comprised of a few valid reorderings between a fence and a memory access of the same or stronger type. In contrast to the previous set of transformations, these new ones remove some synchronisation edges but only to fence instructions. As fences do not access any data, there are no axioms constraining these incoming and outgoing synchronisation edges to and from fences, and hence they can be safely removed.

Theorem 7. *For a monotone M, if $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(sb, a, b)$, $\mathsf{Adj}(asw, a, b)$, $\mathsf{isAcq}(a)$, and $lab(b) = \mathsf{F}_{\mathrm{ACQ}}$, then*
(i) $\mathsf{Consistent}_M(lab, sb \cup \{(a,b)\}, asw, W)$ and
(ii) $\mathsf{Racy}_M(lab, sb, W) \Rightarrow \mathsf{Racy}_M(lab, sb \cup \{(a,b)\}, W)$.

Theorem 8. *For a monotone M, if $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(sb, a, b)$, $\mathsf{Adj}(asw, a, b)$, $lab(a) = \mathsf{F}_{\mathrm{REL}}$ and $\mathsf{isRel}(b)$, then*
(i) $\mathsf{Consistent}_M(lab, sb \cup \{(a,b)\}, asw, W)$ and
(ii) $\mathsf{Racy}_M(lab, sb, W) \Rightarrow \mathsf{Racy}_M(lab, sb \cup \{(a,b)\}, W)$.

That is, we can reorder an acquire command over an acquire fence, and a release fence over a release command.

$$\mathsf{acq}\,;\mathsf{F}_{\mathrm{ACQ}} \overset{\text{Thm. 7 \& 3}}{\rightsquigarrow} \mathsf{F}_{\mathrm{ACQ}}\,;\mathsf{acq} \qquad \mathsf{F}_{\mathrm{REL}}\,;\mathsf{rel} \overset{\text{Thm. 8 \& 3}}{\rightsquigarrow} \mathsf{rel}\,;\mathsf{F}_{\mathrm{REL}}$$

7. Verifying Instruction Eliminations

Next, we consider eliminating redundant memory accesses, as would be performed by standard optimisations such as common subexpression elimination or constant propagation. To simplify the presentation (and the proofs), in §7.1, we first focus on the cases where eliminating an instruction is justified by an adjacent instruction (e.g., a repeated read, or an immediately overwritten write). In §7.2, we will then tackle the general case.

Consider the following program:

$$x = y = 0;$$
$$\begin{array}{l|l} y.\mathsf{store}(1, \text{RLX}); & t_2 = x.\mathsf{CAS}(0, 1, \text{ACQ}); \\ \mathsf{fence}(\text{REL}); & t_3 = y.\mathsf{load}(\text{RLX}); \\ t_1 = x.\mathsf{load}(\text{RLX}); & \\ x.\mathsf{store}(t_1, \text{RLX}); & \end{array}$$
$$t_4 = x.\mathsf{load}(\text{RLX});$$

The outcome

$$t_1 = 0,\ t_2 = 0,\ t_3 = 0,\ t_4 = 1$$

is not possible. If, however, we remove the $x.\mathsf{store}(\ldots)$ then this outcome becomes possible.

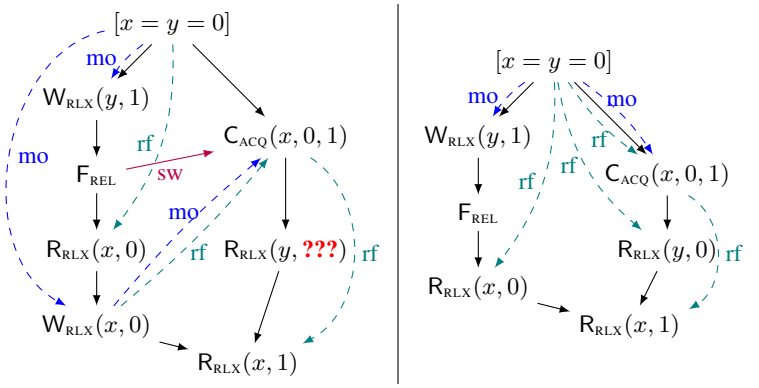


Figure 9. Counterexample for the 'write after read' elimination optimisation.

7.1 Elimination of Redundant Adjacent Accesses [celim.v]

Repeated Read. The first transformation we consider is eliminating the second of two identical adjacent loads from the same location. Informally, if two loads from the same location are adjacent in program order, it is possible that both loads return the value written by the same store. Therefore, if the loads also have the same access type, the additional load will not introduce any new synchronisation, and hence we can always remove one of them, say the second.

$$\mathsf{R}_X(\ell, v)\,;\mathsf{R}_X(\ell, v) \rightsquigarrow \mathsf{R}_X(\ell, v)\,;\mathsf{skip} \qquad \text{(RAR-adj)}$$

Formally, we say that a and b are adjacent if a sequenced before b and they adjacent according to sb and asw. That is:

$$\mathsf{Adj}(a, b) \stackrel{\text{def}}{=} sb(a, b) \wedge \mathsf{Adj}(sb, a, b) \wedge \mathsf{Adj}(asw, a, b)$$

We can prove the following theorem:

Theorem 9 (RaR-Adjacent)**.** *For a monotone memory model M, if $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(a, b)$, $lab(a) = \mathsf{R}_X(\ell, v)$, $lab(b) = \mathsf{skip}$, and $lab' = lab[b := \mathsf{R}_X(\ell, v)]$, then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$,
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$, *and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$.

This says that any consistent execution of the target of the transformation can be mapped to one of the program prior to the transformation. To prove this theorem, we pick $rf' := rf[b \mapsto rf(a)]$ and extend the sc order to include the (a, b) edge in case $X = \text{SC}$.

Read after Write. Similarly, if a load immediately follows a store to the same location, then it is always possible for the load to get the value from that store. Therefore, it is always possible to remove the load.

$$\mathsf{W}_X(\ell, v)\,;\mathsf{R}_Y(\ell, v) \rightsquigarrow \mathsf{W}_X(\ell, v)\,;\mathsf{skip} \qquad \text{(RAW-adj)}$$

Formally, we prove the following theorem:

Theorem 10 (RaW-Adjacent)**.** *For a monotone memory model M, if $\mathsf{Consistent}_M(lab, sb, asw, W)$, $\mathsf{Adj}(a, b)$, $lab(a) = \mathsf{W}_X(\ell, v)$, $lab(b) = \mathsf{skip}$, $lab' = lab[b := \mathsf{R}_Y(\ell, v)]$ and either $Y \neq \text{SC}$ or $M \neq (\_, \_, \mathsf{STorig}, \_)$, then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$,
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$, *and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$.

Overwritten Write. If two stores to the same location are adjacent in program order, it is possible that the first store is never read by any thread. So, if the stores have the same access type we can always remove the first one. That is, we can do the transformation:

$$\mathsf{W}_X(\ell, v')\,;\mathsf{W}_X(\ell, v) \rightsquigarrow \mathsf{skip}\,;\mathsf{W}_X(\ell, v) \qquad \text{(OW-adj)}$$

To prove the correctness of the transformation, we prove the following theorem saying that any consistent execution of the target program corresponds to a consistent execution of the source program.

Theorem 11 (OW-Adjacent)**.** *For a monotone memory model M, if $\mathsf{Consistent}_M(lab, sb, asw, W)$ and $\mathsf{Adj}(a, b)$ and $lab(a) = \mathsf{skip}$ and $lab(b) = \mathsf{W}_X(\ell, v)$ and $lab' = lab[a := \mathsf{W}_X(\ell, v')]$ and $\ell \neq \texttt{world}$, then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$,
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$, *and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$.

Note that as a special case of this transformation, if the two stores are identical, we can alternatively remove the second one:

$$\mathsf{W}_X(\ell,v)\,;\mathsf{W}_X(\ell,v) \overset{\text{(OW-adj)}}{\rightsquigarrow} \mathsf{skip}\,;\mathsf{W}_X(\ell,v) \overset{\text{reorder}}{\rightsquigarrow} \mathsf{W}_X(\ell,v)\,;\mathsf{skip}$$

Write after Read. The next case to consider is what happens when a store immediately follows a load to the same location, and writes the same value as observed by the load.

$$\mathsf{R}_X(\ell, v)\,;\mathsf{W}_{\text{RLX}}(\ell, v) \rightsquigarrow \mathsf{R}_X(\ell, v)\,;\mathsf{skip}$$

In this case, can we eliminate the redundant store?

Well, actually, no, we cannot. Figure 9 shows a program demonstrating that the transformation is unsound. The program uses an atomic read-modify-write instruction, CAS, to update x, in parallel to the thread that reads x to be 0 and then writes back 0 to x.

Consider an execution in which the load of x reads 0 (enforced by $t_1 = 0$), the CAS succeeds (enforced by $t_2 = 0$) and is in modification order after the store to x (enforced by $t_4 = 1$ and the CohWR axiom). Then, because of the atomicity of CAS (axiom AtRMW), the CAS must read from the first thread's store to x, inducing a synchronisation edge between the two threads. As a result, by the CohWR axiom, the load of y cannot read the initial value (i.e., necessarily $t_3 \neq 0$).

If, however, we remove the store to x from the left thread, the outcome in question becomes possible as indicated by the second execution shown in Figure 9.

In essence, this transformation is unsound because we can force a operation to be ordered between the load and the store (according to the communication order). In the aforementioned counterexample, we achieved this by the atomicity of RMW instructions.

We can also construct a similar counterexample without RMW operations, by exploiting SC fences, a more advanced feature of C11, which for simplicity we do not model in this paper.

7.2 Elimination of Redundant Non-Adjacent Operations

We proceed to the general case, where the removed redundant operation is in the same thread as the operation justifying its removal, but not necessarily adjacent to it.

In the appendix, we have proved three theorems generalising the theorems of Section 7.1. The general set up is that we consider two actions a and b in program order (i.e., $sb(a, b)$), accessing the same location ℓ (i.e., $\mathsf{loc}(a) = \mathsf{loc}(b) = \ell$), without any intermediate actions accessing the same location (i.e., $\nexists c.\ sb(a, c) \wedge sb(c, b) \wedge \mathsf{loc}(c) = \ell$). In addition, for the generalisations of Theorems 9 and 10 (respectively, of Theorem 11), we also require there to be no acquire (respectively, release) operation in between.

Under these conditions, we can reorder the action to be eliminated (using Theorem 6) past the intermediate actions to become adjacent to the justifying action, so that we can apply the adjacent elimination theorem. Then we can reorder the resulting "skip" node back to the place the eliminated operation was initially.

8. Related Work

The C11 model was introduced by the 2011 revisions of the C and C++ standards [8, 7]. A rigorous mathematical formalisation of the C11 memory model was given by Batty et al. [4] and was later extended to cover read-modify-write and fence instructions [13].

Sample compilation schemes for atomic accesses have been proved correct both for the x86-TSO architecture [4] and for the Power/ARM architecture [3, 13]. The aim here was to study how expensive it is to enforce the intended C11 semantics on widespread architectures: the idealised compiler considered naively applies a one-to-one mapping from C memory accesses to machine memory accesses, attempting no optimisations at all.

Out-of-thin-air behaviours are being recognised as the most troublesome corner of the design of modern language memory models. The Java memory model [10] tried to effectively prohibit out-of-thin-air results in its specification. Complicated causality rules were introduced for this purpose, which turned out to forbid some program transformations that the reference HotSpot compiler actually performs [19]. The work of Ševčík is closely bound to the specificities of the Java memory model, and his counterexamples cannot be translated to C. The existence of causality cycles is vaguely acknowledged in the C and C++ language standards, and is stated clearly in [4, Sec. 4]. Since then, independent lines of research, including program logics [2, 16] and model checkers [12] bumped into issues related to causality cycles; it is today acknowledged that code verification is infeasible in their presence.

It turns out that it is very difficult to define a language memory model that both allows programmers to take full advantage of weakly-ordered memory accesses but still correctly disallows out-of-thin-air results. The quest for an updated model for Java is still open; it is the objective of the OpenJDK JEP 188 but no concrete design has yet been proposed. Surprisingly, the simpler requirements of the C language did not lead to a quick fix. A brute-force solution preventing relaxed loads from being reordered with subsequent relaxed stores has been proposed by Boehm [5, 6] and by Vafeiadis and Narayan [16], which we also studied in this paper. This condition imposes a non-negligible cost on some architectures (ARM, GPUs) and its adoption in the standard is unclear.

As already mentioned, the study of correctness of compiler optimisations in an idealised DRF model was done by Ševčík [18] and later adapted to C11 for some optimisations by Morisset et al. [11]. This paper uses the same setup but explores in a far greater depth the interaction between optimisations and low-level atomic accesses, with the surprising results presented.

The certified compilers CompCert [9] and CompCertTSO [20] (the latter extending an earlier version of the former to concurrent shared memory programming with a TSO-based memory semantics) share the same memory model for all the intermediate languages. A hypothetical CompCertC11 compiler could not use the C11 memory model for this purpose: expression linearisation is performed in the first pass of CompCert and, as we have shown, it cannot be proved correct in the C11 model. Unless the C11 model is fixed along the lines we discussed, the hypothetical CompCertC11 would have to expand the compilation of atomic accesses immediately after parsing, and then reason in terms of the target architecture memory model. This is not an option for an efficiently implementable, general purpose, programming language: hardware memory models are not DRF models and prevent most optimisations on memory accesses.

Acknowledgements

This work is supported by the EC FP7 FET project ADVENT and the ANR grant WMC (ANR-11-JS02-011). We would like to thank Brian Demsky, Jean-Jacques Lévy, Peter Sewell and the anonymous reviewers for their helpful feedback. Morisset was supported by a Google European PhD Fellowship.

References

[1] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *TOPLAS*, 36(2):7:1–7:74, 2014.

[2] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

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

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[5] H.-J. Boehm. N3710: Specifying the absence of "out of thin air" results, 2013. Available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html`.

[6] H.-J. Boehm and B. Demsky. Outlawing ghosts: avoiding out-of-thin-air results. In *MSPC*, 2014.

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

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

[9] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[10] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[11] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, 2013.

[12] B. Norris and B. Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA*, 2013.

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

[14] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *TOPLAS*, 10(2):282–312, 1988.

[15] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak-memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.

[16] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

[17] J. Ševčík. The Sun Hotspot JVM does not conform with the Java memory model. Technical Report EDI-INF-RR-1252, School of Informatics, University of Edinburgh, 2008.

[18] J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[19] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[20] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.

A. Unsafe Reordering Examples for All Models

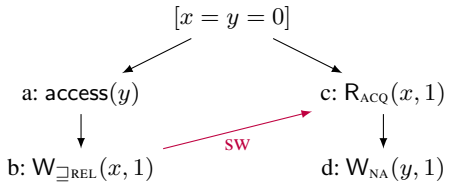
We present a series of counterexamples showing that certain adjacent instruction permutations are invalid.

A.1 $\mathsf{access}; \mathsf{W}_{\mathbf{REL}|\mathbf{SC}} \rightsquigarrow \mathsf{W}_{\mathbf{REL}|\mathbf{SC}}; \mathsf{access}$

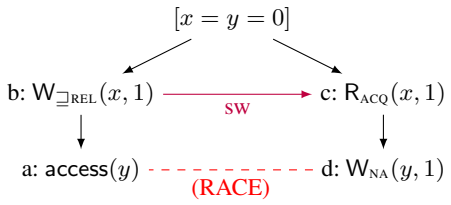
Our first counterexample shows that we cannot reorder a memory access past a release store. Consider the following program:

$$\begin{array}{l} \mathsf{access}(y); \\ x.\mathsf{store}(1, \sqsupseteq\text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{ACQ})) \\ \quad y = 1; \end{array}$$

This program is race-free as illustrated by the following execution:



If we reorder the instructions in the first thread, the following racy execution is possible.



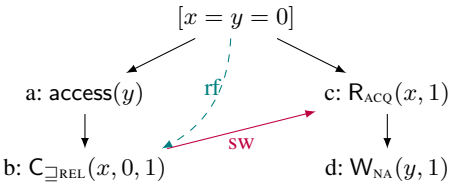
This means that the reordering introduced new behaviour and is therefore unsound.

A.2 $\mathsf{access}; \mathsf{C}_{\sqsupseteq\mathbf{REL}} \rightsquigarrow \mathsf{C}_{\sqsupseteq\mathbf{REL}}; \mathsf{access}$

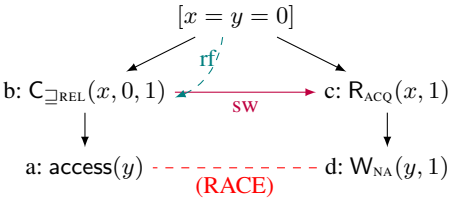
We can construct a similar example to show that reordering an access past a release RMW is also unsound. Consider the program:

$$\begin{array}{l} \mathsf{access}(y); \\ x.\mathsf{CAS}(0, 1, \sqsupseteq\text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{ACQ})) \\ \quad y = 1; \end{array}$$

This program is race free because $y = 1$ can happen only after the load of x reads 1 and synchronises with the successful CAS.



If we reorder the instructions in the first thread, the following racy execution is possible, and therefore the transformation is unsound.

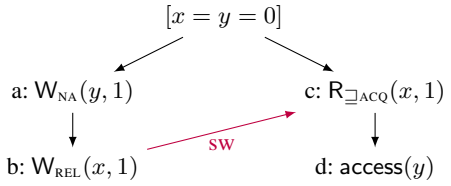


A.3 $\mathsf{R}_{\sqsupseteq\mathbf{ACQ}}; \mathsf{access} \rightsquigarrow \mathsf{access}; \mathsf{R}_{\sqsupseteq\mathbf{ACQ}}$

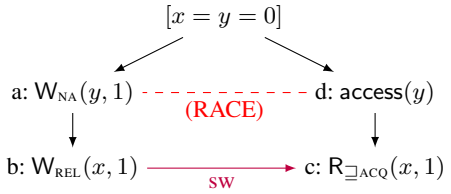
Next, we construct dual examples showing that reordering an acquire read past a memory access is unsound. Consider the following example, where on the second thread, we use a busy loop to wait until the store of x has happened.

$$\begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{while}\ (1 \neq x.\mathsf{load}(\sqsupseteq\text{ACQ})) \\ \mathsf{access}(y); \end{array}$$

The program is race-free as demostrated by the following execution:



Reordering the $\mathsf{access}(y)$ above the load of x is unsound, because then the following execution would be possible:



This execution contains racy accesses to y, and therefore exhibits more behaviours than the original program does.

This shows that reordering an acquire load past a memory access is unsound. In a similar fashion, we can show that reordering an acquire RMW past a memory access is also unsound. All we have to do is consider the following program:

$$\begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{while}\ (\neg x.\mathsf{CAS}(1, 2, \sqsupseteq\text{ACQ})); \\ \mathsf{access}(y); \end{array}$$

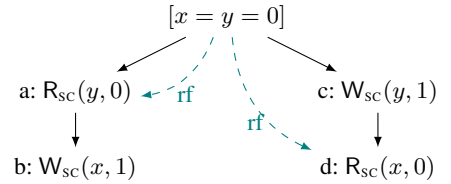
where we have replaced the acquire load by an acquire CAS.

A.4 $\mathsf{W}_{\mathbf{SC}}; \mathsf{R}_{\mathbf{SC}} \rightsquigarrow \mathsf{R}_{\mathbf{SC}}; \mathsf{W}_{\mathbf{SC}}$

Consider the store-buffering program with SC accesses:

$$\begin{array}{l} x.\mathsf{store}(1, \text{SC}); \\ r_1 = y.\mathsf{load}(\text{SC}); \end{array} \Bigg\| \begin{array}{l} y.\mathsf{store}(1, \text{SC}); \\ r_2 = x.\mathsf{load}(\text{SC}); \end{array}$$

Since SC accesses are totally ordered, the outcome $r_1 = r_2 = 0$ is not possible. This outcome, however, can be obtained by reordering the actions of the left thread.

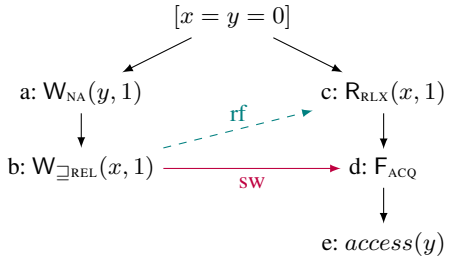


A.5 $\mathsf{F}_{\mathbf{ACQ}}; \mathsf{access} \rightsquigarrow \mathsf{access}; \mathsf{F}_{\mathbf{ACQ}}$

Consider the following program

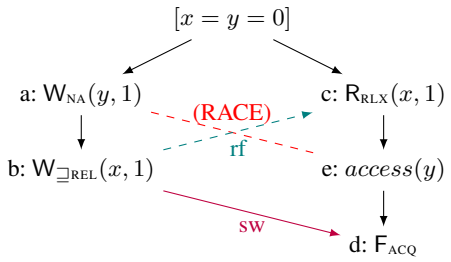
$$\begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{while}\ (1 \neq x.\mathsf{load}(\text{RLX})); \\ \mathsf{fence}(\text{ACQ}); \\ \mathsf{access}(y); \end{array}$$

The execution trace is as follows



In this execution $(a, e) \in hb$ and hence the execution is data race free.

The reordering $d; e \rightsquigarrow e; d$ would result in following execution



In this case the $(a, e) \notin hb$ and hence result in data race.

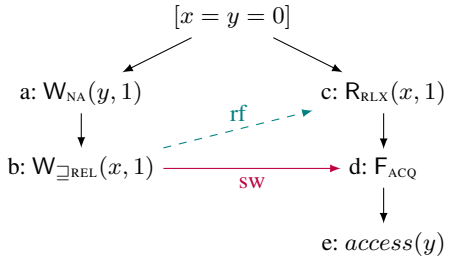
Hence the transformation is unsafe.

A.6 $\mathsf{R}_{\text{RLX}}; \mathsf{F}_{\text{ACQ}} \rightsquigarrow \mathsf{F}_{\text{ACQ}}; \mathsf{R}_{\text{RLX}}$

Consider the following program

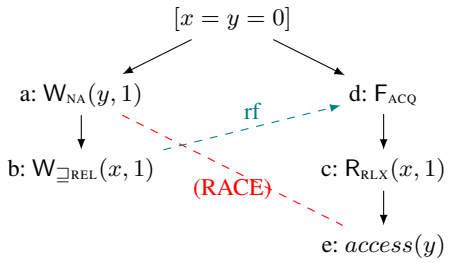
$$\begin{array}{l} y = 1; \\ x.\mathsf{store}(1, \text{REL}); \end{array} \Bigg\| \begin{array}{l} \mathsf{while}\ (1 \neq x.\mathsf{load}(\text{RLX})); \\ \mathsf{fence}(\text{ACQ}); \\ \mathsf{access}(y); \end{array}$$

This program is well synchronised, because for the access to y to occur, the relaxed load must have read from the release store of x, and therefore the release store and the acquire fence synchronise.



In this execution $(a, e) \in \mathsf{hb}$ and hence the execution is data race free.

The reordering $c; d \rightsquigarrow d; c$ would result in following execution



In this case $(a, e) \notin hb$ and hence there is a data race. Therefore, the transformation is unsafe.

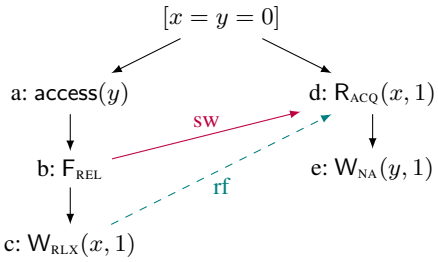
With a similar program, we can show that moving a RLX or REL RMW past an acquire fence is also unsafe. (Just replace $1 \neq x.\mathsf{load}(\text{RLX})$ with $\neg x.\mathsf{CAS}(1, 2, \{\text{RLX}, \text{REL}\})$.)

A.7 access; $\mathsf{F}_{\text{REL}} \rightsquigarrow \mathsf{F}_{\text{REL}}$; access

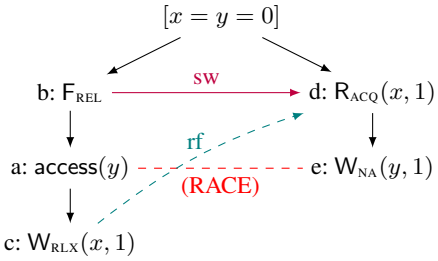
Similarly consider the program below

$$\begin{array}{l} \mathsf{access}(y); \\ \mathsf{fence}(\text{REL}); \\ x.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \mathsf{if}\ (x.\mathsf{load}(\text{ACQ})) \\ \quad y = 1; \end{array}$$

This program is data race free, because for a store of y to occur, the load of x must read 1 and therefore synchronize with the fence:



Reordering $a; b \rightsquigarrow b; a$, however, results in a racy execution:



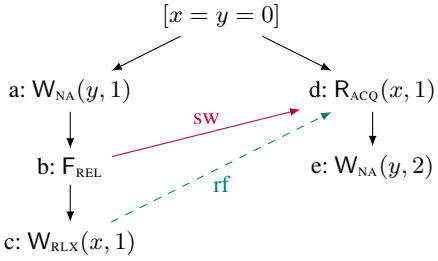
Therefore, the transformation is unsafe.

A.8 $\mathsf{F}_{\text{REL}}; \mathsf{W}_{\text{RLX}} \rightsquigarrow \mathsf{W}_{\text{RLX}}; \mathsf{F}_{\text{REL}}$

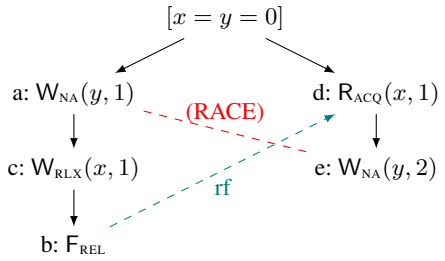
Consider the following program

$$\begin{array}{l} y = 1; \\ \mathsf{fence}(\text{REL}); \\ x.\mathsf{store}(1, \text{RLX}); \end{array} \Bigg\| \mathsf{if}\ (x.\mathsf{load}(\text{ACQ}))\ y = 2;$$

The program is race-free because if $y = 2$ happens, then it happens after the $y = 1$ store because the load of x must synchronise with the fence.



Reordering $b; c \rightsquigarrow c; b$, however, can result in the following racy execution:



Hence, the transformation is unsafe.

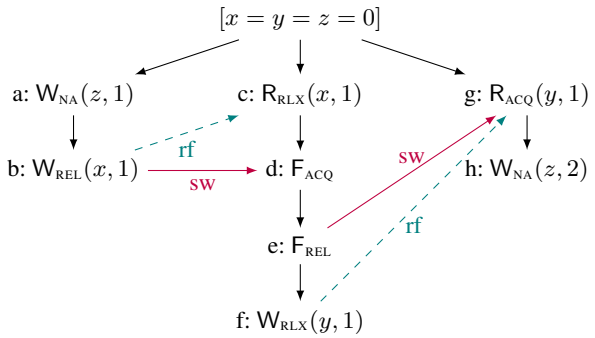
With a similar program, we can show that moving a release fence past a RLX or ACQ RMW is also unsafe. (Just replace $x.\mathsf{store}(1, \text{RLX})$ with $x.\mathsf{CAS}(0, 1, \{\text{RLX}, \text{ACQ}\})$.)

A.9 $\mathsf{F}_{\text{ACQ}}; \mathsf{F}_{\text{REL}} \rightsquigarrow \mathsf{F}_{\text{REL}}; \mathsf{F}_{\text{ACQ}}$

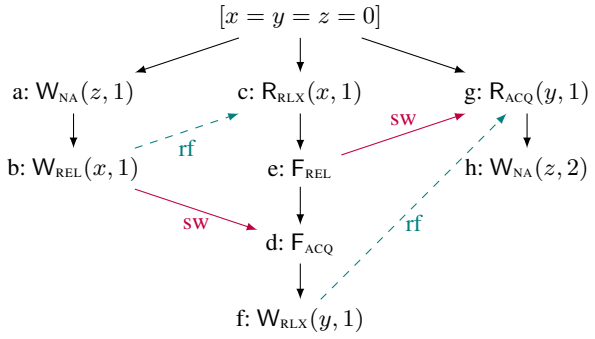
Consider the following program

$$z = 1; \quad x.\mathsf{store}(1, \text{REL}); \;\Big\|\; \begin{array}{l}\mathsf{if}\ (x.\mathsf{load}(\text{RLX}))\ \{\\ \quad \mathsf{fence}(\text{ACQ});\\ \quad \mathsf{fence}(\text{REL});\\ \quad y.\mathsf{store}(1, \text{RLX});\\ \}\end{array} \;\Big\|\; \mathsf{if}\ (y.\mathsf{load}(\text{ACQ}))\ z = 2;$$

The program is race free because the only consistent execution containing conflicting accesses is the following:



which is, however, not racy because $\mathsf{hb}(a, h)$. Reordering d and e does, however, result in the following racy execution and is, therefore, unsound.

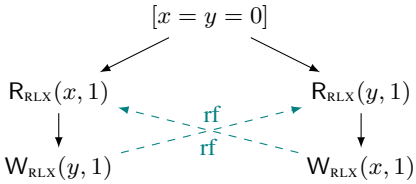


B. Counterexamples for the Arf model

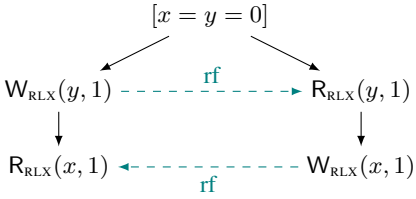
Reordering an atomic read past an adjacent atomic write is unsound in the Arf memory model. We show this first for an atomic load and an atomic store. Consider the following program, where implicitly all variables are initialised to 0.

$$\begin{array}{l} r = x.\mathsf{load}(\sqsupseteq \text{RLX});\\ y.\mathsf{store}(1, \sqsupseteq \text{RLX}); \end{array} \;\Big\|\; \begin{array}{l} r' = y.\mathsf{load}(\text{RLX});\\ x.\mathsf{store}(1, \text{RLX}); \end{array}$$

In this code the outcome $r = r' = 1$ is not possible. The only execution that could yield this result is the following,



which is inconsistent. If, however, we permute the instructions of the first thread, then this outcome is possible by the following execution:



Note that if we replace the load of x with a compare and swap, $x.\mathsf{CAS}(1, 2, \_)$, and/or the store of y with a compare and swap, $y.\mathsf{CAS}(0, 1, \_)$, the displayed source execution remains inconsistent, and while the target execution is valid. Hence reordering any kind of atomic read over any kind of atomic write is unsound in this model.

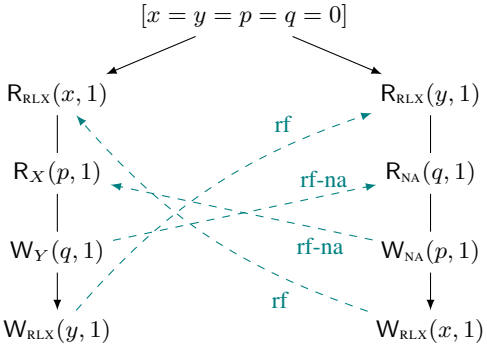
C. Counterexamples for the Arfna model

We show that reodering a non-atomic or atomic read past an adjacent non-atomic or atomic write is unsound. Consider the following program, where implicitly all variables are initialized to 0.

$$\begin{array}{l}\mathsf{if}\ (x.\mathsf{load}(\text{RLX}))\ \{\\ \quad t = p.\mathsf{load}(X);\\ \quad q.\mathsf{store}(1, Y);\\ \quad \mathsf{if}\ (t)\ y.\mathsf{store}(1, \text{RLX});\\ \}\end{array} \;\Big\|\; \begin{array}{l}\mathsf{if}\ (y.\mathsf{load}(\text{RLX}))\\ \quad \mathsf{if}\ (q)\ \{\\ \quad\quad p = 1;\\ \quad\quad x.\mathsf{store}(1, \text{RLX});\\ \quad \}\end{array}$$

(Where X and Y stand for any atomic or non-atomic access modes.)

Note that this program is race-free and its only possible outcome is $p = q = 0$. (The racy execution yielding $p = q = 1$ is inconsistent because it contains a cycle forbidden by the Arfna axiom.)

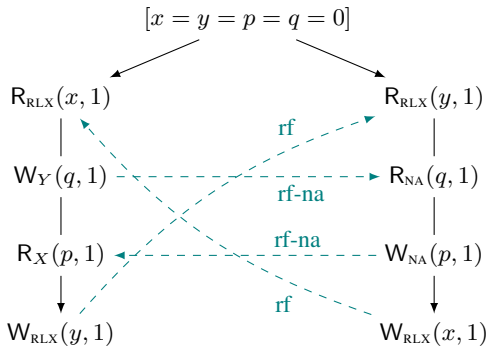


(The reads in this cycle are annotated by "rf-na.")

If, however, we permute the two adjacent non-atomic access of the first thread as follows:

$$t = p; q = 1; \rightsquigarrow q = 1; t = p;$$

then the following racy execution:



is consistent, and therefore the target program has more behaviours than the source program.

Note that if we replace the load $p.\mathsf{load}(X)$ with a compare and swap, $p.\mathsf{CAS}(1, 2, X)$, and/or the $q.\mathsf{store}(1, Y)$ with a compare and swap, $q.\mathsf{CAS}(0, 1, Y)$, the displayed source execution remains inconsistent, and while the target execution is valid. Hence the transformation adds new behaviour and is unsound.

D. Introductions and Eliminations of Accesses

D.1 Introduction of Memory Accesses

Introducing a memory access in a program is unsound in general because there may exist concurrent non-atomic accesses to that variable, which could race with the newly introduced memory access.

We can, however, introduce an unused atomic load or an overwritten store adjacent to another access of the same location, as we can then ensure that the irrelevant access would not introduce any races that were not already present in the original program.

More precisely, we can introduce an action, b, adjacently before an action, a, if $\mathsf{loc}(a) = \mathsf{loc}(b) \wedge (\mathsf{iswrite}(b) \Rightarrow \mathsf{iswrite}(a)) \wedge (\mathsf{isNA}(b) \Rightarrow \mathsf{isNA}(a))$.

We can introduce action b adjacently after action a if $\mathsf{loc}(a) = \mathsf{loc}(b) \wedge \neg\mathsf{iswrite}(b) \wedge (\mathsf{isNA}(b) \Rightarrow \mathsf{isNA}(a))$.

D.2 Elimination of Irrelevant Loads

We say that a load is irrelevant if the value read never used by the program. Removing such a load is valid under an interleaving semantics and under the TSO memory model.

In C11, however, eliminating an irrelevant load is not always valid, because doing so may remove an opportunity for synchronisation, thereby yielding more behaviours.

If, however, the eliminated load is non-atomic, then it cannot be involved in a synchronisation, and cannot be observed by other threads. Consequently, *eliminating irrelevant non-atomic loads is valid*.

D.3 Eliminations of Non-Adjacent Redundant Accesses

As mentioned in Section 7.2, Theorems 9, 10 and 11 from Section 7.1 can be extended to handle the non-adjacent cases. Here are the statements and the proof sketches of the corresponding theorems.

Theorem 12 (Read after Read)**.** *For a monotone memory model M, if* $\mathsf{Consistent}_M(lab, sb, asw, W)$ *and* $lab(a) = \mathsf{R}_X(\ell, v)$ *and* $lab(b) = \mathsf{skip}$ *and* $sb(a, b)$ *and* $\mathsf{Adj}(asw, a, b)$ *and* $\forall c.\ sb(a, c) \wedge sb(c, b) \implies \mathsf{loc}(c) \neq \mathsf{loc}(a) \wedge \neg\mathsf{isAcq}(c)$ *and* $lab' = lab[b := \mathsf{R}_X(\ell, v)]$*, then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$*,*
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$*, and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$*.*

Proof sketch. We apply the following sequence of transformations:

$$\begin{aligned}
& \mathsf{R}_X(\ell, v)\,;C\,;\mathsf{R}_X(\ell, v) \\
\xRightarrow{\textbf{reorder}}\ & \mathsf{R}_X(\ell, v)\,;\mathsf{R}_X(\ell, v)\,;C && \text{since } \neg\mathsf{isAcq}(C) \\
\xRightarrow{\text{(RAR-adj)}}\ & \mathsf{R}_X(\ell, v)\,;\mathsf{skip}\,;C \\
\xRightarrow{\textbf{reorder}}\ & \mathsf{R}_X(\ell, v)\,;C\,;\mathsf{skip}
\end{aligned}$$

□

Theorem 13 (Read after Write)**.** *For a monotone memory model M, if* $\mathsf{Consistent}_M(lab, sb, W)$ *and* $lab(a) = \mathsf{W}_X(\ell, v)$ *and* $lab(b) = \mathsf{skip}$ *and* $sb(a, b)$ *and* $\mathsf{Adj}(asw, a, b)$ *and* $\forall c.\ sb(a, c) \wedge sb(c, b) \implies \mathsf{loc}(c) \neq \mathsf{loc}(a) \wedge \neg\mathsf{isAcq}(c)$ *and* $Y \neq \mathrm{SC}$*, and* $lab' = lab[b := \mathsf{R}_Y(\ell, v)]$ *then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$*,*
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$*, and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$*.*

Proof sketch. We apply the following sequence of transformations:

$$\begin{aligned}
& \mathsf{W}_X(\ell, v)\,;C\,;\mathsf{R}_Y(\ell, v) \\
\xRightarrow{\textbf{reorder}}\ & \mathsf{W}_X(\ell, v)\,;\mathsf{R}_Y(\ell, v)\,;C && \text{since } \neg\mathsf{isAcq}(C) \\
\xRightarrow{\text{(RAW-adj)}}\ & \mathsf{W}_X(\ell, v)\,;\mathsf{skip}\,;C \\
\xRightarrow{\textbf{reorder}}\ & \mathsf{W}_X(\ell, v)\,;C\,;\mathsf{skip}
\end{aligned}$$

□

Theorem 14 (Overwritten Write)**.** *For a monotone memory model M, if* $\mathsf{Consistent}_M(lab, sb, W)$ *and* $lab(a) = \mathsf{skip}$ *and* $lab(b) = \mathsf{W}_X(\ell, v)$ *and* $\ell \neq \texttt{world}$ *and* $sb(a, b)$ *and* $\mathsf{Adj}(asw, a, b)$ *and* $\forall c.\ sb(a, c) \wedge sb(c, b) \implies \mathsf{loc}(c) \neq \ell \wedge \neg\mathsf{isRel}(c)$ *and* $lab' = lab[a := \mathsf{W}_X(\ell, v')]$*, then there exists W' such that*
(i) $\mathsf{Consistent}_M(lab', sb, asw, W')$*,*
(ii) $\mathsf{Observation}(lab', W') = \mathsf{Observation}(lab, W)$*, and*
(iii) $\mathsf{Racy}_M(lab, sb, asw, W) \Rightarrow \mathsf{Racy}_M(lab', sb, asw, W')$*.*

Proof sketch. We apply the following sequence of transformations:

$$\begin{aligned}
& \mathsf{W}_X(\ell, v')\,;C\,;\mathsf{W}_X(\ell, v) \\
\xRightarrow{\textbf{reorder}}\ & C\,;\mathsf{W}_X(\ell, v')\,;\mathsf{W}_X(\ell, v) && \text{since } \neg\mathsf{isRel}(C) \\
\xRightarrow{\text{(OW-adj)}}\ & C\,;\mathsf{skip}\,;\mathsf{W}_X(\ell, v) \\
\xRightarrow{\textbf{reorder}}\ & \mathsf{skip}\,;C\,;\mathsf{W}_X(\ell, v)
\end{aligned}$$

□