P3064R0: How to Avoid OOTA Without Really Trying

Alan Stern
stern@rowland.harvard.edu

Paul E. McKenney
paulmck@kernel.org

Michael Wong
fraggamuffin@gmail.com

Maged Michael
maged.michael@gmail.com

April 5, 2024 (Post-Tokyo)

Audience: SG1

Abstract

The out-of-thin-air (OOTA) properties of `memory_order_relaxed` in the C++ memory model have caused considerable consternation over the years. Attempts to create memory models that rule out OOTA behaviors have been either non-executable, complex, or unloved by C++ implementers. But at the same time, we know of no instances of OOTA behavior in real C++ implementations.

We focus on shared-memory programs and C++ implementations based on traditional compilers and computing hardware, including CPUs and GPGPUs. This context permits us to consider the detailed relations between source code and machine code required by the restricted nature of volatile atomic accesses. The OOTA problem and the related challenge of coming to grips with semantic dependency are much more tractable at the hardware level than at the source level, thanks to existing formal hardware models. We show that these models' constraints prevent OOTA cycles from occurring in undefined-behavior-free C++ programs running on compiler-based implementations, provided the cycles involve only volatile atomics. We also extend this work to nonvolatile atomics by defining "quasi-volatile" behavior that we expect C++ implementations will adhere to if they perform single-thread analysis.

Contents

```
 1 atomic<int> x(0);
 2 atomic<int> y(0);
 3
 4 void thread1()
 5 {
 6   int r1 = x.load(memory_order_relaxed);
 7   y.store(r1, memory_order_relaxed);
 8 }
 9
10 void thread2()
11 {
12   int r2 = y.load(memory_order_relaxed);
13   x.store(r2, memory_order_relaxed);
14 }
```

Listing 1: Simple OOTA

1 Introduction and Background

This paper shows that compiler-based C++ implementations subject to reasonable constraints on how they treat accesses to atomic objects cannot exhibit out-of-thin-air (OOTA) cycles. It follows that these implementations need to take almost no special actions to avoid OOTA. In fact, for many compilers the constraints boil down to a simple "Don't invent or duplicate atomic accesses", which the compiler probably wouldn't do anyway.

We begin with a brief overview of the OOTA problem, followed by an equally brief summary of prior work in this area, and ending with a quick overview of the `herd7` tool that will be used to evaluate litmus tests.

1.1 Brief OOTA Overview

In broad terms, OOTA occurs theoretically when a group of threads load from each others' stores and each thread's store depends on the value returned by that thread's load. The collection of loads and stores forms an *OOTA cycle*. In the most extreme cases a nonsensical value can pop up "out of thin air"; however, as shown by Listing 3 below, OOTA cycles need not involve nonsensical values.

1.1.1 Simple OOTA Cycle

Listing 1 [19] shows a simple example where an OOTA cycle might result in all of `x`, `y`, `r1`, and `r2` having final values of 42, despite the fact that there is nothing in the initial values or the executable code to support such an outcome:

1. Line 6 loads from `x` into `r1`, claiming to read the value of line 13's store rather than `x`'s initial value and somehow obtaining 42.

2. Line 7 stores `r1`, and thus 42, to `y`.

3. Line 12 loads 42 from `y` to `r2`.

4. Line 13 stores 42 to `x`, justifying the value loaded by line 6.

Because nothing else in the C++ memory model rules out such OOTA cycles, the C++ standard explicitly prohibits them in 33.5.4p8 (`[atomics.order]`) [13]:

> Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.

The standard's prohibition of OOTA is of course important, but those of us writing code in the real world must rely on actual C++ implementations. And in these implementations, this prohibition is in fact enforced by TSO ordering in strongly ordered systems and by data-dependency ordering in weakly ordered systems.[1]

In Listing 1 there is a *semantic dependency* from line 6 to line 7 and another from line 12 to line 13. (Roughly speaking, there is a semantic dependency from a given load to a given store when *all other things being equal, a change in the value loaded can change the value stored or prevent the store from occurring at all.* Here the dependencies are trivial because the values stored simply *are* the values that were loaded.) Since real-world CPUs cannot store something until they have determined its value,[2] the stores in lines 7 and 13 cannot take place until the CPU knows what values are loaded by lines 6 and 12, respectively. Thus the hardware orders these stores after their corresponding loads, and this ordering prevents the OOTA result.

1.1.2 Simple Reordering

It is important to distinguish true OOTA cycles from OOTA-like behavior caused by simple reordering. An example of simple reordering is shown in Listing 2 [19]. Both the C++ compiler and the CPU are within their rights to reorder lines 12 and 13, which can result in all of `x`, `y`, `r1`, and `r2` having the value 42 as follows:

1. Line 13 stores 42 to `x`.

2. Line 6 loads 42 from `x` into `r1`.

3. Line 7 stores `r1`, and thus 42, to `y`.

4. Line 12 loads 42 from `y` to `r2`.

Current C++ implementations can and do exhibit this reordering behavior.

This paper will follow P2055R0 [19] in using the term *full C++* to denote the standard including the prohibition of OOTA mentioned above. Unlike that article, we will use the term *loose C++* (rather than *strict C++*, which seems too similar to *full*

[1] The need to prohibit simple OOTA is one reason why compiler-based value speculation optimizations require checks on such speculation, and these checks must be based on actual values loaded.

[2] Another way of saying this is that real-world CPUs do not make their stores visible to other CPUs until those stores are no longer speculative. See Section 7.1 for more discussion about hardware speculation.

```
 1 atomic<int> x(0);
 2 atomic<int> y(0);
 3
 4 void thread1()
 5 {
 6   int r1 = x.load(memory_order_relaxed);
 7   y.store(r1, memory_order_relaxed);
 8 }
 9
10 void thread2()
11 {
12   int r2 = y.load(memory_order_relaxed);
13   x.store(42, memory_order_relaxed);
14 }
```

Listing 2: Simple Reordering

C++ for comfort) to denote a hypothetical standard that excludes this prohibition but is otherwise identical to full C++. Unqualified *C++* means full C++.

The next section looks at how prior work has refined these issues.

1.2 Prior Work

All OOTA workers owe a debt to the foundational work in the infamous "Causality Test Cases",[3] a version of which may be found in Appendix E.

Some executable C++ memory models correctly flag at least some executions involving OOTA cycles [2].[4] However, because these models are atemporal, they cannot reject OOTA executions other than by flagging the OOTA value as arbitrary, which some in fact do in at least some cases.

P0442R0 ("Out-of-Thin-Air Execution is Vacuous") [21] provided a decision procedure for classifying behaviors as permitted misordering on the one hand or disallowed OOTA on the other, using a perturbation method based on the insight that all OOTA behaviors are fixed-point computations.

Some workers recommend avoiding OOTA by forcing prior relaxed loads to be ordered before subsequent relaxed stores [7, 6, 14], but this can require real instructions be executed [17, Section 7.1], consuming real time and real electrical power to solve a strictly theoretical problem. This might have been acceptable in the 1960s of some of the authors' youths, but it is now the year 2024.

Other workers recommend various procedures to identify and avoid OOTA cycles [14, 26, 15, 3], but none of these have been looked upon favorably by C++ implementers. Some of these workers appear to have abandoned this effort, but as of early 2024, Mark Batty is persisting with modular relaxed dependencies.

[3] `http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html`.

[4] Others cleverly avoid this issue by forbidding atomic stores of nonconstant values [4].

```
 1 C oota-ctrl
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r0 = atomic_load_explicit(y, memory_order_relaxed);
 9   if (r0 == 42)
10     atomic_store_explicit(x, 42, memory_order_relaxed);
11   else
12     atomic_store_explicit(x, r0, memory_order_relaxed);
13 }
14
15 P1(atomic_int *x, atomic_int *y) {
16   int r1 = atomic_load_explicit(x, memory_order_relaxed);
17   if (r1 == 42)
18     atomic_store_explicit(y, 42, memory_order_relaxed);
19   else
20     atomic_store_explicit(y, r1, memory_order_relaxed);
21 }
22
23 locations[x;y]
24 exists(0:r0=42 /\ 1:r1=42)
```

Listing 3: OOTA Cycle

Goldblatt looked at interactions between OOTA cycles and undefined behavior (UB) [10]. This document will concentrate on examples lacking UB.

All this work focused on either identifying OOTA or seeing how C++ implementations could avoid it. None applied real-world hardware ordering constraints to the problem of avoiding OOTA cycles, yet doing so might help explain why no known real-world C++ implementation results in OOTA behavior. We therefore dig more deeply into OOTA cycles in the light of these constraints.

1.3 Code-Analysis Tool

This paper will use the `herd7`[5] tool to analyze fragments of C++ code. This tool carries out the moral equivalent of full state-space searches of concurrent code fragments. In some cases its output will include executions with OOTA cycles, on occasion reporting undefined values for the variables involved in the cycle.[6]

Listing 3 shows a code fragment that under loose C++ has an OOTA cycle (although the cycle is of course prohibited in full C++). This section describes the fragment, thereby giving an overview of the `herd7` tool.

The first line identifies it as a C-language litmus test and gives it a name, and this name identifies the litmus test's source file within the `litmus` directory in the `https://github.com/paulmckrcu/oota` repository. Lines 2–5 initialize variables, in this case setting the initial values of the global shared variables `x` and `y` to zero. (Variables that are not explicitly initialized are initialized to zero by default.) Lines 7–13

[5] Available at `https://github.com/herd/herdtools7`.

[6] This happens only some of the time because of idiosyncrasies in the algorithm used by `herd7`'s self-consistency solver.

```
 1 Test oota-ctrl Allowed
 2 States 2
 3 0:r0=0; 1:r1=0; [x]=0; [y]=0;
 4 0:r0=42; 1:r1=42; [x]=42; [y]=42;
 5 Ok
 6 Witnesses
 7 Positive: 1 Negative: 3
 8 Condition exists (0:r0=42 /\ 1:r1=42)
 9 Observation oota-ctrl Sometimes 1 3
10 Time oota-ctrl 0.00
11 Hash=db3300c0e3cc86ab1a1477ca446dac5e
```

Listing 4: OOTA Cycle, `herd7` Output

define the first thread, `P0()`, and lines 15–21 define the second thread, `P1()`.[7] The arguments to both `P0()` and `P1()` specify which of the global shared variables each thread may access, in this case, `x` and `y`. The body of each thread contains C++ code, written in a slightly stilted manner to keep the load operations separate from the rest and because `herd7`'s knowledge of C++ is limited.

Line 23 has a `locations` clause, which causes `herd7` to dump out the final values of `x` and `y`. Finally, line 24 specifies an `exists` clause, which gives a condition to check for the final values of the specified variables. The `0:` prefix denotes a variable local to `P0()` and the `1:` prefix denotes a variable local to `P1()`. The `/\` is a boolean AND, and the = signs are equality comparisons. If a variable appears in the `exists` clause then the final value of that variable constitutes observable behavior.

Listing 4 shows the corresponding output of the `herd7` tool. Lines 3 and 4 show the possible states, with line 4 showing the counterintuitive outcome where both threads load the value 42. Normally these lines would include only those variables mentioned in the `exists` clause, but because of the `locations` clause the values of `x` and `y` are also listed, which can be helpful for debugging. Line 9 contains `Sometimes` (as opposed to `Never` or `Always`), indicating that some executions satisfy the `exists` clause and others do not.

Later examples will usually combine the litmus test and the `herd7` output into one listing, as shown in Listing 5, which recasts Listing 2 into litmus-test form. Listing 1 can also be recast as a `herd7` litmus test, as shown in Listing 31 on page 74. Other OOTA-related litmus tests may be found in Appendix D and Appendix E.

2 OOTA and Semantic Dependencies

Section 2.1 demonstrates advantages of formulating an OOTA cycle as a cycle in sdep ∪ rfe instead of the traditional sdep ∪ rf. Section 2.2 then discusses properties of semantic dependences, showing that they are functions of executions rather than strictly of source code, among other things.

[7] The "P" stands for "process", which is `herd7`'s name for "thread".

```
 1 C simple-reordering
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   atomic_store_explicit(y, r1, memory_order_relaxed);
10 }
11
12 P1(atomic_int *x, atomic_int *y) {
13   int r2 = atomic_load_explicit(y, memory_order_relaxed);
14   atomic_store_explicit(x, 42, memory_order_relaxed);
15 }
16
17 exists(0:r1=42 /\ 1:r2=42)
```

Analysis by "`herd7 -c11 litmus/simple-reordering.litmus`":

```
 1 Test simple-reordering Allowed
 2 States 3
 3 0:r1=0; 1:r2=0;
 4 0:r1=42; 1:r2=0;
 5 0:r1=42; 1:r2=42;
 6 Ok
 7 Witnesses
 8 Positive: 1 Negative: 3
 9 Condition exists (0:r1=42 /\ 1:r2=42)
10 Observation simple-reordering Sometimes 1 3
11 Time simple-reordering 0.00
12 Hash=1b186d8f9445998c9c4ac29e062ffb74
```

Listing 5: Simple Reordering as Litmus Test

2.1 OOTA: rf versus rfe

Semantic dependencies form only one type of link in an OOTA cycle. The other type extends from a given store to a load that returns the value stored. It is tempting to argue that the store must precede the load in global time and combine this with the intuitive notion that any real C++ implementation must consume global time when computing a semantic dependency. This combination suggests that OOTA cycles cannot occur. The idea has been formalized by defining an OOTA cycle as a cycle in sdep $\cup$ rf [20], where sdep is the set of semantic dependencies within each thread and rf is the set of store-to-load "reads-from" links, whether within a thread (rfi) or between threads (rfe).[8]

This is a fine definition and is consistent with the words in the C++ standard, but it has a problem with intrathread rfi links as exemplified by the following code:

```
1   int r2 = atomic_load_explicit(&x, memory_order_relaxed);
2   atomic_store_explicit(&y, r2, memory_order_relaxed);
3   int r3 = atomic_load_explicit(&y, memory_order_relaxed);
4   atomic_store_explicit(&z, r3, memory_order_relaxed);
```

This is an elaboration of `thread2()` from Listing 1 that adds `z` along with lines 3 and 4. The problem is that a C++ implementation may note that line 3 could well execute immediately after line 2, giving other threads no chance to modify `y` in between. Such an implementation might therefore behave as if the source code had instead been as follows:

```
1   int r2 = atomic_load_explicit(&x, memory_order_relaxed);
2   atomic_store_explicit(&y, r2, memory_order_relaxed);
3   // int r3 = atomic_load_explicit(&y, memory_order_relaxed);
4   atomic_store_explicit(&z, r2, memory_order_relaxed);
```

Here line 3 has been optimized away in favor of line 4 storing the same value to `z` that was stored to `y` by line 2. And given that the load from `y` no longer exists, it cannot possibly act as a temporal constraint.

In order to avoid these rfi links we will substitute rfe for rf, defining an OOTA cycle—for now—as a cycle in sdep $\cup$ rfe. Any rfi links in a cycle can instead be interpreted as part of sdep. Although this does shunt additional complexity onto the term "semantic dependency", it also enables us to cleanly separate the interthread and intrathread portions of any given OOTA cycle.

The inability of rfi links to act as temporal constraints is not the only, or even the main, weakness in the intuitive argument against OOTA cycles. The primary difficulty lies in the fact that the code transformations performed by optimizing compilers can destroy dependencies, including semantic ones (depending on one's definition). That is, even when the potential for a dependency exists in the source code for a thread, there might be no dependency in the machine code produced by a compiler. There would then be no constraint forcing the implementation to execute the thread's store later in global time than the load it supposedly depends on, and thus no impediment to the occurrence of an OOTA cycle. We will see examples of this destruction in Sections 4.2 and 4.3 below.

[8] See Appendix A for definitions and properties of rf, rfe, and rfi.

2.2 Properties of Semantic Dependencies

This section uncovers some semantic-dependency complexities. Section 2.2.1 shows that semantic dependencies are functions of executions, rather than being strict functions of the source code. Section 2.2.2 shows that semantic dependencies do not necessarily extend from a single load to a single store, but can instead involve multiple loads. Section 2.2.3 shows that semantic dependencies can be affected by cross-thread optimizations. Sections 2.2.4 and 2.2.5 show that `if` statements can have surprising effects on semantic dependencies, up to and including eliminating them completely. Finally, Section 2.2.6 demonstrates some challenges in determining which of a group of stores is involved in a given semantic dependency.

2.2.1 Semantic Dependencies and Source Code

Some discussions of semantic dependencies assume that they are strictly functions of the source code. Although there are ways of making this work, many instances of semantic dependency must be considered functions of particular executions. Consider for example:

```
x = y * z;
```

(Here and below, we have written shared-variable accesses without annotations, for brevity. Please imagine they are all relaxed atomic.)

As long as `z` is zero, changes in the value of `y` will not cause a change in the value stored to `x`. As a result, the semantic dependency from `y` to `x` exists only in executions where `z` is nonzero, which shows it is a property of the execution, not just of the source code.

2.2.2 Semantic Dependencies Can Be Many-To-One

Suppose that in some execution of the previous example, both `y` and `z` are zero. Then changes to either `y` or `z` will not cause a change in the value stored to `x`. In other words, in this execution there is no semantic dependency from either `y` or `z` to `x`. But there *is* a semantic dependency from the *pair* {`y`, `z`} to `x`, because changes to both `y` and `z` can cause the value stored in `x` to change. This means that, prior work [21] notwithstanding, accurate definitions of sdep cannot always rely on single-variable perturbations; they must consider changes to multiple variables. See Appendix D.2 for examples and additional discussion.

Since we can no longer regard sdep as always relating a single load to a store, the notion of a cycle involving sdep appears problematic. We are forced to change our definition of an OOTA cycle again; we will say that an execution is an instance of OOTA if in that execution:

> There are stores $W_0, \ldots, W_m$, where each W_i semantically depends on a set of loads $\{R_{i,0}, \ldots, R_{i,n_i}\}$, such that each $R_{i,j}$ reads from one of the W_k stores in a different thread.

This makes OOTA more complicated than a simple cycle but we will continue to refer to "OOTA cycles" out of habit. Note that this new definition includes and generalizes the earlier "cycle in sdep ∪ rfe" definition.

2.2.3 Semantic Dependencies Affected by Cross-Thread Optimizations

Consider the following:

```
x = y - z;
```

There appear to be semantic dependencies from `y` to `x` and from `z` to `x`. However, if the implementation somehow knows that `y` is always equal to `z` at this point then there is no semantic dependency; the implementation can act as if the statement were simply "`x = 0`". We leave aside the question of how the implementation would know this, given that `y` and `z` cannot be updated simultaneously[9] and are subject to change at any time by other threads (a point we will return to in Section 4.2).

2.2.4 Semantic Dependencies Affected by `if` Statements

Consider the following `if` statement:

```
r1 = x;
if (r1 > 0)
    y = r1;
else
    z = r1;
```

Here there is a semantic dependency from `x`, but in some executions it extends to `y` and in others to `z`. This is an example of a load affecting not the value of a given store, but rather whether or not that store is executed at all.

2.2.5 Semantic Dependencies Not Affected by `if` Statements

Compare this example to the previous one:

```
if (x > 0)
    y = 42;
else
    y = 42;
```

Because the stores executed on each arm of the `if` statement write identical values to identical addresses, one could equally well regard the two statements as performing two different stores or as performing for all intents and purposes a single store, independent of `x`. Reasonable C++ implementations might disagree on this matter and therefore on whether or not the example has a semantic dependency. It is the implementation's choice.

[9] At least not by any means within the confines of the standard.

2.2.6 Semantic Dependencies and Matching Up Stores

Suppose we take the view that the previous example involves only one store. This opens up the door to greater complexity:

```
if (x > 0) {
    L1: y = 42;
} else {
    y = 53;
    y = 42;
}
```

Consider an execution in which `x` is greater than zero, so the statement labeled `L1` runs. Is it semantically dependent on `x`? The answer isn't immediately clear. If the other arm of the `if` is taken then a store of the same value 42 to `y` occurs, but 53 is written before it. Which of these two stores should be compared with the store in `L1`?

One way to cut the Gordian knot is to match up the stores by the order they occur: Since `L1` is the first store to `y` in its arm of the `if` statement, it should be matched up with the first store to `y` in the other arm. Those two stores write different values so there is a semantic dependency.

On the other hand, a compiler may decide to drop the `y = 53` store entirely, leaving it out of the machine code, on the grounds that it's always possible for the two adjacent stores to `y` to execute in such quick succession that no other thread manages to read the value 53 before it gets overwritten with 42. If the compiler does this then the first store to `y` in that arm of the `if` statement *would* agree with the store in `L1`, and so there would not be a semantic dependency. Once again, the decision is up to the implementation.

We have seen several examples showing that semantic dependencies may vary according to the execution and even the implementation. This raises several questions, of which the first is: What exactly is an execution?

3 What is an Execution?

This section looks more carefully at executions, in some cases revisiting example code from Section 2.2. Section 3.1 looks at executions from the viewpoint of the abstract machine, and Section 3.2 looks at them from the viewpoint of the computer hardware. Finally, Section 3.3 describes how to relate these two viewpoints.

3.1 Abstract Executions

The C++ standard describes the execution of a program in terms of "a parameterized nondeterministic abstract machine" in 4.1.2p1 (`[intro.abstract]`). This description specifies how the abstract machine carries out the operations of a source program in great, but not complete, detail:

- Some of the abstract machine's characteristics are implementation defined, including things like the number of bits in the various integer types or whether the `char` type is signed.

- Some aspects of an execution are unspecified or nondeterministic, including things like the order of evaluation of the operands of most binary operators or of the arguments in a function call. Implementations may choose from a set of allowed behaviors.
- Some actions are deemed to have undefined results; the standard says essentially nothing about programs that can give rise to undefined behavior.
- Asynchronous actions (i.e., signal handlers) are largely ignored.
- Input and output are not described in any detail.

In addition to these points, the standard does not specify which store an atomic load must read from, beyond requiring that the overall pattern of loads and stores be consistent with the C++ memory model. We assume that programs will not indulge in any computations that could vary spontaneously from one execution to another, such as basing a dependency on the time of day or a process ID.

The implementation-defined aspects can affect whether or not an abstract execution contains a semantic dependency. As an example consider the following, where the type of `c` is `char`:

```
y = (c >= 0);
```

Here `y` is semantically dependent on `c` in executions running on implementations in which `char` is a signed type, but not those for which it is unsigned.

The same is true for the nondeterministic aspects of an execution. Consider this example, with `i` initially zero:

```
int foo(int a, int b)
{
   return a / b;
}

r1 = foo(++i, ++i);
x = r1 * z;
```

Because early C implementers could not come to agreement, the standard does not specify the order of evaluation of function arguments, so the value calculated for `r1` might be zero ($1/2$ truncated) or two ($2/1$). In the former case there is no semantic dependency from `z` to `x`, but in the latter case there is.[10]

(According to the current version of the standard, conflicting side effects in unsequenced subexpressions constitute undefined behavior, although there are proposals to make them defined in both C++ [25] and C [8]. Nevertheless, the example above is allowed because the order of evaluation of arguments to a function call is "indeterminately sequenced" (7.6.1.3p7 `[expr.call]`) rather than unsequenced, a subtle distinction.)

The abstract executions we use will be fully specified. This means that all the missing information must be supplied: the implementation-defined characteristics, the

[10] Thanks to Peter Sewell for pointing out this possibility.

selections for the nondeterministic pathways, and most notably, for each load, the store from which it reads and the value of the load. We ignore issues of signal handlers and I/O; in any case our litmus-test programs don't use them (but see the discussion of volatile loads in Section 3.3 below). The totality of this information—along with the program's source code, of course—determines within each thread a unique, linearly ordered series of steps to be carried out by the abstract machine. However, with a few exceptions[11] there is no ordering relation between steps carried out in different threads. Even if a relaxed atomic load in one thread reads from a relaxed atomic store in another thread, the standard does not require the store to come before the load in any meaningful way.

With the compiler-based implementations we are considering, the choices for the nondeterministic pathways are "frozen" into the machine-code executable file and thus are completely determined at runtime. A consequence of this is that if two abstract executions of the same thread under the same implementation agree on the values obtained by the load operations during their first N steps then they will agree in every respect during those steps, although they may diverge later.

3.2 Hardware Executions

The outcome when a given computer executes the machine code in a file has historically been much better defined than the executions of the C++ abstract machine. The hardware's behavior is typically specified with great precision by the designer or manufacturer, and there are formal, executable memory models describing exactly what patterns of loads and stores can occur. Thus, leaving aside questions of asynchronous interrupts and system calls, the behavior of a CPU executing a particular thread within a program is entirely determined by the values obtained by the various memory-load instructions.[12]

For this reason, the hardware executions we use will comprise (along with the machine code being run) the computer architecture and for each load instruction, the store instruction from which it reads and the value obtained. At this level, the fact that the original program was in C++ is irrelevant; the same concepts apply to the execution of a program in any compiled language.

A computer may execute the instructions in a thread out of order. The architecture specifies the extent to which this may happen, and it also specifies circumstances under which some pairs of instructions must be executed in order. Nevertheless, we will consider an execution to be determined by the values obtained by its loads. As with abstract executions, if two hardware executions of the same thread on the same type of computer agree on the values obtained by the load instructions during their first N steps then they will agree in every respect during those steps, although they may diverge later.

3.3 Relation Between Abstract and Hardware Executions

The C++ standard requires that for any valid implementation, when a program runs its observable behavior must be the same as that of some abstract execution of the

[11] Such as a load-acquire synchronizing with a store-release.

[12] We regard read-modify-write instructions as consisting of both a memory load and a memory store.

source code given the same input (in the absence of any abstract executions containing undefined behavior).[13] This means:

- The program's output must be the same as that of the abstract execution.
- Volatile accesses "are evaluated strictly according to the rules of the abstract machine" (4.1.2p6.1 `[intro.abstract]`).
- (There is a condition on the timing and interleaving of input and output, which does not matter for our purposes.)

We will say that the abstract execution is *realized by* the hardware execution.

Under any particular implementation, a single program can have many different abstract executions, varying in their decisions about which store each load reads from and thus the value obtained. It's worth noting, however, that not all the possible abstract executions of a program need be realizable by the machine-code executable file produced by that implementation. In fact, we will see that *none* of the possible OOTA executions allowed by the loose C++ abstract machine will ever be realized by the executables produced by many compilers,

Exactly what the standard's restriction on volatile accesses means isn't entirely clear. The handling of volatiles, as understood by compiler developers, has been described as more folklore or a gentlemen's agreement than anything else. To help guide C++ users and implementers, the standard adds these suggestive comments (9.2.9.2p5 and 6 `[dcl.type.cv]`):

> The semantics of an access through a volatile glvalue are implementation-defined.
>
> `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation.

Taking our cue from the folklore, we propose to recognize formally that programs with volatile objects can execute in two different kinds of environment: a benign one in which accesses to these objects work the same as nonvolatile memory accesses, and a nonbenign one in which accesses to volatile objects are subject to outside interference and act more like I/O. In particular, when it runs in a nonbenign environment, a program's volatile loads can return unpredictable values. They don't necessarily read from stores (in contrast to nonvolatile loads, which always must return the value of the store they read from). This implies that volatile load-acquires do not synchronize with volatile store-releases in the sense of the C++ memory model,[14] so they do not contribute to the happens-before relation. Also, in these environments the rfe relation does not apply to volatile loads and stores, and hence the accesses in an OOTA cycle must be nonvolatile.

Of course, the machine-code file produced by a compiler must work properly in either kind of environment. Therefore the compiler must generally treat accesses to

[13] This requirement is the standard's "as-if" rule.

[14] However, they might instead synchronize with store-releases in device firmware (or vice versa), roughly speaking.

volatile objects as a form of I/O, and it may not invent, omit, merge, or reorder these accesses, as we will discuss in Section 4.4 below.

Given this relation between abstract and hardware executions, it is time to turn our attention to the tools that manage hardware executions so as to enforce that relation, namely, compilers.

4 C++ Compilers

A complete C++ implementation consists of much more than just a compiler. Among other things, for example, it might have a collection of `.h` header files, a linker, runtime libraries, and a dynamic loader. Nevertheless, for our purposes the compiler is the most important component because it is what primarily determines the translation from a C++ source program to directly executable machine code. We will therefore use the terms "compiler" and "implementation" interchangeably.

Section 4.1 shows that C++ compilers can be influenced by their users as well as by the standard. Section 4.2 presents an example showing that global optimizations can destroy semantic dependencies, and then Section 4.3 analyzes examples showing that inventing atomic loads can destroy semantic dependencies. Section 4.4 then presents required properties of volatile atomic operations, and also defines properties of quasi-volatile atomic operations.

4.1 Users Influence the Behavior of Compilers

The exact definition of a computer language is a subject of some debate, with standards, implementations, and users all having some degree of influence [23, 24], and each being prone to change over time. In areas that are not well settled or where users might reasonably want to resist the dictates of the standard, compilers often provide switches to override their default behaviors. An example is GCC's `-funsigned-char` command-line argument, which causes it to treat variables of type `char` as unsigned. More examples of user control over language semantics are given in Appendix B and by the discussion in [23, 24].

We will consider these user-specified compiler switch settings to fall within the implementation-defined parameters of the C++ abstract machine. They should be provided, implicitly or explicitly, as part of any abstract execution.

4.2 Global Optimization Can Destroy Dependencies

Recall the Simple OOTA example in Listing 1 on page 5, in which `thread1()` loads the value of `x` and stores it in `y` while `thread2()` does the reverse. A globally optimizing loose C++ compiler given that program might transform it to the following before translating it into machine code, if the compiler is sufficiently perverse:

```
1 atomic<int> x(0);
2 atomic<int> y(0);
3
```

```
 4 void thread1()
 5 {
 6   int r1 = 42;
 7   y.store(r1, memory_order_relaxed);
 8 }
 9
10 void thread2()
11 {
12   int r2 = 42;
13   x.store(r2, memory_order_relaxed);
14 }
```

The loads previously on lines 6 and 12 have been replaced by constants. Such a transformation complies with the loose C++ standard, even though the resulting executable file would produce an unintuitive OOTA outcome every time it runs!

The only justification a compiler could have for generating output like this is that it knows exactly what accesses will be performed by both threads, and therefore it knows that it will not violate the loose C++ memory model by assuming each thread's load reads from the other's store.[15] A similar justification can underlie the reasoning in Section 2.2.3; in principle an analysis of the complete program could lead a compiler to conclude that `y` will always be equal to `z` whenever a particular `y - z` expression is evaluated, allowing the compiler to replace the expression with a constant `0`.

By contrast, a compiler that analyzes only one thread at a time when performing its optimizations and other code transformations will not have this kind of global knowledge, and consequently it would not perform the OOTA-ful transformation shown here.

Because we seek to find characteristics of compilers that will guarantee the absence of OOTA behavior in the machine code they generate, we will for now confine our attention to compilers that analyze only one thread of source code at a time. In more precise terms, we want the compilers under consideration always to generate the same machine-code output for threads having the same source code, regardless of the rest of the code in the programs containing those threads. Later on we will return to globally optimizing compilers.

4.3 Inventing Atomic Loads Can Destroy Semantic Dependencies

Consider this code, in which the final values of `r1` and `r2` are observable:

```
int r1 = (x != 0);
int r2 = (y != 0);
z = (r1 == r2);
```

It is clear that the store to `z` semantically depends on the load from `y`, because the value of `z` will change whenever `y` changes between zero and nonzero (all else being equal).

However, an especially devious compiler might transform the source into the following form before translating it to machine code:

[15] A less perverse compiler could choose to avoid the OOTA cycle simply by not making this transformation.

```
 1  int r1;
 2  int r1a = (x != 0);
 3  int r1b = (x != 0);
 4  int r2 = (y != 0);
 5  if (r1a != r1b) {
 6      r1 = r2;
 7      z = 1;
 8  } else {
 9      r1 = r1b;
10      z = (r1 == r2);
11  }
```

The idea is that `r1a`, `r1b`, and `r2` can each be only zero or one, so if `r1a` and `r1b` differ then one of them must be equal to `r2`. In executions where this happens—because another thread writes to `x` between the two loads—the implementation can choose at runtime to use for `r1` whichever value agrees with `r2`, as shown on line 6. Then the value stored to `z` on line 7 will always be one, with no dependence on the value loaded from `y`. This example is discussed more fully in Appendix D.4.

A noteworthy aspect of this transformation is that it invents an atomic load: The original form of the code reads `x` only once, whereas the transformed code reads it twice. Therefore we can rule out transformations like this one by insisting the compiler not invent (or duplicate) atomic loads. In fact, we will require that atomic accesses be treated as "quasi volatile", in that the compiler is allowed to omit, merge, or reorder them but not invent them.

Just what does this mean?

4.4 Volatile and Quasi Volatile Accesses

Declaring objects to be volatile is a way for the programmer to indicate that the hardware should perform all accesses to these objects exactly as written in the source code, perhaps because they represent memory-mapped device registers or DMA buffers rather than normal memory locations. In any event, we expect compilers' translations of volatile-object accesses into machine code to be as close to verbatim as possible.

To express this idea in more formal terms, and to explain what we mean by "quasi-volatile" object accesses, we augment the requirements for a hardware execution H to realize an abstract execution A. Each realization must include a map from the set of accesses of volatile objects in A to the set of instructions in H that access these objects, having the following properties:

- The map connects accesses of the same type (loads to loads and stores to stores) and to the same object.
- The map connects accesses in a thread of A to accesses in the corresponding thread of H.
- The map is value-preserving: The value of an access in A must be the same as the value of the access it maps to in H.

- In benign environments the map must preserve the rf relation. That is, if volatile load R in A reads from store W then then the instruction it maps to in H must read from the instruction that W maps to.
- The map is order-preserving: Two accesses in the same thread of A must map to accesses occurring in the same order in H. (In other words, the compiler may not reorder accesses to volatile objects.)
- The map is onto: For every access in H to a volatile object there must be an access in A that maps to it. (In other words, the compiler may not invent accesses to volatile objects.)
- The map is one-to-one: Different accesses in A map to different accesses in H. (In other words, the compiler may not merge accesses to volatile objects.)
- The map is total: Every access in A to a volatile object maps to some access in H. (In other words, the compiler may not omit accesses to volatile objects.)

Most of these are direct consequences of the fact that volatile-object accesses are considered to be a form of I/O when the program runs in a nonbenign environment. But to be clear, these requirements apply in all environments.

By contrast, accesses to quasi-volatile objects are normal memory accesses, not subject to unpredictable interference in nonbenign environments (otherwise the program's behavior would be undefined). However, we do impose most of the requirements above on quasi-volatile object accesses. The last two bullet points are left out: Compilers are allowed to merge or omit accesses to these objects. Because of this, the bullet point about preserving the rf relation applies only when R is not omitted, in which case W must not be omitted either, but now it applies in all environments. Lastly, the requirement for order preservation is weakened; it applies only to pairs of accesses to the same quasi-volatile object. Accesses to different objects may be reordered relative to each other.

Section 7.2 presents examples illustrating some of these requirements.

5 Hardware Dependencies, Instruction Ordering, and the Fundamental Property

Section 5.1 discusses hardware-level dependencies involving machine instructions. Section 5.2 then examines the relationships between various instruction-ordering mechanisms and semantic dependencies. This material feeds into Section 5.3, which presents the fundamental property of semantic dependencies and related complications.

5.1 Dependencies at the Hardware Level

Dependencies between machine instructions can be understood in terms of the flow of information within a CPU. Each instruction has a set of inputs and is the source of a set of outputs, some of which flow to the inputs of later instructions. The inputs determine what an instruction will do. A few examples:

- An `add` instruction would typically have two inputs (the register values to be added together) and two outputs (the sum to be stored in a general-purpose register and some condition-code bits—e.g., Zero, Carry, and Overflow—to be stored in a status register).
- A conditional-move instruction would have as inputs the condition-code bits to test, the source register value, and the target register value; the output would be the value to be stored in the target register.
- A conditional- or computed-branch instruction would have as inputs the condition-code bits to test, the address of the following instruction (to be used if the condition is false) and the destination address (to be used if the condition is true). The output is the new address to be written to the instruction-pointer register.
- A memory-load instruction's input is the address to load from, and its output is the value obtained by the load, to be stored in a register.
- A memory-store instruction's inputs are the value to store and the address at which to store it; there are no outputs.

Note: Hardware dependency analysis considers only the information flowing *within* a CPU, not information flowing between the CPU and memory, which is handled separately as part of the action taken by an instruction.

Using this scheme, we say that an instruction *J* is dependent on another instruction *I* when any of *I*'s outputs flow into *J*'s inputs, perhaps indirectly via some intermediate instructions. Tracing back the flow of information between instructions, you can see that any hardware dependency must ultimately emanate from a load instruction or the thread's initial register values (or possibly some sort of input instruction, but we will not consider that complication here). These are the only sources of truly new information in a thread.

The concept is simple, but it is important because of the way dependencies affect instruction ordering in weakly ordered architectures.

5.2 Instruction Ordering

A CPU may start executing an instruction speculatively, but at some point it must *commit* to a decision either to definitely execute the instruction's action with some collection of well defined inputs and outputs or else to abandon it. Thanks to the law of cause and effect, a CPU is not able to commit an instruction or its outputs until the relevant inputs' sources have committed. The reason is obvious: An input is subject to change at any time until its source commits to its value, and the instruction and its outputs can't commit until the CPU has fully determined what it will do and what they will be. (On the other hand, the CPU need not wait for inputs that won't affect the instruction's results. For instance, a conditional-move instruction wouldn't need to wait for the source register input to be determined once it knew that the condition was definitely false. And, although no architectures we are aware of do this, there is nothing in principle to prevent a CPU from committing a `multiply` instruction as soon as either one of its inputs' sources has committed to the value zero.)

Of course, an instruction's inputs don't all have to be outputs from earlier instructions; some of them may be immediate constants present in the instruction itself. In this case there is no need to wait for those inputs because their values are fully determined from the start.

The overall effect of these hardware dependencies is that if a change to an output of instruction I would lead to a change in the action or outputs of a later instruction J, then the CPU must commit I's output before committing J's action and outputs. Thus dependencies force instructions to commit in order, even on weakly ordered architectures.

We will say that one instruction is *ordered after* another to mean that the CPU forces it to commit after the other one commits. Hence instructions are ordered after the instructions they depend on. (Note that this implies nothing about when a load instruction retrieves its value from memory; it may do so long before it or a prior instruction commits.)

Dependencies aren't the only mechanism that can lead to ordering in hardware executions. The simplest alternative is a load reading from a store in another thread, i.e., the rfe relation. As mentioned in Section 1.1.1, a CPU does not make the value of a store instruction available for other CPUs to load until the store commits or some time later. And after this happens, it takes some time for the information about the store to travel from that CPU to others, owing to the inescapable facts that processors have nonzero size and information cannot travel faster than the speed of light. Since the load instruction cannot commit until its output (the value it reads) is fully determined, it must commit after the store it reads from. (To be fair, this should be considered more as a *result* of ordering than as a *cause* of ordering, in that if a load commits before a store on another thread then it cannot possibly read from that store.)

A similarly straightforward mechanism for ordering is conditional or computed branches. An instruction executing after such a branch cannot commit until the CPU has committed to whether the branch will be taken and if taken, where it will branch to (that is, the output value it will send to the instruction-pointer register); until then the CPU cannot know whether the later instruction should be executed at all. Therefore instructions following a conditional- or computed-branch instruction in a hardware execution must commit after the branch, and hence after the source for the branch's condition or destination input. (There could theoretically be an exception for a branch that conditionally jumps to the immediately following instruction. We can ignore this possibility by treating such branches as no-ops.)

5.3 The Fundamental Property of Semantic Dependencies

With the contents of the last few sections under our belts, we can formulate the Fundamental Property that we would like all semantic dependencies to satisfy. Note that this formulation makes sense only for implementations in which all atomic objects are treated as volatile or quasi volatile.

> Let W be a store which semantically depends on loads $\{R_0, \ldots, R_n\}$ in some abstract execution A, and suppose that W is not omitted in some hardware execution H realizing A. Then for some i, load R_i is not omitted

in H and the instruction it maps to is ordered before the instruction W maps to.

It's quite straightforward to show that under any implementation in which semantic dependencies satisfy the Fundamental Property, no OOTA cycle has a nontrivial realization.

Indeed, suppose that abstract execution A is realized by hardware execution H and A has an OOTA cycle. This means there are atomic stores W_i in A, semantically depending on atomic loads $\{R_{i,j}\}$ where each of the loads reads from one of the W_k stores in a different thread. Let W'_i and $R'_{i,j}$ be the hardware instructions these accesses map to in H, if they aren't omitted. Assuming that the stores are not all omitted, one of the W'_i instructions, let's say W'_0, must commit first. By the Fundamental Property, one of the loads that W_0 depends on, let's say $R_{0,0}$, is not omitted and $R'_{0,0}$ is ordered before W'_0. But now we have a contradiction:

- W'_0 commits after $R'_{0,0}$;
- $R'_{0,0}$ commits after the store instruction W'_k it reads from;
- W'_k commits no earlier than W'_0.

If all the stores in the OOTA cycle are omitted then all the reads must be omitted as well. In effect, the entire cycle has been optimized out of existence by the compiler. Although we are unable to prove it, we conjecture that in this situation there must be another abstract execution which has the same observable effects as A and is also realized by H, but in which the OOTA cycle does not occur. Thus there would be no way to tell, merely by observing the effects of H, whether there was an OOTA cycle or not. For this reason we declare realizations of OOTA cycles in which all the accesses are omitted to be *trivial*.

6 A Definition of Semantic Dependency

Semantic dependency is a notoriously difficult concept to define rigorously and precisely. A large part of the reason is because it was never a completely clear concept to begin with, especially when there are multiple accesses to the variables involved. In this section we will stick our necks out by offering just such a definition. No doubt many people will object to it for various reasons, but we nevertheless hope it will help move the discussion forward.

The definition given below is applicable only to C++ implementations that treat all atomic objects as though they are volatile or quasi volatile. (For compilers that perform only single-thread analysis, not global analysis, quasi volatile is sufficient.) In this setting we can relate abstract and hardware executions by means of the map of accesses described in Section 4.4. The key insight is that this allows us to consider semantic dependencies at the level of the machine code, where they are much more tractable.

Section 6.1 focuses on compilers that restrict their analysis to a single thread, Section 6.2 relaxes this single-thread restriction, Section 6.3 verifies the fundamental property of semantic dependencies, and Section 6.4 discusses general questions regarding our definitions.

6.1 For Compilers Using Single-Thread Analysis

In this section we consider implementations whose compilers perform only single-thread analysis and treat atomic objects as quasi volatile. This implies that if two different programs contain the same thread (i.e., the same source code for the functions and objects in the thread), the machine code generated by the compiler for the thread will be the same in the two programs.

We begin by recognizing that semantic dependencies are relative to a particular execution and a particular implementation, as described in Section 2.2. The same source code may or may not contain a semantic dependency, according to the details of the execution in question and the machine code produced by the compiler. For this reason we will characterize semantic dependencies in a given abstract execution realized by a given hardware execution. (While it possible to argue about semantic dependencies in abstract executions that have no hardware realizations, doing so seems pointless.)

Let A be an abstract execution of some program P containing a thread T, and let H be a hardware execution realizing A. Let W in T be a store to an atomic object, and let $\{R_0, \ldots, R_n\}$ in T be a set of loads from atomic objects on which W might or might not depend. We can dispose of one case immediately: If W is omitted in H then the issue of semantic dependency is moot. You can give either answer since it will have no effect. Therefore we'll assume that W is not omitted. Then:

> There is a semantic dependency from $\{R_i\}$ to W in A and H, relative to the compiler used to produce H, if there is another abstract execution B realized by hardware execution G under the same compiler that together *witness the semantic dependency*.

To be a proper witness, B must be an execution of some program Q, not necessarily the same as P but which contains the same thread T. The thread should start out with the same initial state in A and B, and all loads in A coming before any of the R_i should obtain the same value as they do in B (this is part of our interpretation of "all else being equal"). It follows that the two abstract executions of T will be identical up to the first of the R_i loads.

Let W' and $\{R_i'\}$ be the accesses in H that W and the non-omitted $\{R_i\}$ loads map to. We then require that the hardware executions of T in H and G be identical for an initial period lasting up to the first of the R_i'. Following this initial period there will be a common period, during which H and G execute the same machine instructions but do not necessarily compute the same values. This common period ends when one of the hardware executions takes a conditional branch that the other doesn't, or when a computed branch leads to different addresses in the two executions, or when T ends, whichever comes first. Past this point H and G diverge and are no longer directly comparable, as they execute different instructions. Our third requirement for being a witness is that each load in the common period either must obtain the same value in H and G, or must itself be one of the R_i' loads, or must be ordered in H after one of the R_i' loads (this is the remaining part of our interpretation of "all else being equal".)

Finally, we need to determine an instruction X' in G that corresponds to W'. If W' is in the initial or common period of H this is no problem; we can take X' to be W' itself. But if W' is in the divergent part of H then things aren't so simple. The choice

is somewhat arbitrary, and so we will fall back on the earlier proposal of matching up stores by the order they occur. Let y be the atomic object that W' stores to, and suppose W' is the Nth store to y within the divergent part of H. Then X' will be the Nth store to y in the divergent part of G, if such a store exists. Our last requirement for being a witness to a semantic dependency is that X' act differently from W': it doesn't exist, it stores a different value, or it stores to an object other than y.

6.2 For Compilers Using Global Analysis

As promised earlier, we now consider implementations whose compilers may use global analysis. In order to obtain the desired results we have to require that these compilers treat all atomic objects as volatile. Equivalently, the machine code generated by such a compiler must be the same for a given program as for a "volatilized" form of the program in which all the atomic objects are defined to be volatile.

In this context our definition of semantic dependency is essentially the same as before. Since we can no longer expect the machine code for a thread to be the same regardless of the program it belongs to, the program Q in the earlier definition (of which B and G are executions) must be P or its volatilized form. However, we do now allow the possibility that the executions B and G take place in a nonbenign environment. Aside from these minor adjustments, the definition remains unchanged.

6.3 Verifying the Fundamental Property

Of course we want to check that our definition of semantic dependency satisfies the Fundamental Property of Section 5.3. Given the information we have already presented, the demonstration is easy.

Suppose we have W, $\{R_i\}$, A, and H as in the definition. The Fundamental Property assumes that W is not omitted in H, so there is an abstract execution B with hardware realization G witnessing the semantic dependency of the store W on the loads $\{R_i\}$ in A and H. We must show that some R_i is not omitted and R_i' is ordered before W' in H. The proof splits into three cases.

First case: W' lies in the initial period of H. During the initial period of the hardware executions, H and G behave identically and therefore W' performs the same write in both. This contradicts the fact that B and G witness the semantic dependency.

Second case: W' lies in the common period of H. Since the action of W' in H is different from its action in G, at least one of its inputs must differ between the two hardware executions. Therefore the source instruction for that input must behave differently, and so must one of its sources, going back until we reach a load instruction that obtains differing values in H and G. Then W' depends on this load and so is ordered after it. And since the load must lie in the common period of H, by the definition of semantic dependency it must either be one of the R_i' or be ordered after one of them. Therefore W' is ordered after one of the R_i' in H, which certainly means that R_i is not omitted.

Third case: W' lies in the divergent part of H. This happens when W' comes after the conditional or computed branch which marks the end of the common period by going different ways in H and G. Just as in the previous case, since the branch behaves

differently in the two executions it must be ordered after one of the R'_i loads. And then so must W', because any instruction following a conditional- or computed-branch instruction must commit after the branch commits. QED.

A corollary of this result is that if an implementation's compiler either

- uses single-thread analysis and treats atomic objects as quasi volatile, or
- uses global analysis and treats atomic objects as volatile,

then programs produced by that implementation will never exhibit OOTA. Thus the implementation will automatically comply with full C++, even though it may be been designed only to comply with loose C++.

6.4 Outstanding Issues

Here we consider some general questions related to our definition of semantic dependency.

6.4.1 Relative versus Absolute Dependency

A drawback of the definition is that it is only relative to a specific compiler or implementation. This may strike some people as wishy-washy and avoiding the real problem, in that a given piece of code should either contain or not contain a semantic dependency, independent of the implementation used to run it.

We can address this drawback by defining an *absolute semantic dependency* as one that is present relative to any valid loose C++ implementation, past, present, or future, real or imagined. Of course this notion has its own problems, including that it is extremely nonconstructive and impossible to apply in practice. However it may be the best we can do with our current understanding of computing systems.

There is one thing we can definitely state: Programs produced by an implementation of the right sort will never exhibit absolute OOTA (that is, an OOTA cycle in which all the semantic dependencies are absolute). This is simply because an absolute semantic dependency is *a fortiori* a semantic dependency relative to the compiler in use.

But in fact we have shown more than this: Programs will never exhibit an OOTA cycle relative to the compiler used to build them, even when that cycle is not absolute. In this sense we have gone beyond the requirement of full C++.

6.4.2 Global Analysis and Volatile versus Quasi Volatile

In principle we don't need to require global-analysis compilers to treat atomic objects as volatile; our results would still hold if they merely treated them as quasi volatile. We chose not to do this because it would violate our intuitions about semantic dependencies.

For example, consider the Simple OOTA program, repeated here in simplified form:

```
void thread1() {
   int r1 = x;
   y = r1;
```

```
}

void thread2() {
   int r2 = y;
   x = r2;
}
```

A loose C++ compiler using global analysis and treating `x` and `y` as quasi-volatile objects could omit the two loads, replacing them in the machine code with simple assignments "`r1 = 42`" and "`r2 = 42`". This would be a valid transformation, and the resulting behavior would not count as OOTA according to our definition because the dependencies in `thread1` and `thread2` would not be semantic.

To see why not, recall that a semantic dependency must have a witness, another execution in which the store acts differently. But this transformed program has no other hardware executions; every time it runs it will store 42 to both `x` and `y`. (Keep in mind also that since the atomic objects are not treated as volatile, they are not subject to unspecified interference when the program runs in a nonbenign environment.)

This unintuitive behavior could not occur if the two loads were not omitted. In fact, the definition of semantic dependency might remain perfectly acceptable if the requirement for global-analysis compilers were weakened, if the compiler treated atomic objects as quasi volatile and in addition was not allowed to omit accesses to them. This is a possible topic for future research.

6.4.3 Effect of Memory Layout

Part of our demonstration of the Fundamental Property of semantic dependencies relies on the fact, stated in Section 3.1, that an abstract execution of a thread is entirely determined by the values obtained for its loads. But when we compare abstract executions of the same thread in two different programs, this may no longer be entirely true owing to the effect of differing memory layouts.

Consider this simple example:

```
x = (int) &x;
```

Even though the example contains no loads at all, it may store different values when running in different programs because the object `x` may be allocated at differing addresses in those programs. According to our definition, this could count as a degenerate OOTA cycle of length one, in which the store is semantically dependent on an empty set of loads!

To rule out such pathological counterexamples we should require that in a witness to a semantic dependency, the addresses of all the objects and functions referred to in the thread T are the same as in the original execution. This is a very technical restriction but there are occasions when the issue might realistically arise, such as when computing a hash value based on an object's address.

6.4.4 Merging Quasi-Volatile Loads

The compiler is permitted to merge quasi-volatile loads. This can lead to surprising results because a particular load may be merged with an earlier load in one execution and with a later load in another. This is demonstrated in the following, which is a variant of the example in Section 4.3:

```
int r1 = (x != 0);
int r2 = (x != 0);
int r3 = (x != 0);
int r4 = (y != 0);
z = (r2 == r4);
```

Consider an abstract execution in which `r1`, `r2`, and `r4` are zero and `r3` is one (because another thread changed the value of `x` between two of the loads). We would expect that the store to `z` would be semantically dependent on the load of `y` in this execution.

However, a single-thread analysis compiler can translate this into the machine-code equivalent of:

```
int r1 = (x != 0);
int r2;
int r3 = (x != 0);
int r4 = (y != 0);
if (r1 != r3) {
   r2 = r4;
   z = 1;
} else {
   r2 = r1;
   z = (r2 == r4);
}
```

In effect, the `r2` load is merged with the `r1` load in executions where `r1` is equal to `r3` or to `r4`, and it is merged with the `r3` load in other executions.

To demonstrate the semantic dependency in the original code, a suitable witness would have to include a hardware realization of the abstract execution in which `r1` and `r2` are zero and `r3` and `r4` are one. But there are no hardware realizations of this execution with the machine code indicated above! Since `r1` is different from both `r3` and `r4`, the `r2` load will be merged with the `r3` load and so `r2` will necessarily be one, not zero. Thus, relative to this compiler the example does not contain a semantic dependency.

Although it is unexpected, we cannot say that this conclusion is definitely wrong, because our intuitive notions of semantic dependency are not clear in cases where multiple loads of the same variable are present.

Despite these outstanding issues, our definition of semantic dependency allows us to bring to bear the real-world constraints outlined in the next section.

7 Real-World Constraints

The C++ standard imposes many constraints, and these have been considered in prior work. In real-world C++ implementations, however, the standard's abstract machine must be mapped onto real hardware, and this imposes additional constraints stemming from hardware architecture and design. This mapping and these constraints have been only partially accounted for in the standard and other work. Part of the reason for this is that accurate and executable formal descriptions of hardware memory models did not appear until after the standard was released [1, 17]. But they do exist now.

The following sections discuss these constraints, starting with hardware constraints and continuing with additional constraints imposed by the standard. An additional section discusses the OOTA-cycle implications for C++ tooling.

7.1 Hardware Architecture and Design

Computer systems are expected to continue increasing their use of speculative execution. Nevertheless, the main points of this paper will remain unaffected. To see why, keep in mind that on real systems observable effects must eventually be committed, but no effect can be committed while any of the computations that determine the effect remain speculative. Combining this with the obvious fact that a machine instruction cannot commit until it is no longer speculative, we can draw two conclusions:

- A load cannot commit until the store it reads from has committed.[16]
- A store cannot commit until all the loads on which it has a hardware dependency have committed.

As an example showing what can go wrong when these principles are violated, consider Listing 6. Suppose that the storage for `x` is located immediately before that for `y[]`, and suppose further that hardware speculation incorrectly guesses the value of `r1` will be -1, so that the speculated store of 42 on line 11 uses the address of `x`.[17] Then if line 13 reads from the speculative store and commits, it will load the value 42 into `r2`—an observable effect since `r2` is mentioned in the `exists` clause on line 16—although by the rules of the C++ abstract machine the value must instead be zero.

Applied to Listing 3 on page 8, for another example, the principles dictate that if the load on line 8 reads from the store on line 18 and the load on line 16 reads from the store on line 10 (as they do in the execution described by the listing's `exists` clause), then each of the stores must commit before its corresponding load. A further application says that neither line 10 nor line 12 can commit until after the load in line 8 (and also the conditional on line 9) has committed. Similarly, neither line 18 nor line 20 can commit

[16] An early ARMv8 memory model did allow loads that read from an earlier store in the same thread to commit before the store. This apparent exception merely reflects a difference in nomenclature; in the memory model a store was said to commit at the time when it made its new value available to other threads, whereas we say that a store commits at the time when the CPU has irrevocably decided that it will take place with a particular fully determined value and address.

[17] Yes, `y[-1]` is UB, but the CPU neither knows nor cares, nor should it.

```
 1 C speculative-store
 2 {
 3   [x] = 0;
 4   [y] = { 0 };
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int y[], atomic_int *z) {
 9   int r1 = atomic_load_explicit(z, memory_order_relaxed);
10   // Speculatively executed store
11   atomic_store_explicit(&y[r1], 42, memory_order_relaxed);
12   // Non-speculatively executed load
13   int r2 = atomic_load_explicit(x, memory_order_relaxed);
14 }
15
16 exists(0:r2=42)
```

Listing 6: Speculated Store and Non-Speculated Load

until after line 16 (and 17) has committed. Taken together, these constraints imply that the OOTA cycle in the listing cannot be realized, because to do so would require that none of the machine instructions corresponding to lines 8, 10, 16, and 18 could commit before the others. Furthermore, it's not possible to circumvent this reasoning by suggesting that some of those lines could commit at the same time, because instructions take time to execute even when executing speculatively.

Yes, this does mean that hardware can produce OOTA cycles during speculative execution, but the hardware is required to prevent such speculative cycles from committing. As a special case of this requirement, if a load obtains a value from a speculated store that has not yet committed, and the store gets squelched, then the load must be restarted before it commits even if the load and the store are executed by different CPUs.

The requirement applies within a single multithreaded core as well as between cores. The fact that intracore communication is faster than intercore communication does not magically give loads the ability to commit before the stores they read from, no matter what cores they execute on.

Another feature we can expect of upcoming computer systems is the addition of new hardware instructions. For example, one could imagine a conditional-store instruction, similar to existing conditional-move instructions but carrying out a store to memory rather than a move to a register. The condition would control whether or not the store takes place. A compiler doing single-thread analysis could use a conditional-store instruction to generate code corresponding to lines 10 and 11 of Listing 29 on page 72 without use of a conditional branch, potentially enabling more hardware optimizations than would otherwise be permitted. Nevertheless, the conditional store could not be committed until the instructions it depends on (lines 8 and 9) have committed. Any new instruction will be subject to this constraint, just as the instructions in current systems are.

A key conclusion to draw from this discussion is that regardless of how advanced a computer system may be, it cannot commit a store having a hardware dependency on some set of loads until after (in global time) those loads have committed.

7.2 Constraints of the Standard

Volatile atomic accesses constitute observable behavior [13, `intro.abstract`] and must be executed in strict accordance with the rules of the C++ abstract machine. This means that a volatile atomic store in the source code corresponds directly to a machine-code store instruction, which can commit only after all loads whose values are used to compute the store's address and value have committed. OOTA aficionados will recognize this as a special case of ordering relaxed loads before relaxed stores, albeit one not requiring expensive memory-fence instructions on weakly ordered architectures.

Although C++ nonvolatile accesses to atomic objects are not observable behavior, any compiler that restricts its code analysis to a single thread must assume (unless it can prove otherwise) that a given relaxed atomic store can affect observable behavior in a different thread that loads the value stored. This assumption does not constrain the compiler to anywhere near the extent that a volatile relaxed atomic store would, but it does add significant constraints over those related to nonvolatile non-atomic stores.

For example, nonvolatile relaxed atomic accesses are subject to the C++ memory model. The "order-preserving" bullet point from Section 4.4 prohibits reordering of quasi-volatile accesses when they are to the same object, but such reordering is in any case directly forbidden by the memory model (the four coherence rules discussed in 6.9.2.2p14–19 `[intro.races]`). The "onto" bullet point also largely reiterates restrictions that follow from the standard, in particular, that compilers should not invent atomic stores, duplicate atomic stores, or invent atomic loads. On the other hand, it is permissible for compilers to omit redundant atomic stores or fuse nonvolatile atomic accesses of adjacent objects under appropriate circumstances; these topics correspond to the "total" and "one-to-one" bullet points of Section 4.4.

Invented Atomic Stores The reason that atomic stores should not be invented is that doing so can introduce new (and almost certainly undesirable) behaviors that are forbidden by the abstract machine. To see this, consider Listing 7, a `herd7` litmus test that demonstrates such a behavior. Without line 8, only the values 0 and 3 can be loaded into `r1` on line 13. With line 8, the additional value 42 can also be loaded into `r1`. The compiler is therefore forbidden from inventing the store of 42 unless it can prove that doing so does not negatively affect the program's observable behaviors. For example, the compiler might be able to prove that there are no other accesses to `x` at the time of the invented store—but it's very hard to imagine how a compiler that analyzes only a single thread at a time could prove such a thing.

Duplicated Atomic Stores Duplicating atomic stores is a special case of inventing them, but it is worth illustrating the fact that duplicating stores can also introduce new and undesirable behaviors. To see this, consider Listing 8, a litmus test that demonstrates such a behavior. Without line 9, if the value loaded into `r1` is one then the final value of `x` must be two. With line 9, the final value of `x` can be one even when the value loaded into `r1` is one. The compiler is therefore forbidden from duplicating atomic stores unless it can prove that doing so does not negatively affect the program's observable behaviors. Again, the compiler might be able to prove that there are no other accesses to `x` at the time of the duplicated store.

```
 1 C invented-store
 2 {
 3   [x] = 0;
 4 }
 5
 6 P0(atomic_int *x) {
 7   // Invented store followed by intended store.
 8   atomic_store_explicit(x, 42, memory_order_relaxed);
 9   atomic_store_explicit(x, 3, memory_order_relaxed);
10 }
11
12 P1(atomic_int *x) {
13   int r1 = atomic_load_explicit(x, memory_order_relaxed);
14 }
15
16 exists(1:r1=42)
```

Analysis by "herd7 -c11 litmus/invented-store.litmus":

```
 1 Test invented-store Allowed
 2 States 3
 3 1:r1=0;
 4 1:r1=3;
 5 1:r1=42;
 6 Ok
 7 Witnesses
 8 Positive: 1 Negative: 2
 9 Condition exists (1:r1=42)
10 Observation invented-store Sometimes 1 2
11 Time invented-store 0.00
12 Hash=5eb5631d90f3aa70212fcdb018817d8c
```

Listing 7: Example Invented Store

Invented Atomic Loads Although one can argue that it is functionally correct to invent atomic loads as long as the loaded value is not used, doing so can result in extra cache misses for no good purpose. And duplicating an atomic load can also lead to incorrect results if done carelessly. To see this, consider Listing 9. Suppose that the load on line 13 were duplicated, for example, by uncommenting line 15. Then the values of `a` and `b` passed to line 17's call to `do_something_with()` might be from different instances of the `foo` structure, something that `do_something_with()` probably isn't prepared to deal with.

For these reasons, C++ compilers should avoid inventing atomic loads, and in fact should avoid even duplicating them.

Omitted Redundant Atomic Stores In contrast, a pair of back-to-back nonvolatile atomic stores to the same object always might be executed such that no other thread accesses the object during the time between those two stores. This means that if the compiler omitted the first store, the user would have no way to prove this fact short of inspecting the assembly code. In cases where such omissions are undesirable, the user can resort to volatile atomic stores or to inline assembly[18] to prevent them.

Omitting a redundant store cannot create an OOTA cycle, because any valid execution (OOTA or not) of the program with the first of a back-to-back pair of nonvolatile

[18]For example, by placing the Linux-kernel `barrier()` macro between the two stores. This macro is an empty GCC `asm` that specifies the `memory` clobber.

```
C duplicated-store
{
  [x] = 0;
}

P0(atomic_int *x) {
  atomic_store_explicit(x, 1, memory_order_relaxed);
  // Duplicated store following intended store.
  atomic_store_explicit(x, 1, memory_order_relaxed);
}

P1(atomic_int *x) {
  int r1 = atomic_load_explicit(x, memory_order_relaxed);
  atomic_store_explicit(x, 2, memory_order_relaxed);
}

exists(1:r1=1 /\ x=1)
```

Analysis by "`herd7 -c11 litmus/duplicated-store.litmus`":

```
Test duplicated-store Allowed
States 4
1:r1=0; [x]=1;
1:r1=0; [x]=2;
1:r1=1; [x]=1;
1:r1=1; [x]=2;
Ok
Witnesses
Positive: 1 Negative: 5
Condition exists (1:r1=1 /\ [x]=1)
Observation duplicated-store Sometimes 1 5
Time duplicated-store 0.01
Hash=1e3a5591a2624bee2b8493ae633198c3
```

Listing 8: Example Duplicated Store

```
struct foo {
  int a;
  int b;
};
_Atomic struct foo *globalfoop;

void bar()
{
  int a;
  int b;
  struct foo *fp;

  fp = atomic_load_explicit(&globalfoop, memory_order_acquire);
  a = fp->a;
  // fp = atomic_load_explicit(&globalfoop, memory_order_acquire);
  b = fp->b;
  do_something_with(a, b);
}
```

Listing 9: Example Duplicated Load

Actual Execution	Tooling	Result
sdep	sdep	Ordered
	¬sdep	False Positive (Warning?)
¬sdep	sdep	False Negative (Warning?)
	¬sdep	Unordered

Table 1: Dependency Classification For Tools

relaxed atomic stores omitted is a valid execution of the program with the store present.

Fused Accesses of Adjacent Nonvolatile Atomic Objects Suppose a pair of adjacent nonvolatile atomic objects, when combined, form a single machine-word aligned and machine-word sized unit. Suppose further that the source code contains a pair of atomic loads, one from each of these objects, with the last of the pair being a relaxed load. Then the compiler could generate instead a single atomic load of the combined pair, with the memory-ordering semantics of the first load. A similar fact applies to pairs of stores.

This holds even when the atomic objects are treated as quasi volatile, providing an example of how the "one-to-one" bullet point of Section 4.4 does not apply to them.

7.3 Semantic Dependencies and Tooling

Where a C++ implementation's primary function is to correctly execute a C++ program, the function of tooling is often to evaluate properties of possible executions of that same program. While implementations should avoid breaking semantic dependencies, tools are in some cases permitted to misclassify them, as shown in Table 1.

Any misclassification will result in either a false positive or a false negative, except that many tools give some indication of a misclassification, for example, printing a warning message indicating that the code is too complex for it to analyze. Some projects would choose to restructure the code in such cases, on the grounds that code that is difficult for an automated tool to understand is also likely to be misunderstood by its all-too-human developers.

One advantage that tools often have over C++ implementations is the ability to devote much more computational power to the problem at hand. In fact, some tools respond to excessive complexity by consuming all available CPU and memory, which can also be interpreted as a good and sufficient warning message.

However, there is an important special case for tools that are closely associated with a C++ compiler. Such tools can simply use that compiler's classification of dependencies as semantic on the one hand or nonsemantic on the other, whether by working with the compiler's intermediate representations, examining binaries produced by the compiler, tracing the program's execution, or, in the case of dynamic tools, actually executing the program. Either way, this approach reduces the problem of analysis from the level of the C++ abstract machine to a much simpler lower level of abstraction. While taking this approach sacrifices significant generality, it also has the significant benefit of greatly reducing the cost of dependency analysis, potentially all the way down to zero.

8 Future Directions

This paper focuses solely on compilers, but it should be possible to extend these results to some classes of interpreters and JITs on the one hand and to link-time optimizations (LTO) on the other.

This paper focuses primarily on compilers that do only local per-thread analysis. Further explorations could consider additional classes of global analysis, for example, analyses that demonstrate that a given expression will always evaluate to a particular constant. JMM Causality Test Case 1 discussed in Appendix E.1 provides an example of such a global analysis.

This paper assumes that the user defines threads, and that the C++ implementation executes those threads unchanged. Future analyses might examine the possibility of "flattening" optimizations that combine multiple threads into one. (Note that such optimizations are more difficult than they might first appear.)

This paper shows that semantic dependencies are relative to specific executions of a program and the specific compiler in use rather than being determined solely by the source code. Future work might expand on Section 6.4.1, exploring special cases in which absolute semantic dependencies are functions just of the source code.

This paper considers only programs whose threads communicate using shared memory. Future work might include the effects of input and output or other operations which can vary from one execution to another, such as those based on the current time, process IDs, or pointer values (see Section 6.4.3).

Finally, future work might expand on Sections 6.4.2, and 6.4.4 so as to more precisely delineate the limits of permissible behavior for quasi-volatile object accesses.

9 Conclusion

This paper has presented samples of code containing non-trivial semantic dependencies, uncovering some shortcomings in typical definitions of "semantic dependency". It pointed out examples where semantic dependencies are a function of an execution rather than of the source code, and examples where a single semantic dependency extends from multiple loads to a store but not from any one of those loads. These complex dependencies motivated a generalization of the definition of "OOTA cycle", which is shown in Section 2.2.2.

This generalization, combined with a focus on compiler-based loose C++ implementations using single-thread analysis, and a consideration of the relation between source code and machine code, led to the formulation of the Fundamental Property of semantic dependency shown in Section 5.3. This in turn led to a precise definition of "semantic dependency" given in Section 6.1. This definition was used in Section 6.3 to demonstrate that these implementations are incapable of producing OOTA cycles in UB-free programs, provided they treat all nonvolatile atomic objects as quasi volatile, obeying the limitations outlined in Section 4.4.

A variant definition of "semantic dependency" given in Section 6.2, appropriate for compiler-based loose C++ implementations that don't restrict their code transformations to those based on single-thread analysis, was used to show that such implementations

are incapable of producing OOTA cycles in UB-free programs provided they treat all atomic objects as volatile.

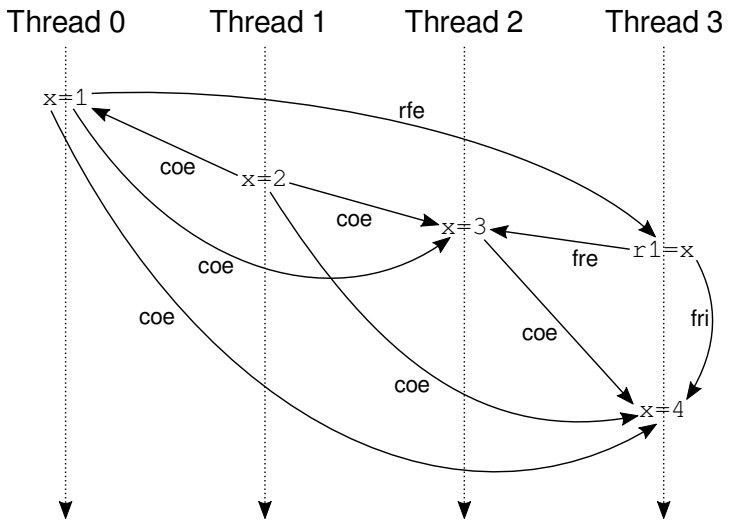

Figure 1: ITC Diagram for coe, fre, and rfe

A Interthread Communications

This section more precisely defines the "coe", "fre", and "rfe" types of interthread communication (also collectively called "links") and presents information about their temporal qualities.

First, naming. The final "e" in all three acronyms stands for "external" (that is, between threads) as opposed to "internal" communication (within a thread: "coi", "fri", and "rfi") and also as opposed to all communication, whether interthread or intrathread ("co", "fr", and "rf").

Next, definitions:

coe Coherence-order (external), which connects a store to any other store (in a different thread) to the same object that comes later in the object's modification order, i.e., that overwrites either the first store's value or some later value.

fre From-read (external), which connects a load to any store (in a different thread) to the same object that overwrites either the value that the load returned or some later value.

rfe Reads-from (external), which connects a store to any load (in a different thread) from the same object that returns the value stored. (Note: This means that the load retrieves the information written by the store, as opposed to retrieving the information from a different store that happened to write the same value.)

Thirdly, Figure 1 illustrates the coe, fre, fri, and rfe links for four threads with time advancing from the top to the bottom of the figure. Start with coe: Even though

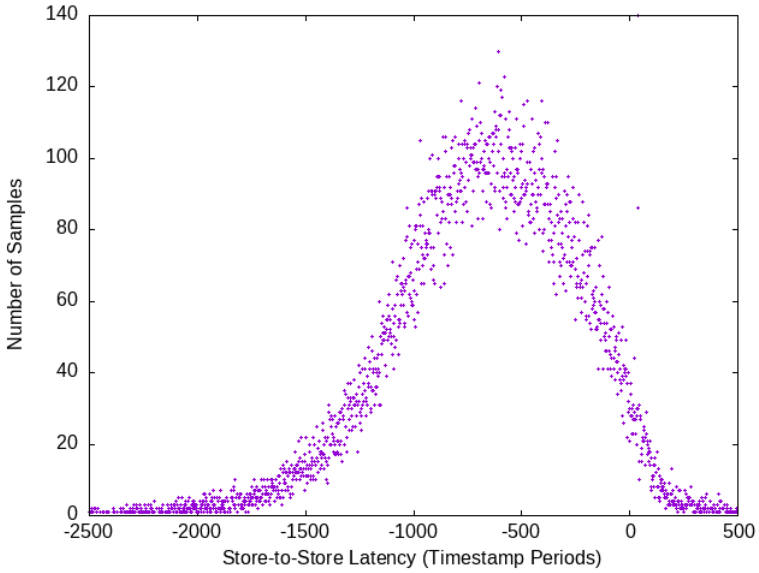

Figure 2: On x86, coe Links Are Atemporal

Thread 1's store is later in global time than that of Thread 0, the Thread 1 store comes first in `x`'s modification order, then Thread 0's store, then those of Threads 2 and 3. There is thus a coe link from Thread 1's store to every other store, from Thread 0's store to those of Threads 2 and 3, and from Thread 2's store to that of Thread 3. This situation can occur due to the initial placement and subsequent movement of cache lines.

Thread 3's load reads the value written by Thread 0's store, so there is an rfe link from that store to that load. In addition there is an fre link from the load to Thread 2's earlier store and an fri link to Thread 3's later store. Note that the fre link goes backwards in time.

Finally, the reader might desire hard evidence that coe and fre links really can go backwards in time. We provide this evidence on x86 to demonstrate that these effects are not confined to weakly ordered architectures. The machine is a dual-socket system with Intel(R) Xeon(R) Gold 6138 CPUs @ 2.00 GHz, each socket with 20 cores and each core having a pair of hardware threads, for a grand total of 80 hardware threads. The code generating this data may be found in the `CodeSamples/cpu` directory of the git archive of the book "Is Parallel Programming Hard, And, If So, What Can You Do About It?" [18].[19] See especially the `perftemporal.sh` script.

Figure 2 shows that coe links really are atemporal, that is, they can go backwards in time, and quite frequently at that, even on x86. During a run of the test, each of the threads stores a distinct value to the same atomic integer variable (all at about the same time), and each one records timestamps just before and just after its store. The "winning" store is the one whose value is retained in the variable at the end of the run,

[19] `git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git`

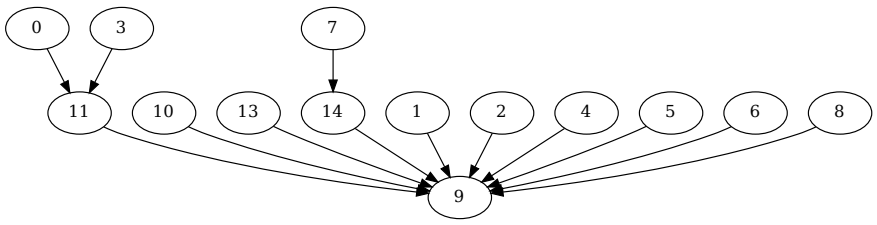

Figure 3: On x86, coe Links Form a Coherent Partial Order

overwriting all the others. The store-to-store latency is measured from the beginning of the nonwinning store that started latest to the end of the winning store. Every data point on the negative x-axis thus represents a group of runs in which a nonwinning store started after the winning store finished. This demonstrates that the winning store quite commonly is not the last one, even on x86.

Figure 3 shows that the sequence of values read from the atomic variable by each thread is consistent with a number of global orders, as required.[20] Working from the far left, one thread saw the values 0, 11, and 9, a second thread saw 3, 11, and 9, and so on. Although different threads saw different sequences of values, there is no disagreement on the order of the values that they could see. For example, all threads agree that the value 9 came last. This is an example of single-variable sequential consistency, and it meets the coherence requirements of the C++ standard.

Figure 4 plots a histogram of the elapsed time from the beginning of a load that returned an old value to the end of the store that provided the new value. Most of the data falls into negative time. In fact, in the most commonly occurring case the last load of an old value executes about 60 timestamp periods (about 30 nanoseconds) *after* the store which overwrote that old value. This shows that fre links can and do go backwards in global time, corresponding to the fre link between Thread 3 and Thread 2 in Figure 1.

Figure 5 plots a histogram of the elapsed time from the store of a new value to the first load of that new value. Here all of the data falls into positive time, indicating that rfe links always go forward in global time, as those familiar with computer hardware, limits on speculative execution, and the laws of physics would expect.[21]

Figure 6 shows how propagation delay explains the temporal properties of coe, fre, and rfe. The upper portion of the figure shows an atemporal coe, in which CPU 0 stores 1 to `x` after CPU 3 stores 2 but the modification order of `x` is decided after the fact. In this case, the value of 2 from CPU 3's store overwrites the value of 1 from CPU 0's store, despite the fact that CPU 0's store happened later in global time. One way this could happen is if the cache line containing `x` arrived at CPU 0 before arriving at CPU 3.

[20] Note that this data was not measured on the 80-thread system but rather on a 16-thread x86 laptop, to avoid an unreadable diagram containing 79 bubbles.

[21] Our apologies to those who might feel that this is belaboring the obvious. And please understand that these points have proven helpful to some of those whose work has never strayed too close to hardware.

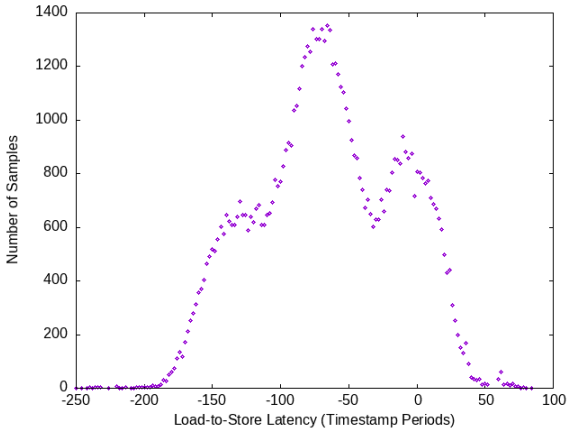

Figure 4: On x86, fre Links Are Atemporal

A more fanciful explanation is that both values arrived at an interconnect at the same time, and the interconnect made an arbitrary choice between the two. Either way, the end result is that an earlier store can overwrite a later store.

The middle portion of the figure shows an atemporal fre, in which CPU 3's load obtains the old value of zero from `x` despite having executed long after CPU 0's store, courtesy of the fact that the new value of `x` had not yet propagated to CPU 3.

The lower portion of the figure shows a temporal rfe. Because a load on CPU 3 cannot obtain the value stored by CPU 0 until after that value has propagated to CPU 3, the fact that CPU 3 obtains the new value implies that the load was executed after the store.

And this is exactly why the C++ memory model guarantees ordering from rfe links but not from coe and fre links for relaxed accesses (in the absence of other ordering from stronger atomic memory accesses or `atomic_thread_fence(memory_order_seq_cst)`).

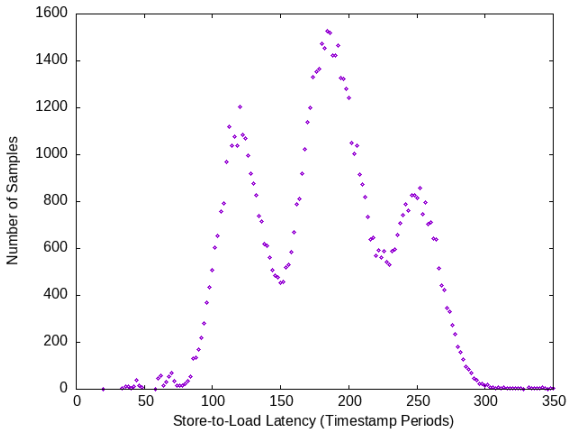

Figure 5: On x86, rfe Links Are Temporal

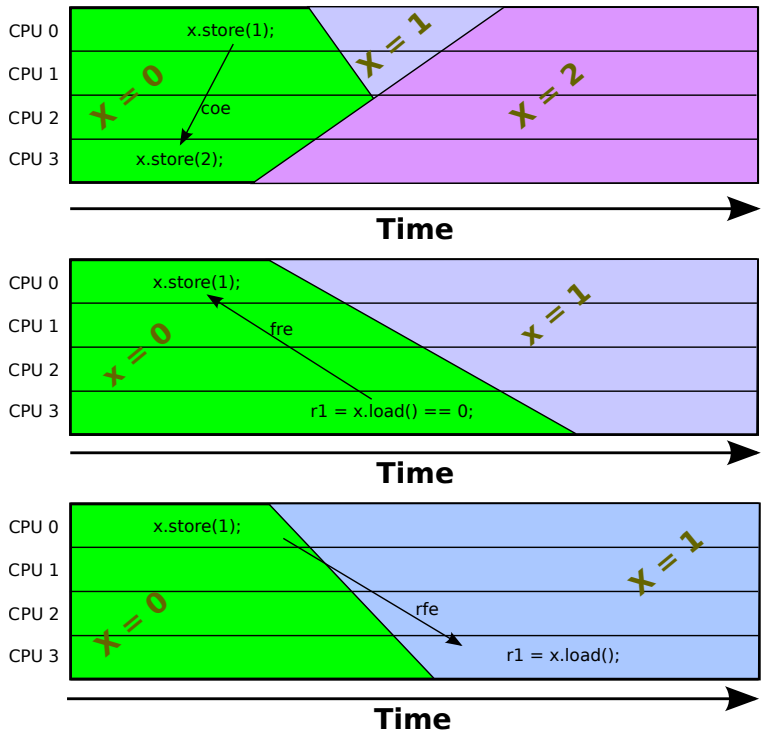

Figure 6: Propagation Delay and Temporal Properties of coe, fre, and rfe

B User Influence Over Language Semantics

As noted earlier, the exact definition of a computer language is subject to some debate, with standards, implementations, and users all having some degree of influence [23, 24]. It is natural to dismiss user influence when compared to the text of standards or the code in implementations, but both of these are subject to change and do change over time. An especially easy way for users to influence the implementation is by means of command-line flags and switch settings. As an example, consider the following command line used to compile the Linux kernel's `kernel/rcu/tree.c` C-language source file:

```
gcc -Wp,-MMD,kernel/rcu/.tree.o.d -nostdinc -I./arch/x86/include
-I./arch/x86/include/generated -I./include -I./arch/x86/include/uapi
-I./arch/x86/include/generated/uapi -I./include/uapi
-I./include/generated/uapi
-include ./include/linux/compiler-version.h
-include ./include/linux/kconfig.h
-include ./include/linux/compiler_types.h
-D__KERNEL__ -fmacro-prefix-map=./= -Werror -std=gnu11 -fshort-wchar
-funsigned-char -fno-common -fno-PIE -fno-strict-aliasing
-mno-sse -mno-mmx -mno-sse2 -mno-3dnow -mno-avx
-fcf-protection=branch -fno-jump-tables -m64 -falign-jumps=1
-falign-loops=1 -mno-80387 -mno-fp-ret-in-387
-mpreferred-stack-boundary=3 -mskip-rax-setup
-mtune=generic -mno-red-zone -mcmodel=kernel -Wno-sign-compare
-fno-asynchronous-unwind-tables -mindirect-branch=thunk-extern
-mindirect-branch-register -mindirect-branch-cs-prefix
-mfunction-return=thunk-extern -fno-jump-tables
-fpatchable-function-entry=16,16 -fno-delete-null-pointer-checks
-O2 -fno-allow-store-data-races -fstack-protector-strong
-fomit-frame-pointer -fno-stack-clash-protection -falign-functions=16
-fno-strict-overflow -fno-stack-check -fconserve-stack -Wall -Wundef
-Werror=implicit-function-declaration -Werror=implicit-int
-Werror=return-type -Werror=strict-prototypes -Wno-format-security
-Wno-trigraphs -Wno-frame-address
-Wno-address-of-packed-member -Wframe-larger-than=2048 -Wno-main
-Wno-unused-but-set-variable -Wno-unused-const-variable -Wvla
-Wno-pointer-sign -Wcast-function-type -Wno-array-bounds
-Wno-alloc-size-larger-than -Wimplicit-fallthrough=5
-Werror=date-time -Werror=incompatible-pointer-types
-Werror=designated-init -Wenum-conversion -Wno-unused-but-set-variable
-Wno-unused-const-variable -Wno-restrict -Wno-packed-not-aligned
-Wno-format-overflow -Wno-format-truncation -Wno-stringop-overflow
-Wno-stringop-truncation -Wno-missing-field-initializers
-Wno-type-limits -Wno-shift-negative-value -Wno-maybe-uninitialized
-Wno-sign-compare -DKBUILD_MODFILE='"kernel/rcu/tree"'
```

```
-DKBUILD_BASENAME='"tree"' -DKBUILD_MODNAME='"tree"'
-D__KBUILD_MODNAME=kmod_tree
-c -o kernel/rcu/tree.o kernel/rcu/tree.c
```

We do not propose to explain all of these, and sufficiently motivated readers can avail themselves of the GCC documentation. We instead look at representative members of several categories.

The `-funsigned-char` causes the `char` type to be unsigned, which overrides per-architecture defaults, some of which treat `char` as signed and others as unsigned. This choice prevents a class of bugs, and also allows the kernel to make reliable use of the uppermost bit of variables of type `char`. It also affects the definition of "semantic dependency" by changing the arithmetic proporties of this type. In theory, the standard could have specified the signedness of `char` but the variety of existing practice in the 1980s prevented this. Plus, optimizers were much less capable back then, so the signedness of `char` was much more of a performance concern than it is today.

The `-mno-sse` prevents GCC from making use of the processor's SSE hardware. This is done for performance reasons, as it avoids the overhead of saving and restoring the state of this hardware when switching between user and kernel contexts. Similarly, the `-mcmodel=kernel` causes the kernel binary to be placed in the uppermost 2GB of the address space, again reducing the overhead of switching between user and kernel contexts. These are cases where the standard does not specify anything, nor should it.

The `-fpatchable-function-entry=16,16` causes GCC to emit 16 `nop` instructions at the beginning of each function, with the function's entry point being just after this string of `nop` instructions. The resulting buffers are used by the Linux-kernel tracing infrastructure which is in turn used for debugging, performance measurement, and monitoring. This is again clearly outside the scope of the standard.

The `-fstack-protector-strong` causes GCC to emit code that provides some protection against some classes of attacks based on buffer overflows. One could rightly argue code should simply avoid ever overflowing buffers, but things like memory allocators and userspace memory accesses must use code that can be difficult to distinguish from buffer overflows. It is not clear that the ever-increasing variety of attacks should affect the standard.

The `-fno-strict-overflow` causes GCC to act as if signed integer overflow is defined behavior, which also affects the definition of "semantic dependency". This might be a controversial choice, and another option would be to add a new set of signed integer types to the standard for which overflow is defined as wrapping, similar to the situation with unsigned integers.

The `-Werror=strict-prototypes` causes GCC to warn if old-style non-ANSI function prototypes are used. This helps avoid certain classes of bugs. Warnings are by design outside the scope of the standard.

To sum up, user preference can exert a nontrivial influence over language semantics, and in particular can affect some aspects of the definition of "semantic dependency".

C But What About Tooling?

This paper focuses primarily on showing that, under commonly occurring constraints, OOTA cycles cannot form in real-world C++ implementations.

But what about tooling?

One entirely reasonable reaction is that, given the issues raised in Section 2.2, Section 4, and Section 6.4, load/store reordering is the least of the problems faced by tooling. Nevertheless, this appendix expands on Section 7.3 by looking into the possibility of:

- Tooling focusing on only part of the language,
- Changing the language to eliminate some aspects that are troublesome for tooling (and potentially accepting performance and energy-efficiency shortfalls), and
- Changing the language to better delineate those portions that tooling can easily accommodate.

But first, the next section looks at why some weakly ordered computer hardware appears to forbid reordering of prior relaxed loads with later relaxed stores,[22] despite the architecture permitting such reordering.

C.1 Load/Store Ordering: Hardware View for Software Hackers

Both the ARM and PowerPC architectural memory models permit reordering prior loads against subsequent stores, but actual tests on recent hardware fail to produce any evidence of such reordering. Nevertheless, thus far neither ARM nor PowerPC hardware architects have been willing to strengthen their memory models so as to forbid such reordering, despite a number of requests to do so [11].

This appendix offers some possible reasons for this odd juxtaposition of negative test results and flat refusal.

One key point is that some hardware (including ARM and PowerPC) provides precise exceptions. For example, if a given load results in a segmentation violation exception, that exception will occur before any instructions following that load have committed. This means that a given load instruction's execution must have proceeded beyond the point where an exception might occur before any subsequent store can be permitted to commit, even if that store is completely unrelated to that load. For example, the later store cannot commit until all prior loads' address translations have completed successfully.

However, if the load suffers a cache miss, address translation will have completed long before the load returns its value, meaning that a later store might well commit before the load completes. In fact, that later store might commit before the store which supplies the value returned by the load! Which explains why some hardware systems really do reorder prior loads and later stores.

The question then becomes "Why would weakly ordered systems fail to reorder prior loads with later stores?"

[22] Or, in memory-order-speak, forbid all load-buffering litmus tests.

One explanation is ECC errors.

If correctable ECC errors were fixed up in hardware, then only uncorrectable ECC errors would be directly visible to software. If the only possible reaction to an uncorrectable ECC error was to terminate the program suffering that ECC error, it would be safe to allow subsequent stores to commit as soon as all prior loads' address translations had completed successfully.

However, some kernels and applications have ways of handling even uncorrectable errors. Operating-system kernels encountering an ECC error might note that the affected data was being used by only one user process, and might react by killing that user process, taking care to account for memory shared with other processes. Some user applications might note that the corrupted data affected only a particular computation, and might react by restarting that computation. On the other hand, it is only reasonable to react to an uncorrectable ECC error in a load from a read-only mapped file by re-reading that data from that file, then restarting that load instruction. In such cases, the kernel and the user application would need to continue execution, and would thus require a precise exception.

Systems that offload processing of correctable ECC errors to software also require precise exceptions. After all, the software is going to need to be able to correct the error and then fix up the state to make it appear that the load had returned the fixed-up value.

ECC errors are detected near the end of their corresponding load instructions' execution, and so any need for precise exceptions rules out committing any subsequent stores until after the load has fetched (and possibly corrected) its value.

So why are hardware architects reluctant to tighten their memory models to forbid reordering of earlier loads and later stores?

If you would like an authoritative answer to this question, you should of course ask your friendly local hardware architect. In the meantime, here is some semi-informed speculation on this topic:

- Some hardware might choose to forego ECC, for example, in order to reduce cost for low-end systems or to improve energy efficiency for battery-powered systems.
- ECC error correction might still be done in hardware, avoiding the need for precise software-visible exceptions.
- Some systems might prefer to immediately shut down in response to an uncorrectable ECC error, perhaps due to safety considerations. This would entirely avoid the need for software-visible ECC-related exceptions.[23]
- Someone might come up with a clever way of correcting ECC errors in firmware, avoiding the need for precise software-visible exceptions.
- As late as early 2024, some GPGPUs have been observed reordering earlier loads against later stores. Other hardware architects might therefore feel the need to keep this option open for their own systems.

[23] One motivation for immediate shutdown is that if there are uncorrectable errors, there might soon be errors that change one valid bit pattern to another valid bit pattern, which could result in a lack of safety.

So although it is not unreasonable to continue asking hardware vendors to tighten their memory models so as to prohibit reordering of earlier loads and later stores, it also would not be too surprising for them to continue to refuse.

C.2 Status Quo and Focused Tooling

The theoretical possibility of OOTA cycles causes some tools to have difficulty identifying precisely which outcomes are impossible on real-world C++ implementations. One approach is for tooling to reject programs containing instances of `memory_order_relaxed` and `memory_order_consume`, so that developers desiring their code to be analyzed by such tools would avoid using these `memory_order` values. The is strong precedent for this strategy, with the common prohibition against side effects in function arguments being but one example.

However, there is a large body of existing code that uses `memory_order_relaxed`, and it would be unfortunate if such tools could not be applied to this code.

Another approach would be to add flags to C++ implementations to cause `memory_order_relaxed` to be interpreted as either `memory_order_acquire` (for loads), `memory_order_release` (for stores), or `memory_order_acq_rel` (for read-modify-write operations). On TSO systems and on systems featuring precise ECC exceptions, this would allow tooling to be brought to bear without performance or energy-efficiency consequences beyond forgone optimizations involving memory reference reordering.

However, this would mean that programs compiled in this way would not be guaranteed to run correctly on weakly ordered systems that lack ECC (or that lack precise exceptions). In addition, this change is more strict than necessary, forbidding optimizations that tooling could in fact analyze. Therefore, the next two sections look at standardizing less severe restrictions.

C.3 Change Relaxed to Forbid Load Buffering

Tooling does not require relaxed accesses become fully acquire and/or release, but rather only that implementations be forbidden from reordering prior relaxed loads with subsequent relaxed stores, as has been suggested many times over the years [7, 6, 14].

This preserves portability while enabling tooling to handle all members of the `memory_order` enumeration, but inflicts some performance and energy-efficiency penalties [9] in code not requiring these ordering restrictions [19].

C.4 Add Load-Store Memory Order that Forbids Load Buffering

The addition of a `memory_order_load_store` member to the `memory_order` enumeration has been suggested starting many years ago [6, 16]. This does not resolve all shortcomings in the C++ memory model [22, 16], but it would provide a portable `memory_order` that minimally restricted compiler and hardware optimizations while still permitting full analyzability by current software tools.

This change would also leave `memory_order_relaxed` in place, allowing minimal-overhead accesses in fastpaths. These fastpaths would not be analyzable by generic tools, but could be handled by special-case tools that analyze the binaries to verify that required orderings are enforced by machine-language properties such as dependencies. And a tool that checks control dependencies has in fact been prototyped [12].

This suggests a combined strategy of adding `memory_order` members as needed to extend the reach of general-purpose tooling, while also identifying `memory_order_relaxed` idioms that are checked at the machine level using special-purpose tooling.[24]

[24] Kudos to Peter Sewell for clearly articulating this possibility as part of an overall strategy.

```
1 atomic<int> x, y;
2
3 void thread1()
4 {
5   int r1 = x.load(memory_order_relaxed);
6   y.store(r1, memory_order_relaxed);
7 }
```

Listing 10: Non-Volatile Accesses and Dependencies

```
1 volatile atomic<int> x, y;
2
3 void thread1()
4 {
5   int r1 = x.load(memory_order_relaxed);
6   y.store(r1, memory_order_relaxed);
7 }
```

Listing 11: Volatile Accesses and Dependencies

D Illustrative Litmus Tests

These litmus tests helped illuminate important aspects of the problem of defining sdep and identifying OOTA cycles. This appendix presents these tests in roughly decreasing order of importance, surprise, and illumination. It evaluates them using the `herd7` tool and also using manual analysis.

D.1 Semantic Dependencies and `volatile`

The Linux kernel uses volatile accesses to constrain the compiler, and it is worth looking at the example shown in Listing 10 to see how this works.

Because both `x` and `y` are nonvolatile, a C++ compiler is free to assume that the value loaded by line 5 will be either some value stored to `x` or its initial value. And because there are no stores to `x`, the only remaining possibility is the default-initialized value of zero. Therefore, the compiler is within its rights to substitute the constant zero for `r1` on line 6, eliminating the semantic dependency that would otherwise extend from line 5 to line 6.

One way to preserve this dependency is use of `volatile`, as shown in Listing 11. Now the compiler is forbidden from assuming that the value loaded by line 5 has any relation to any stores to or initialization of `x`.

In this case, the `volatile` keyword alerts the compiler to the possibility of interfering changes to `x` from outside the program, for example, due to `x` being:

- Allocated in an MMIO region of the address space.
- Modified by an unknown-to-the-compiler thread or signal handler.

- Subject to I/O-device DMA operations.
- Modified by debugger commands.
- Modified using facilities such as `/dev/mem`.
- Modified by as-yet-unwritten dynamically linked libraries.
- Modified by some other mechanism of the reader's choosing.

These possibilities force the compiler to preserve the semantic dependency from line 4 to line 5, and this preserved dependency helps prevent the formation of OOTA cycles in environments where there is no interference.[25]

The developer is free to arrange for `x` to be free of any interference, thus obtaining predictable behavior along with the preserved semantic dependency. However, this example invalidates the rough definition of semantic dependency from Section 1.1.1 because the value loaded from `x` will always be the initial value of zero, so that it is at best dubious to talk about any changes in the value loaded.

One way forward is to assume that interference could happen when determining which dependencies are semantic, and then do further analysis using that determination. This forms the basis for the volatile approach presented in Section 6.4.2.

D.2 Non-Trivial Semantic Dependencies

Most of the examples in this paper involve simple semantic dependencies connecting a single load to a single store. One exception appears in Appendix D.6, but this section will present a more difficult example.

Listing 12 shows an example that builds on the multiplication-by-zero discussion in Section 2.2.1 on page 12. Line 11 of `P0()` stores to `z` the product of the values loaded from `x` and `y` on lines 9 and 10, respectively. Similarly, lines 16 and 17 store to `x` and `y`, respectively, the value that line 15 loads from `z`. Because all three variables are initialized to zero, and because loads can be ordered before any store, there is an execution in which all three loads return the value zero, in which case all three stores will of course store the value zero. In any related execution where the value loaded from `y` remains zero, nonzero values loaded from `x` cannot affect the value stored by line 11 to `z`. Only executions which load different (that is, nonzero) values from both `x` and `y` can cause the value stored to `z` to change from zero to a nonzero value.

Therefore, as noted in Section 2.2.1, in an execution where the loads from both `x` and `y` return zero, there is no semantic dependency from the load from `x` to the store to `z`, nor is there a semantic dependency from the load from `y` to the store to `z`. However, there *is* a semantic dependency from the *combination* of the loads from `x` and `y` to the store to `z`. An example is given by the execution that satisfies the `exists` clause on line 20, which would be an OOTA cycle.

For further evidence that there is no semantic dependency from `y` to `z` in executions where the load from `x` is zero, see the equivalent[26] program shown in Listing 13. Here,

[25] When there is interference, the definition of "OOTA cycle" does not apply.

[26] But equivalent only because the atomics are all nonvolatile!

```
 1 C oota-mult-0
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   int r2 = atomic_load_explicit(y, memory_order_relaxed);
11   atomic_store_explicit(z, r1 * r2, memory_order_relaxed);
12 }
13
14 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
15   int r3 = atomic_load_explicit(z, memory_order_relaxed);
16   atomic_store_explicit(x, r3, memory_order_relaxed);
17   atomic_store_explicit(y, r3, memory_order_relaxed);
18 }
19
20 exists(0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
```

Analysis by "`herd7 -c11 litmus/oota-mult-0.litmus`":

```
 1 Test oota-mult-0 Allowed
 2 States 1
 3 0:r1=0; 0:r2=0; 1:r3=0;
 4 No
 5 Witnesses
 6 Positive: 0 Negative: 5
 7 Condition exists (0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
 8 Observation oota-mult-0 Never 0 5
 9 Time oota-mult-0 0.00
10 Hash=dc0cd2d300923fdda699b77d8cab9b88
```

Listing 12: OOTA Multiplication Example, Initial Value Zero

```
 1 C oota-mult-0-cond
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   int r2;
11   if (r1)
12     r2 = atomic_load_explicit(y, memory_order_relaxed);
13   else
14     r2 = 0;
15   atomic_store_explicit(z, r1 * r2, memory_order_relaxed);
16 }
17
18 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
19   int r3 = atomic_load_explicit(z, memory_order_relaxed);
20   atomic_store_explicit(x, r3, memory_order_relaxed);
21   atomic_store_explicit(y, r3, memory_order_relaxed);
22 }
23
24 exists(0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
```

Analysis by "`herd7 -c11 litmus/oota-mult-0-cond.litmus`":

```
 1 Test oota-mult-0-cond Allowed
 2 States 1
 3 0:r1=0; 0:r2=0; 1:r3=0;
 4 No
 5 Witnesses
 6 Positive: 0 Negative: 3
 7 Condition exists (0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
 8 Observation oota-mult-0-cond Never 0 3
 9 Time oota-mult-0-cond 0.00
10 Hash=41de4a0ad40a252f3a665e913e10b265
```

Listing 13: OOTA Conditional Multiplication Example, Initial Value Zero

```
 1 C oota-mult-1
 2 {
 3   [x] = 1;
 4   [y] = 1;
 5   [z] = 1;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   int r2 = atomic_load_explicit(y, memory_order_relaxed);
11   atomic_store_explicit(z, r1 * r2, memory_order_relaxed);
12 }
13
14 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
15   int r3 = atomic_load_explicit(z, memory_order_relaxed);
16   atomic_store_explicit(x, r3, memory_order_relaxed);
17   atomic_store_explicit(y, r3, memory_order_relaxed);
18 }
19
20 exists(0:r1=0 /\ 0:r2=0 /\ 1:r3=0)
```

Analysis by "herd7 -c11 litmus/oota-mult-1.litmus":

```
 1 Test oota-mult-1 Allowed
 2 States 1
 3 0:r1=1; 0:r2=1; 1:r3=1;
 4 No
 5 Witnesses
 6 Positive: 0 Negative: 5
 7 Condition exists (0:r1=0 /\ 0:r2=0 /\ 1:r3=0)
 8 Observation oota-mult-1 Never 0 5
 9 Time oota-mult-1 0.00
10 Hash=1d4226b931b416aec475704b1c92d27b
```

Listing 14: OOTA Multiplication Example, Initial Value 1

there is not even a load from y in executions where the load from x returns zero, so there cannot possibly be a semantic dependency involving this non-existent load. A similar transformation would load x only in executions where the load from y returned nonzero, which similarly illustrates the lack of a semantic dependency between x and z.

In both Listings 12 and 13, an OOTA cycle having non-zero values (such as those indicated by their respective exists clauses) would need to affect both the x and the y components of this compound semantic dependency. This situation poses a challenge to current work on semantic dependencies, which are typically restricted to a single load and store.

In contrast, consider Listing 14, which differs from Listing 12 only in the initial values (1 instead of 0) and the exists clause (0 instead of 1). Note that an execution satisfying the updated exists clause is again an OOTA cycle.

Starting with the intuitive execution where the values loaded from x and y on lines 9 and 10 are the value 1 (that is, nonzero), executions that differ only in the value loaded from either x or y will now affect the value stored to z by line 11. There is therefore one semantic dependency from the load from x to the store to z and another separate semantic dependency from the load from y to the store to z, which matches typical definitions of semantic dependency. Yet if the initial values were instead obtained as input, Listing 12 and Listing 14 could be thought of as being two different executions of the exact same program.

```
 1 C oota-mult3-1
 2 {
 3   [w] = 2;
 4   [x] = 2;
 5   [y] = 2;
 6   [z] = 2;
 7 }
 8
 9 P0(atomic_int *w, atomic_int *x, atomic_int *y, atomic_int *z) {
10   int r1 = atomic_load_explicit(w, memory_order_relaxed);
11   int r2 = atomic_load_explicit(x, memory_order_relaxed);
12   int r3 = atomic_load_explicit(y, memory_order_relaxed);
13   atomic_store_explicit(z, r1 * r2 * r3, memory_order_relaxed);
14 }
15
16 P1(atomic_int *w, atomic_int *x, atomic_int *y, atomic_int *z) {
17   int r4 = atomic_load_explicit(z, memory_order_relaxed);
18   atomic_store_explicit(w, r4, memory_order_relaxed);
19   atomic_store_explicit(x, r4, memory_order_relaxed);
20   atomic_store_explicit(y, r4, memory_order_relaxed);
21 }
22
23 exists(0:r1=1 /\ 0:r2=1 /\ 0:r3=1 /\ 1:r4=1)
```

Analysis by "`herd7 -c11 litmus/oota-mult3-0.litmus`":

```
 1 Test oota-mult3-1 Allowed
 2 States 2
 3 0:r1=2; 0:r2=2; 0:r3=2; 1:r4=2;
 4 0:r1=2; 0:r2=2; 0:r3=2; 1:r4=8;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 9
 8 Condition exists (0:r1=1 /\ 0:r2=1 /\ 0:r3=1 /\ 1:r4=1)
 9 Observation oota-mult3-1 Never 0 9
10 Time oota-mult3-1 0.01
11 Hash=175b736e7d4b4e5db15f171333472c9f
```

Listing 15: OOTA Three-Factor Multiplication Example, Initial Value Zero

Because the only two solutions to $x^2 = x$ are the values zero and one, Listings 12 and 14 each have only two OOTA cycles. Note well that although software cannot distinguish the initial values from those of an OOTA cycle having those same values, there is a very real difference in the corresponding executions. To wit, an execution not corresponding to an OOTA cycle will read at least one initialization value, while an execution corresponding to an OOTA cycle with those same values will read only from that cycle's stores. On the other hand, in a similar program having initial values of (say) two, both executions containing OOTA cycles would be distinguishable from the initial values.

Listing 15 goes one step further by multiplying the values obtained from three different loads. Because $x^3 = x$ has three solutions ($\{-1, 0, 1\}$), this execution could have up to three OOTA cycles.

Much more elaborate examples can be constructed, which raises the question of how to determine the set of semantic dependencies in a given execution of a given fragment of code. The short answer is that when running on a real-world system, any C++ program is a finite-state machine, which, unlike Turing-complete systems, can be analyzed, as demonstrated in Section 5.

One might ask why semantic dependencies can be so complicated when hardware dependencies are much more straightforward. The reason is that, unlike compilers, hardware:

- Respects dependencies even when they reduce to a constant,
- Optimizes much less aggressively, if at all, and
- Lacks undefined behavior.

This situation further underscores the per-execution nature of semantic dependencies as well as the need for an improved definition of "semantic dependency". In Section 6, we expand the traditional definition to include dependencies extending from groups of loads to a single store.

D.3 Why rfe Instead of Tried-And-True rf?

Listings 16 and 17 show the examples from Section 2.1 on page 11 in the form of litmus tests. A compiler is allowed to transform the program in Listing 16 to the form of Listing 17, which demonstrates that intrathread reads-from (rfi) links may be eliminated by the compiler and therefore should not be used in a definition of OOTA. This section provides a more involved example to drive the point home.

The first step in this direction is Listing 18, which expands Listing 31 (discussed in Appendix E.4) from two threads to three. The `herd7` tool reports OOTA values, and all threads have straightforward semantic dependencies from their loads to their stores.

Listing 19 revises Listing 18 so that the value stored to `y` is forced to be at most 17 and the value stored to `z` is forced to be at least 17. This of course means that the value stored to `z` must always be 17, as can be seen in the `herd7` output. (However, a single-thread analysis could not prove this fact.)

Listing 20 further revises the litmus test to flatten `P0()` and `P1()` into a single thread, so that the resulting `P0()` has an rfi link connecting the store to and load from

```
 1 C oota-3-2-proc
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   atomic_store_explicit(y, r1, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
14   int r2 = atomic_load_explicit(y, memory_order_relaxed);
15   atomic_store_explicit(z, r2, memory_order_relaxed);
16   int r3 = atomic_load_explicit(z, memory_order_relaxed);
17   atomic_store_explicit(x, r3, memory_order_relaxed);
18 }
19
20 locations [x;y;z]
21 exists(0:r1=17 /\ 1:r2=17 /\ 1:r3=17)
```

Analysis by "`herd7 -c11 litmus/oota-3-2-proc.litmus`":

```
 1 Test oota-3-2-proc Allowed
 2 States 2
 3 0:r1=S12; 1:r2=S12; 1:r3=S12; [x]=S12; [y]=S12; [z]=S12;
 4 0:r1=0; 1:r2=0; 1:r3=0; [x]=0; [y]=0; [z]=0;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 4
 8 Condition exists (0:r1=17 /\ 1:r2=17 /\ 1:r3=17)
 9 Observation oota-3-2-proc Never 0 4
10 Time oota-3-2-proc 0.01
11 Hash=be011d398f752228fa778c7ed9b3f0fc
```

Listing 16: OOTA With an Intrathread Store-Load Link

```
 1 C oota-3-2-proc-opt
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   atomic_store_explicit(y, r1, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
14   int r2 = atomic_load_explicit(y, memory_order_relaxed);
15   atomic_store_explicit(z, r2, memory_order_relaxed);
16   // int r3 = atomic_load_explicit(z, memory_order_relaxed);
17   atomic_store_explicit(x, r2, memory_order_relaxed);
18 }
19
20 locations [x;y;z]
21 exists(0:r1=17 /\ 1:r2=17 /\ 1:r3=17)
```

Analysis by "`herd7 -c11 litmus/oota-3-2-proc-opt.litmus`":

```
 1 Test oota-3-2-proc-opt Allowed
 2 States 2
 3 0:r1=S10; 1:r2=S10; 1:r3=0; [x]=S10; [y]=S10; [z]=S10;
 4 0:r1=0; 1:r2=0; 1:r3=0; [x]=0; [y]=0; [z]=0;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 4
 8 Condition exists (0:r1=17 /\ 1:r2=17 /\ 1:r3=17)
 9 Observation oota-3-2-proc-opt Never 0 4
10 Time oota-3-2-proc-opt 0.00
11 Hash=216c3aae839fa4152ca602f6f41739a6
```

Listing 17: OOTA With an Intrathread Store-Load Link, Optimized

```
 1 C oota-3proc
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   atomic_store_explicit(y, r1, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
14   int r2 = atomic_load_explicit(y, memory_order_relaxed);
15   atomic_store_explicit(z, r2, memory_order_relaxed);
16 }
17
18 P2(atomic_int *x, atomic_int *y, atomic_int *z) {
19   int r3 = atomic_load_explicit(z, memory_order_relaxed);
20   atomic_store_explicit(x, r3, memory_order_relaxed);
21 }
22
23 locations [x;y;z]
24 exists(0:r1=17 /\ 1:r2=17 /\ 2:r3=17)
```

Analysis by "`herd7 -c11 litmus/oota-3proc.litmus`":

```
 1 Test oota-3proc Allowed
 2 States 2
 3 0:r1=S12; 1:r2=S12; 2:r3=S12; [x]=S12; [y]=S12; [z]=S12;
 4 0:r1=0; 1:r2=0; 2:r3=0; [x]=0; [y]=0; [z]=0;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 8
 8 Condition exists (0:r1=17 /\ 1:r2=17 /\ 2:r3=17)
 9 Observation oota-3proc Never 0 8
10 Time oota-3proc 0.01
11 Hash=fab7159e60865f712a5844fbf5b1a7e7
```

Listing 18: Three-Process Version of JMM Causality Test Case 4

```
 1 C oota-whyrfe-3
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   int r2 = r1;
11   if (r2 > 17)
12     r2 = 17;
13   atomic_store_explicit(y, r2, memory_order_relaxed);
14 }
15
16 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
17   int r3 = atomic_load_explicit(y, memory_order_relaxed);
18   int r4 = r3;
19   if (r4 < 17)
20     r4 = 17;
21   atomic_store_explicit(z, r4, memory_order_relaxed);
22 }
23
24 P2(atomic_int *x, atomic_int *y, atomic_int *z) {
25   int r5 = atomic_load_explicit(z, memory_order_relaxed);
26   atomic_store_explicit(x, r5, memory_order_relaxed);
27 }
28
29 locations [x;y;z]
30 exists(0:r1=17 /\ 1:r3=17 /\ 2:r5=17)
```

Analysis by "`herd7 -c11 litmus/oota-whyrfe-3.litmus`":

```
 1 Test oota-whyrfe-3 Allowed
 2 States 3
 3 0:r1=0; 1:r3=0; 2:r5=0; [x]=0; [y]=0; [z]=17;
 4 0:r1=0; 1:r3=0; 2:r5=17; [x]=17; [y]=0; [z]=17;
 5 0:r1=17; 1:r3=0; 2:r5=17; [x]=17; [y]=17; [z]=17;
 6 No
 7 Witnesses
 8 Positive: 0 Negative: 7
 9 Condition exists (0:r1=17 /\ 1:r3=17 /\ 2:r5=17)
10 Observation oota-whyrfe-3 Never 0 7
11 Time oota-whyrfe-3 0.01
12 Hash=8fe42ada73440acb408eab0981627c29
```

Listing 19: Three-Process Version of JMM Causality Test Case 4 With Max and Min

```
 1 C oota-whyrfe
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   int r2 = r1;
11   if (r2 > 17)
12     r2 = 17;
13   atomic_store_explicit(y, r2, memory_order_relaxed);
14   int r3 = atomic_load_explicit(y, memory_order_relaxed);
15   int r4 = r3;
16   if (r4 < 17)
17     r4 = 17;
18   atomic_store_explicit(z, r4, memory_order_relaxed);
19 }
20
21 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
22   int r5 = atomic_load_explicit(z, memory_order_relaxed);
23   atomic_store_explicit(x, r5, memory_order_relaxed);
24 }
25
26 locations [x;y;z]
27 exists(0:r1=17 /\ 0:r3=17 /\ 1:r5=17)
```

Analysis by "`herd7 -c11 litmus/oota-whyrfe.litmus`":

```
 1 Test oota-whyrfe Allowed
 2 States 2
 3 0:r1=0; 0:r3=0; 1:r5=0; [x]=0; [y]=0; [z]=17;
 4 0:r1=0; 0:r3=0; 1:r5=17; [x]=17; [y]=0; [z]=17;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 3
 8 Condition exists (0:r1=17 /\ 0:r3=17 /\ 1:r5=17)
 9 Observation oota-whyrfe Never 0 3
10 Time oota-whyrfe 0.01
11 Hash=89aad1d6fb7f77eae05ac3f0cc73bcf8
```

Listing 20: Why rfe Instead of Tried-And-True rf?

`y`, rather than the rfe link in Listing 19. As a result, even a single-thread analysis could conclude that the value stored to `z` will always be 17, by examining `P0()` in isolation and using the fact that it is valid for a C++ compiler to assume the value loaded from `y` will always be the value just stored and so to omit the load.[27]

If a compiler makes this observation and optimizes the program by storing a constant 17 to `z` rather than going through the computations on lines 15–17 of Listing 20, it will generate an executable which could produce the all-17's result described by the `exists` clause. (See Listing 21, where the store to `z` has been moved up to line 9.[28]) This would not be an OOTA cycle, because in either form of the program the store to `z` in `P0` is not semantically dependent on the load from `x`, so there is no cycle in (sdep $\cup$ rfe). A different compiler that does not make this optimization will generate an executable

[27]This is not an option in C because in that language, atomic accesses are volatile and hence must be preserved in the machine code.

[28]The load from `y` on line 15 is retained even though it is not used, because `r3` appears in the `exists` clause and hence its final value is considered observable behavior.

```
 1 C oota-whyrfe-z17
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   atomic_store_explicit(z, 17, memory_order_relaxed);
10   int r1 = atomic_load_explicit(x, memory_order_relaxed);
11   int r2 = r1;
12   if (r2 > 17)
13     r2 = 17;
14   atomic_store_explicit(y, r2, memory_order_relaxed);
15   int r3 = atomic_load_explicit(y, memory_order_relaxed);
16 }
17
18 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
19   int r5 = atomic_load_explicit(z, memory_order_relaxed);
20   atomic_store_explicit(x, r5, memory_order_relaxed);
21 }
22
23 locations [x;y;z]
24 exists(0:r1=17 /\ 0:r3=17 /\ 1:r5=17)
```

Analysis by "herd7 -c11 litmus/oota-whyrfe-z17.litmus":

```
 1 Test oota-whyrfe-z17 Allowed
 2 States 3
 3 0:r1=0; 0:r3=0; 1:r5=0; [x]=0; [y]=0; [z]=17;
 4 0:r1=0; 0:r3=0; 1:r5=17; [x]=17; [y]=0; [z]=17;
 5 0:r1=17; 0:r3=17; 1:r5=17; [x]=17; [y]=17; [z]=17;
 6 Ok
 7 Witnesses
 8 Positive: 1 Negative: 3
 9 Condition exists (0:r1=17 /\ 0:r3=17 /\ 1:r5=17)
10 Observation oota-whyrfe-z17 Sometimes 1 3
11 Time oota-whyrfe-z17 0.01
12 Hash=0ac1705891c439cf09b4d06a27690eb6
```

Listing 21: Why rfe Instead of Tried-And-True rf When z=17?

which cannot produce the all-17's result at all, because of hardware dependencies and consequent instruction ordering in the machine code.

Either way, no OOTA cycle can occur. But what if we chose to define an OOTA cycle with (sdep ∪ rf) instead of (sdep ∪ rfe)?

In that case, it would be necessary to consider separately the dependency in Listing 20 connecting `P0()`'s load from `x` to its store to `y`, and that connecting its load from `y` to its store to `z`. The first is a genuine semantic dependency; the second might or might not be, depending on the implementation. A compiler that uses single-thread analysis and does not omit the load from `y` would conclude that it is, because of the possibility that the value loaded from `y` might be larger than 17. From this point of view, the all-17's result *would* be considered an OOTA cycle.

This example demonstrates another reason why defining OOTA cycles in terms of (sdep ∪ rf) is problematic. Doing so can lead to classifying a cycle as OOTA even though there is no genuine semantic dependency in one of the participating threads.

```
 1 C oota-no-invented-load
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed) != 0;
10   int r2 = atomic_load_explicit(y, memory_order_relaxed) != 0;
11   atomic_store_explicit(z, r1 == r2, memory_order_relaxed);
12 }
13
14 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
15   int r3 = atomic_load_explicit(z, memory_order_relaxed);
16   atomic_store_explicit(y, r3, memory_order_relaxed);
17 }
18
19 P2(atomic_int *x, atomic_int *y, atomic_int *z) {
20   atomic_store_explicit(x, 1, memory_order_relaxed);
21 }
22
23 locations [x;y;z]
24 exists(0:r1=1 /\ 0:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-no-invented-load.litmus`":

```
 1 Test oota-no-invented-load Allowed
 2 States 3
 3 0:r1=0; 0:r2=0; [x]=1; [y]=0; [z]=1;
 4 0:r1=0; 0:r2=0; [x]=1; [y]=1; [z]=1;
 5 0:r1=1; 0:r2=0; [x]=1; [y]=0; [z]=0;
 6 No
 7 Witnesses
 8 Positive: 0 Negative: 6
 9 Condition exists (0:r1=1 /\ 0:r2=1)
10 Observation oota-no-invented-load Never 0 6
11 Time oota-no-invented-load 0.01
12 Hash=a7b618b70ecf436570edce45bc5ffeec
```

Listing 22: No Invented Atomic Loads

D.4 Inventing Atomic Loads

This section contains several example litmus tests expanding on the material presented in Section 4.3. on page 19.

Consider Listing 22. At first glance, it might appear straightforward. `P0()`'s store to `z` clearly depends on its load from `y`, and `P1()`'s store to `y` clearly depends on its load from `z`. If both dependencies are semantic, the OOTA cycle cannot be realized.

But is `P0()`'s dependency truly semantic? Imagine what might happen if a C++ compiler is permitted to duplicate `P0()`'s load from `x` and then make use of both loads' values. This would allow a compiler to transform Listing 22 into Listing 23.

Because of the comparisons against zero on lines 9 and 10 of Listing 22, the values of `r1` and `r2` are known to be either zero or one. Hence if the values of `r1a` and `r1b` on lines 9 and 10 of Listing 23 differ (say because `P2()` changes the value of `x` during the time between the two loads) then one of them must be equal to the value that would be obtained for `r2`. If the compiler is further permitted to make a choice at runtime between the two values loaded from `x`, it could always choose to use for `r1` the one that

```
 1 C oota-load-invented
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1a = atomic_load_explicit(x, memory_order_relaxed) != 0;
10   int r1b = atomic_load_explicit(x, memory_order_relaxed) != 0;
11   int r1;
12   int r2;
13   if (r1a != r1b) {
14     atomic_store_explicit(z, 1, memory_order_relaxed);
15     r2 = atomic_load_explicit(y, memory_order_relaxed) != 0;
16     r1 = r2;
17   } else {
18     r1 = r1b;
19     r2 = atomic_load_explicit(y, memory_order_relaxed) != 0;
20     atomic_store_explicit(z, r1 == r2, memory_order_relaxed);
21   }
22 }
23
24 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
25   int r3 = atomic_load_explicit(z, memory_order_relaxed);
26   atomic_store_explicit(y, r3, memory_order_relaxed);
27 }
28
29 P2(atomic_int *x, atomic_int *y, atomic_int *z) {
30   atomic_store_explicit(x, 1, memory_order_relaxed);
31 }
32
33 locations [0:r1a;0:r1b;1:r3;x;y;z]
34 exists(0:r1=1 /\ 0:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-load-invented.litmus`":

```
 1 Test oota-load-invented Allowed
 2 States 6
 3 0:r1=0; 0:r1a=0; 0:r1b=0; 0:r2=0; 1:r3=0; [x]=1; [y]=0; [z]=1;
 4 0:r1=0; 0:r1a=0; 0:r1b=0; 0:r2=0; 1:r3=1; [x]=1; [y]=1; [z]=1;
 5 0:r1=0; 0:r1a=0; 0:r1b=1; 0:r2=0; 1:r3=0; [x]=1; [y]=0; [z]=1;
 6 0:r1=0; 0:r1a=0; 0:r1b=1; 0:r2=0; 1:r3=1; [x]=1; [y]=1; [z]=1;
 7 0:r1=1; 0:r1a=0; 0:r1b=1; 0:r2=1; 1:r3=1; [x]=1; [y]=1; [z]=1;
 8 0:r1=1; 0:r1a=1; 0:r1b=1; 0:r2=0; 1:r3=0; [x]=1; [y]=0; [z]=0;
 9 Ok
10 Witnesses
11 Positive: 1 Negative: 9
12 Condition exists (0:r1=1 /\ 0:r2=1)
13 Observation oota-load-invented Sometimes 1 9
14 Time oota-load-invented 0.01
15 Hash=51dd34ec0d2ed28f0cafc304e1e6336c
```

Listing 23: Atomic Loads Invented

would be equal to `r2` (see line 16). Then the value stored to `z` would always be one, as shown on line 14, where the store has been moved before the load from `y` on line 15 to emphasize the fact that it does not depend on that load. The dependency from `y` to `z` would not be semantic after all, and the outcome `r1 == r2 == 1` would indeed be possible (although it would not be an example of OOTA because of the lack of a semantic dependency).

This is a simple example of a more general phenomenon. Suppose z is given by some function f of x and y. Under what conditions can we say that the value of z doesn't depend on y? The standard answer is that this happens when there are values z_0 and x_0 such that for any y, we have $z_0 = f(x_0, y)$. But if we are allowed to choose from among multiple values of x, the situation gets more complicated. Then the answer would be that there is a value z_0 and a set X of values for x such that for any y, there is some $x \in X$ with $z_0 = f(x, y)$. In Listing 23, f is the equality function and X is simply the set $\{0, 1\}$; however, the reasoning applies in any situation where x ranges over a finite collection of possible values. And as the example shows, the more general condition can hold in situations where the simpler condition does not, indicating that our intuitive notions of semantic dependency are not adequate when there can be multiple loads of the same variable.

Clearly something has gone wrong if a valid (in loose C++) transformation like this one is capable of destroying what should be an obvious semantic dependency. One possible reaction is to declare that useful C++ compilers should never invent or duplicate nonvolatile relaxed atomic loads. But what if the initial code was as shown in Listing 24?[29] In this case, the transformation to Listing 23 would not require inventing or duplicating a load, but instead merely inventing a use for the previously discarded value in `r0`.

Let us drop the single-thread-analysis constraint for the moment, and suppose that the compiler is able to prove that the only modification of a given atomic variable is to atomically increment it, as is the case for `y` in Listing 25. Is it okay for a C++ compiler to transform this program into Listing 26? An argument in favor is that the value of `y` must have passed through the value 42 if `r0` and `r1` bracket that value. Arguments against might cite the added overhead of the invented load, or the possibility that the check for the value 42 is intended for statistical sampling.

There are a number of possible reactions to these situations:

1. Listing 22 looks like it has an OOTA cycle, but there is in fact no semantic dependency.

2. Attempts to use real-world constraints to prevent OOTA cycles are futile.

3. Compilers should not invent uses for values from nonvolatile relaxed atomic loads. If it turns out that there are useful optimizations that invent such uses, then such optimizations must be applied only with careful attention to the as-if rule. Some might argue that inventing comparisons between a pair of nonvolatile relaxed atomic loads from the same object should be permitted only if the two loads could be reordered to be adjacent to each other.

[29] Unused loads can easily be generated by complex macros or template metaprograms. Typically compilers then remove them, but they are not obliged to.

```
 1 C oota-unused-load
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r0 = atomic_load_explicit(x, memory_order_relaxed) != 0;
10   int r1 = atomic_load_explicit(x, memory_order_relaxed) != 0;
11   int r2 = atomic_load_explicit(y, memory_order_relaxed) != 0;
12   atomic_store_explicit(z, r1 == r2, memory_order_relaxed);
13 }
14
15 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
16   int r3 = atomic_load_explicit(z, memory_order_relaxed);
17   atomic_store_explicit(y, r3, memory_order_relaxed);
18 }
19
20 P2(atomic_int *x, atomic_int *y, atomic_int *z) {
21   atomic_store_explicit(x, 1, memory_order_relaxed);
22 }
23
24 locations [x;y;z]
25 exists(0:r1=1 /\ 0:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-unused-load.litmus`":

```
 1 Test oota-unused-load Allowed
 2 States 3
 3 0:r1=0; 0:r2=0; [x]=1; [y]=0; [z]=1;
 4 0:r1=0; 0:r2=0; [x]=1; [y]=1; [z]=1;
 5 0:r1=1; 0:r2=0; [x]=1; [y]=0; [z]=0;
 6 No
 7 Witnesses
 8 Positive: 0 Negative: 9
 9 Condition exists (0:r1=1 /\ 0:r2=1)
10 Observation oota-unused-load Never 0 9
11 Time oota-unused-load 0.01
12 Hash=d4779ac32c1b526e80e585491fa63fe8
```

Listing 24: Unused Extra Atomic Load

```
 1 C inc
 2 {
 3   [x] = 0;
 4   [y] = 41;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r0 = atomic_load_explicit(y, memory_order_relaxed);
 9   if (r0 == 42)
10     atomic_store_explicit(x, 1, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y) {
14   atomic_fetch_add_explicit(y, 1, memory_order_relaxed);
15   atomic_fetch_add_explicit(y, 1, memory_order_relaxed);
16 }
17
18 locations [0:r0]
19 exists(x=1)
```

Analysis by "herd7 -c11 litmus/inc.litmus":

```
 1 Test inc Allowed
 2 States 3
 3 0:r0=41; [x]=0;
 4 0:r0=42; [x]=1;
 5 0:r0=43; [x]=0;
 6 Ok
 7 Witnesses
 8 Positive: 1 Negative: 2
 9 Condition exists ([x]=1)
10 Observation inc Sometimes 1 2
11 Time inc 0.00
12 Hash=20a9539cadd844faec01b1b57ed8890d
```

Listing 25: Only Atomic Increment

```
 1 C inc-range
 2 {
 3   [x] = 0;
 4   [y] = 41;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r0 = atomic_load_explicit(y, memory_order_relaxed);
 9   int r1 = atomic_load_explicit(y, memory_order_relaxed);
10   if (r0 <= 42)
11     if (r1 >= 42)
12       atomic_store_explicit(x, 1, memory_order_relaxed);
13 }
14
15 P1(atomic_int *x, atomic_int *y) {
16   atomic_fetch_add_explicit(y, 1, memory_order_relaxed);
17   atomic_fetch_add_explicit(y, 1, memory_order_relaxed);
18 }
19
20 locations [0:r0;0:r1]
21 exists(x=1)
```

Analysis by "`herd7 -c11 litmus/inc-range.litmus`":

```
 1 Test inc-range Allowed
 2 States 6
 3 0:r0=41; 0:r1=41; [x]=0;
 4 0:r0=41; 0:r1=42; [x]=1;
 5 0:r0=41; 0:r1=43; [x]=1;
 6 0:r0=42; 0:r1=42; [x]=1;
 7 0:r0=42; 0:r1=43; [x]=1;
 8 0:r0=43; 0:r1=43; [x]=0;
 9 Ok
10 Witnesses
11 Positive: 4 Negative: 2
12 Condition exists ([x]=1)
13 Observation inc-range Sometimes 4 2
14 Time inc-range 0.01
15 Hash=3ef212d27b7b7e5e3eccf84e0c583af4
```

Listing 26: Only Atomic Increment, Extended

4. Any time a compiler might be tempted to invent uses for values of nonvolatile relaxed atomic loads from the same object, the loads should instead be merged into a single load. This reaction might be attractive to those who carefully consider the pointlessness of keeping two loads that return the same value on the one hand or the cache-miss overhead incurred when closely spaced loads return different values on the other.

5. Compilers that invent nonvolatile atomic loads or invent new uses for nonvolatile atomic loads can adversely affect observed behavior, for example, by introducing error into code attempting to do statistical sampling.

6. What other interesting situations might come to light?

There are situations where inventing loads from non-atomic objects is customary, for example, when hoisting a load out of a loop. In this case, compilers might hoist the load from `x` out of the loop:

```
do_something_with(x);
for (i = 0; i < limit; i++)
    y[i] += x;
```

A compiler might act as if the program had instead been this:

```
r1 = x;
do_something_with(r1);
for (i = 0; i < limit; i++)
    y[i] += r1;
```

Note that passing `x` to `do_something_with()` is important when `x` is not atomic, because this allows the compiler to assume that inventing additional accesses to `x` will not result in a data race. Without this assumption, the above transformation might introduce a data race when `limit` is less than or equal to zero.

But when `x` is a nonvolatile atomic, there are no worries about data races. In theory, if the `do_something_with()` call was not present the compiler could still make the transformation above with no qualms, especially given that the value loaded from `x` is discarded in the case where `limit` is less than or equal to zero. Except that now the data-race worry is replaced by a cache-miss worry; after all, optimizations are supposed to speed things up, not slow them down. A compiler might be wise to adhere to the same restriction that avoids data races for loads from non-atomic variables in order to avoid potentially expensive cache misses.

D.5 Undefined Behavior and Unwise Optimization

Many compilers carry out optimizations based on the assumption that undefined behavior (UB) cannot happen, and if not performed carefully, these optimizations may be invalid because of OOTA-like interactions. A simple example is shown in Listing 27.

On line 14 of this listing, the computation `1 / (r2 <= 0)` will yield one if `r2` is not positive and will result in UB otherwise (a divide-by-zero error). It may not be

```
 1 C oota-div-ub
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(y, memory_order_relaxed);
 9   atomic_store_explicit(x, r1, memory_order_relaxed);
10 }
11
12 P1(atomic_int *x, atomic_int *y) {
13   int r2 = atomic_load_explicit(x, memory_order_relaxed);
14   atomic_store_explicit(y, 1 / (r2 <= 0), memory_order_relaxed);
15 }
16
17 exists(0:r1=1 /\ 1:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-div-ub.litmus`":

```
 1 Test oota-div-ub Allowed
 2 States 2
 3 0:r1=0; 1:r2=0;
 4 0:r1=1; 1:r2=0;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 3
 8 Condition exists (0:r1=1 /\ 1:r2=1)
 9 Observation oota-div-ub Never 0 3
10 Time oota-div-ub 0.00
11 Hash=d16ec1f23db801f79588fc940db3207d
```

Listing 27: OOTA-Like Behavior Due To Divide-By-Zero UB

obvious, but the program is in fact UB-free—there are no abstract executions in which `r2` is greater than zero. To see why not, consider that in any such execution `P1()` would not store anything to `y`; it certainly would not store a positive value. This means that the value loaded on line 8 would have to be `y`'s initial value of zero, and so would the value stored to `x` on line 9, and hence it would not be possible for the load on line 13 to get a positive value for `r2` in the first place.

Even for a compiler analyzing `P1()` in isolation, it is clear that either line 14 will store one to `y` or else the division by zero will cause UB. This allows the compiler to assume that the store will always take place, on the grounds that UB cannot happen, and so the compiler may optimize the comparison and division by replacing the whole expression with a constant `1`. There will then be no dependency from the load on line 13 to the store and thus nothing forcing the store to execute after the load.

When the resulting program runs on a weakly ordered architecture, the execution could go as follows:

1. Line 14 executes out of order, storing the constant `1` to `y`.
2. Line 8 loads one from `y`, setting `r1` to one.
3. Line 9 stores one to `x`.
4. Line 13 loads one from `x`, setting `r2` to one.

The final result would satisfy the `exists` clause, exhibiting observable behavior (`r2` is one at the end) that no abstract execution of the original program could produce. (This could happen even if all of the atomic objects in Listing 27 were converted to volatile.)

This result shows that the proposed optimization was invalid. To prevent the unwanted behavior, the compiler would have to take an extra step to force the optimized store on line 14 to be ordered after the load on line 13, possibly by changing the load to a load-acquire or changing the store to a store-release. Hans Boehm has proposed this store-release alternative as his Interpretation B' [5].

In other words, the compiler would have to preserve the dependency-induced ordering that the UB-based optimization would otherwise destroy. It is tempting to consider using techniques that prevent back-propagation of UB, but these are ineffective in this case because the compiler can detect and exploit UB when considering line 14 in isolation.

D.6 Additional Litmus Tests

This section lists a few other litmus tests of interest:

- The litmus test at `https://github.com/paulmckrcu/oota/blob/master/litmus/oota-two-source.litmus` shows multiple overlapping OOTA cycles.
- The litmus test at `https://github.com/paulmckrcu/oota/blob/master/litmus/oota-non-lb.litmus` shows that OOTA cycles are not confined to the load-buffering (LB) pattern.
- The litmus test at `https://github.com/paulmckrcu/oota/blob/master/litmus/oota-invent-int-load.litmus` shows another invented-load scenario loosely based on Figure 3 on page 8. See Appendix D.4 for more discussion on this topic.

```
 1 C oota-causality-1
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   if (r1 >= 0)
10     atomic_store_explicit(y, 1, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y) {
14   int r2 = atomic_load_explicit(y, memory_order_relaxed);
15   atomic_store_explicit(x, r2, memory_order_relaxed);
16 }
17
18 exists(0:r1=1 /\ 1:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-1.litmus`":

```
 1 Test oota-causality-1 Allowed
 2 States 3
 3 0:r1=0; 1:r2=0;
 4 0:r1=0; 1:r2=1;
 5 0:r1=1; 1:r2=1;
 6 Ok
 7 Witnesses
 8 Positive: 1 Negative: 3
 9 Condition exists (0:r1=1 /\ 1:r2=1)
10 Observation oota-causality-1 Sometimes 1 3
11 Time oota-causality-1 0.00
12 Hash=6be662cf3bb39460321974142224a214
```

Listing 28: Causality Test Case 1

E Litmus Tests from “Causality Test Cases"

These tests might be two decades old, but they are still quite relevant.[30] We have translated them from Java to C++, and we evaluate them using the `herd7` tool and also using manual analysis. Please note that differences in the semantics of the two languages result in changes in verdict for some litmus tests.

E.1 Causality Test Case 1

Listing 28 shows causality test case 1, for which the `r1 == r2 == 1` result is to be allowed. And indeed, this result does show up in the output from the `herd7` tool.

Compilers that can prove the value of `x` is always nonnegative can also determine that there is no semantic dependency between `P0()`'s load from `x` on line 8 and its store to `y` on line 10, that is, no matter which of the possible values is loaded from `x`, the value 1 will always be stored to `y`. Relative to less-omniscient compilers, however, including those using single-thread analysis or treating atomic objects as volatile, there *is* a semantic dependency.

[30] `http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html`.

```
 1 C oota-causality-2
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   int r2 = atomic_load_explicit(x, memory_order_relaxed);
10   if (r1 == r2)
11     atomic_store_explicit(y, 1, memory_order_relaxed);
12 }
13
14 P1(atomic_int *x, atomic_int *y) {
15   int r3 = atomic_load_explicit(y, memory_order_relaxed);
16   atomic_store_explicit(x, r3, memory_order_relaxed);
17 }
18
19 exists(0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
```

Analysis by "herd7 -c11 litmus/oota-causality-2.litmus":

```
 1 Test oota-causality-2 Allowed
 2 States 3
 3 0:r1=0; 0:r2=0; 1:r3=0;
 4 0:r1=0; 0:r2=0; 1:r3=1;
 5 0:r1=1; 0:r2=1; 1:r3=1;
 6 Ok
 7 Witnesses
 8 Positive: 1 Negative: 4
 9 Condition exists (0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
10 Observation oota-causality-2 Sometimes 1 4
11 Time oota-causality-2 0.01
12 Hash=f74880e1a73b0fd96f460486985778df
```

Listing 29: Causality Test Case 2

Even omniscient compilers will observe a semantic dependency from `P1()`'s load from `y` on line 14 to its store to `x` on line 15.

Either way no OOTA cycle will occur, with omniscient compilers having no cycle at all because of the lack of an sdep link in `P0()`, and other compilers prevented by hardware dependency ordering from realizing the cycle.

E.2 Causality Test Case 2

Listing 29 shows causality test case 2, for which the `r1 == r2 == r3 == 1` result is to be allowed. This result appears in the `herd7` output.

Because `P0()`'s loads from `x` on lines 8 and 9 are unordered, a C++ compiler that treats atomics as quasi volatile can act as if the source code loaded from `x` only once and assigned the returned value to both `r1` and `r2`, merging the loads.[31] The compiler could then prove that the condition on line 10 would always true, and act as if the source code had unconditionally executed the relaxed store to `y` on line 11 independent of the value loaded from `x`. Without this dependency there would then be no OOTA cycle.

[31] But not in C, where relaxed atomic operations are volatile, and thus are observable behavior not subject to merging.

```
 1 C oota-causality-3
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   int r2 = atomic_load_explicit(x, memory_order_relaxed);
10   if (r1 == r2)
11     atomic_store_explicit(y, 1, memory_order_relaxed);
12 }
13
14 P1(atomic_int *x, atomic_int *y) {
15   int r3 = atomic_load_explicit(y, memory_order_relaxed);
16   atomic_store_explicit(x, r3, memory_order_relaxed);
17 }
18
19 P2(atomic_int *x, atomic_int *y) {
20   atomic_store_explicit(x, 2, memory_order_relaxed);
21 }
22
23 exists(0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-3.litmus`":

```
 1 Test oota-causality-3 Allowed
 2 States 7
 3 0:r1=0; 0:r2=0; 1:r3=0;
 4 0:r1=0; 0:r2=0; 1:r3=1;
 5 0:r1=0; 0:r2=2; 1:r3=0;
 6 0:r1=1; 0:r2=1; 1:r3=1;
 7 0:r1=2; 0:r2=0; 1:r3=0;
 8 0:r1=2; 0:r2=2; 1:r3=0;
 9 0:r1=2; 0:r2=2; 1:r3=1;
10 Ok
11 Witnesses
12 Positive: 2 Negative: 16
13 Condition exists (0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
14 Observation oota-causality-3 Sometimes 2 16
15 Time oota-causality-3 0.01
16 Hash=6f5e2952498167dd10179059501aff8f
```

Listing 30: Causality Test Case 3

For compilers that do not merge the two loads, the hardware dependency ordering arising from the conditional test on line 10 will prevent the realization of an OOTA cycle.

E.3 Causality Test Case 3

Listing 30 shows causality test case 3, for which the `r1 == r2 == r3 == 1` result is to be allowed. The `herd7` tool reports this result.

The analysis from the previous section applies here. Although the addition of `P2()` allows sequentially consistent executions in which `P0()`'s two loads from `x` return different values, it does not change the fact that those two loads can be merged.

```
 1 C oota-causality-4
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   atomic_store_explicit(y, r1, memory_order_relaxed);
10 }
11
12 P1(atomic_int *x, atomic_int *y) {
13   int r2 = atomic_load_explicit(y, memory_order_relaxed);
14   atomic_store_explicit(x, r2, memory_order_relaxed);
15 }
16
17 exists(0:r1=1 /\ 1:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-4.litmus`":

```
 1 Test oota-causality-4 Allowed
 2 States 2
 3 0:r1=S8; 1:r2=S8;
 4 0:r1=0; 1:r2=0;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 4
 8 Condition exists (0:r1=1 /\ 1:r2=1)
 9 Observation oota-causality-4 Never 0 4
10 Time oota-causality-4 0.00
11 Hash=422eed0591553d8682af4d914c32f7d7
```

Listing 31: Causality Test Case 4

E.4 Causality Test Case 4

Listing 31 shows causality test case 4, for which the `r1 == r2 == 1` result is to be forbidden. The `herd7` tool finds this OOTA result; the `S8` values given on line 3 of its output indicate that in the execution it describes, the variables `r1` and `r2` were never assigned any specific value, so they could just as well be equal to one as to anything else.

This test case is virtually the same as the Simple OOTA cycle discussed in Section 1.1.1 on page 5, the only difference being the final values of the variables.

E.5 Causality Test Case 5

Listing 32 shows causality test case 5, for which the `r1 == r2 == 1` and `r3 == 0` result is to be forbidden. The `herd7` tool reports this OOTA result in line 3 of its output.

This is the same as causality test case 4 with `P2()` and `P3()` added, but those routines play no role in the OOTA cycle. `P2()` has no loads and so cannot participate in an OOTA cycle in any case.

```
C oota-causality-5
{
  [x] = 0;
  [y] = 0;
  [z] = 0;
}

P0(atomic_int *x, atomic_int *y, atomic_int *z) {
  int r1 = atomic_load_explicit(x, memory_order_relaxed);
  atomic_store_explicit(y, r1, memory_order_relaxed);
}

P1(atomic_int *x, atomic_int *y, atomic_int *z) {
  int r2 = atomic_load_explicit(y, memory_order_relaxed);
  atomic_store_explicit(x, r2, memory_order_relaxed);
}

P2(atomic_int *x, atomic_int *y, atomic_int *z) {
  atomic_store_explicit(z, 1, memory_order_relaxed);
}

P3(atomic_int *x, atomic_int *y, atomic_int *z) {
  int r3 = atomic_load_explicit(z, memory_order_relaxed);
  atomic_store_explicit(x, r3, memory_order_relaxed);
}

exists(0:r1=1 /\ 1:r2=1 /\ 3:r3=0)
```

Analysis by "`herd7 -c11 litmus/oota-causality-5.litmus`":

```
Test oota-causality-5 Allowed
States 6
0:r1=S8; 1:r2=S8; 3:r3=0;
0:r1=S8; 1:r2=S8; 3:r3=1;
0:r1=0; 1:r2=0; 3:r3=0;
0:r1=0; 1:r2=0; 3:r3=1;
0:r1=1; 1:r2=0; 3:r3=1;
0:r1=1; 1:r2=1; 3:r3=1;
No
Witnesses
Positive: 0 Negative: 24
Condition exists (0:r1=1 /\ 1:r2=1 /\ 3:r3=0)
Observation oota-causality-5 Never 0 24
Time oota-causality-5 0.01
Hash=8dbd66c410a8f5f93ae61df6689a435a
```

Listing 32: Causality Test Case 5

```
 1 C oota-causality-6
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   if (r1 == 1)
10     atomic_store_explicit(y, r1, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y) {
14   int r2 = atomic_load_explicit(y, memory_order_relaxed);
15   if (r2 == 1) {
16     atomic_store_explicit(x, 1, memory_order_relaxed);
17   }
18   if (r2 == 0) {
19     atomic_store_explicit(x, 1, memory_order_relaxed);
20   }
21 }
22
23 exists(0:r1=1 /\ 1:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-6.litmus`":

```
 1 Test oota-causality-6 Allowed
 2 States 3
 3 0:r1=0; 1:r2=0;
 4 0:r1=1; 1:r2=0;
 5 0:r1=1; 1:r2=1;
 6 Ok
 7 Witnesses
 8 Positive: 1 Negative: 2
 9 Condition exists (0:r1=1 /\ 1:r2=1)
10 Observation oota-causality-6 Sometimes 1 2
11 Time oota-causality-6 0.00
12 Hash=e43632b24cc50595250670702a452ed5
```

Listing 33: Causality Test Case 6

E.6 Causality Test Case 6

Listing 33 shows causality test case 6, for which the `r1 == r2 == 1` result is to be allowed. The `herd7` tool reports this result.

A compiler that can prove `y` is always either zero or one can also prove that there is no semantic dependency from `P1()`'s load on line 14 to its stores on lines 16 and 19. Such a compiler could act as if lines 15–20 had been replaced by a single store to `x`, showing that the dependency isn't semantic.

Less-omniscient compilers will preserve the dependency, and then hardware dependency ordering will prevent the OOTA cycle from being realized.

E.7 Causality Test Case 7

Listing 34 shows causality test case 7, for which the `r1 == r2 == r3 == 1` result is to be allowed. The `herd7` tool finds this result.

This test case has everything to do with reordering and nothing to do with OOTA cycles. The `r1 == r2 == r3 == 1` outcome happens when lines 17, 10, 11, 15,

```
C oota-causality-7
{
  [x] = 0;
  [y] = 0;
  [z] = 0;
}

P0(atomic_int *x, atomic_int *y, atomic_int *z) {
  int r1 = atomic_load_explicit(z, memory_order_relaxed);
  int r2 = atomic_load_explicit(x, memory_order_relaxed);
  atomic_store_explicit(y, r2, memory_order_relaxed);
}

P1(atomic_int *x, atomic_int *y, atomic_int *z) {
  int r3 = atomic_load_explicit(y, memory_order_relaxed);
  atomic_store_explicit(z, r3, memory_order_relaxed);
  atomic_store_explicit(x, 1, memory_order_relaxed);
}

exists(0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-7.litmus`":

```
Test oota-causality-7 Allowed
States 4
0:r1=0; 0:r2=0; 1:r3=0;
0:r1=0; 0:r2=1; 1:r3=0;
0:r1=0; 0:r2=1; 1:r3=1;
0:r1=1; 0:r2=1; 1:r3=1;
Ok
Witnesses
Positive: 1 Negative: 7
Condition exists (0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
Observation oota-causality-7 Sometimes 1 7
Time oota-causality-7 0.01
Hash=1dbef429cffcfc46cc318e1d8cbdf2c0
```

Listing 34: Causality Test Case 7

```
 1 C oota-causality-8
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   int r2 = 1 + r1 * r1 - r1;
10   atomic_store_explicit(y, r2, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y) {
14   int r3 = atomic_load_explicit(y, memory_order_relaxed);
15   atomic_store_explicit(x, r3, memory_order_relaxed);
16 }
17
18 locations [1:r3]
19 exists(0:r1=1 /\ 0:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-8.litmus`":

```
 1 Test oota-causality-8 Allowed
 2 States 2
 3 0:r1=0; 0:r2=1; 1:r3=0;
 4 0:r1=0; 0:r2=1; 1:r3=1;
 5 No
 6 Witnesses
 7 Positive: 0 Negative: 3
 8 Condition exists (0:r1=1 /\ 0:r2=1)
 9 Observation oota-causality-8 Never 0 3
10 Time oota-causality-8 0.00
11 Hash=ca5a44808a0bb9a3da519f93553afcc2
```

Listing 35: Causality Test Case 8

16 and 9 execute in that order. Note that this execution order respects the semantic dependencies in lines 10–11 and 15–16, and thus is not ruled out by hardware dependency ordering. It could have been ruled out if some of the accesses were acquire or release, but here they are all relaxed.

E.8 Causality Test Case 8

Listing 35 shows causality test case 8, for which the `r1 == r2 == 1` result is to be allowed. The `herd7` tool does not report this result, but it does if `r2` is initialized to one before being set to `1 + r1 * r1 - r1`.

Omniscient compilers might note that the computation on line 9 will have the value one when the load from `x` returns either zero or one, which are the only values `x` and `y` can take. Such a compiler could then realize that line 9 always sets `r2` to one, independent of the value loaded in line 8. Then there would be no semantic dependency between `P0()`'s load and store, and thus no OOTA cycle relative to this compiler.

A less-capable compiler would leave the dependency in place, and the resulting hardware dependency ordering would prevent the OOTA cycle from being realized.

```
 1 C oota-causality-9
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   int r2 = 1 + r1 * r1 - r1;
10   atomic_store_explicit(y, r2, memory_order_relaxed);
11 }
12
13 P1(atomic_int *x, atomic_int *y) {
14   int r3 = atomic_load_explicit(y, memory_order_relaxed);
15   atomic_store_explicit(x, r3, memory_order_relaxed);
16 }
17
18 P2(atomic_int *x, atomic_int *y) {
19   atomic_store_explicit(x, 2, memory_order_relaxed);
20 }
21
22 locations [1:r3]
23 exists(0:r1=1 /\ 0:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-9.litmus`":

```
 1 Test oota-causality-9 Allowed
 2 States 4
 3 0:r1=0; 0:r2=1; 1:r3=0;
 4 0:r1=0; 0:r2=1; 1:r3=1;
 5 0:r1=2; 0:r2=3; 1:r3=0;
 6 0:r1=2; 0:r2=3; 1:r3=3;
 7 No
 8 Witnesses
 9 Positive: 0 Negative: 10
10 Condition exists (0:r1=1 /\ 0:r2=1)
11 Observation oota-causality-9 Never 0 10
12 Time oota-causality-9 0.01
13 Hash=15d5fd5b06059a43b192e68ac04d6115
```

Listing 36: Causality Test Case 9

E.9 Causality Test Case 9

Listing 36 shows causality test case 9, for which the `r1 == r2 == 1` result is to be allowed. The `herd7` tool does not report this result.

This is the same as causality test case 8 but with `P2()` added; the store to `x` in `P2` can interfere with the formation of the OOTA cycle.

Unlike in causality test case 8, the computation on line 9 must be carried out because now the load on line 8 might return two. Consequently `P0` and `P1` both have semantic dependencies, and hardware dependency ordering will prevent the OOTA cycle from being realized.

E.10 Causality Test Case 9a

Listing 37 shows causality test case 9a, for which the `r1 == r2 == 1` result is to be allowed. The `herd7` tool does not report this result.

This is the same as causality test case 9 except that here `P2()` sets `x` to zero instead

```
C oota-causality-9a
{
  [x] = 2;
  [y] = 0;
}

P0(atomic_int *x, atomic_int *y) {
  int r1 = atomic_load_explicit(x, memory_order_relaxed);
  int r2 = 1 + r1 * r1 - r1;
  atomic_store_explicit(y, r2, memory_order_relaxed);
}

P1(atomic_int *x, atomic_int *y) {
  int r3 = atomic_load_explicit(y, memory_order_relaxed);
  atomic_store_explicit(x, r3, memory_order_relaxed);
}

P2(atomic_int *x, atomic_int *y) {
  atomic_store_explicit(x, 0, memory_order_relaxed);
}

locations [1:r3]
exists(0:r1=1 /\ 0:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-9a.litmus`":

```
Test oota-causality-9a Allowed
States 4
0:r1=0; 0:r2=1; 1:r3=0;
0:r1=0; 0:r2=1; 1:r3=1;
0:r1=2; 0:r2=3; 1:r3=0;
0:r1=2; 0:r2=3; 1:r3=3;
No
Witnesses
Positive: 0 Negative: 10
Condition exists (0:r1=1 /\ 0:r2=1)
Observation oota-causality-9a Never 0 10
Time oota-causality-9a 0.01
Hash=a2cbf5d7bfcd5ebd996c381edc8e4928
```

Listing 37: Causality Test Case 9a

of two. As as result the computation on line 9 is once again independent of the value loaded on line 8, which makes the analysis nearly the same as that of causality test case 8.

E.11 Causality Test Case 10

Listing 38 shows causality test case 10, for which the `r1 == r2 == 1` and `r3 == 0` result is to be forbidden. The `herd7` tool reports this result.

`P0()` and `P1()` have straightforward semantic dependencies between their respective loads and stores, and so hardware dependency ordering will prevent an OOTA cycle from being realized. `P2()` has no semantic dependency, and `P3()`'s store does not occur in this execution (i.e., the one described by the `exists` clause) because `r3` is zero. Thus neither of them contributes to an OOTA cycle.

E.12 Causality Test Case 11

Listing 39 shows causality test case 11, for which the `r1 == r2 == r3 == r4 == 1` result is to be allowed. The `herd7` tool reports this result.

Like causality test case 7, this test case has nothing to do with OOTA cycles. The result may be obtained by executing lines 20, 12, 13, 18, 19, 10, 11, and 17 in that order.

E.13 Causality Test Case 12

Causality Test Case 12 in Listing 40 uses arrays, which are not yet supported by the `herd7` tool.

An omniscient compiler could note that only `P0()` accesses array `a[]` and act as if the code was simpler, but there would still be semantic dependencies between `P0()`'s and `P1()`s relaxed atomic loads and stores involving `x` and `y`. The resulting hardware dependency ordering will prevent an OOTA cycle from being realized, whether the compiler is omniscient or not.

E.14 Causality Test Case 13

Listing 41 shows causality test case 13, for which the `r1 == r2 == 1` result is to be forbidden. The `herd7` tool reports this result.

This is the same as causality test case 10 without the confounding influence of `P2()` and `P3()`.

E.15 Causality Test Case 14

Listing 42 shows causality test case 14, for which the `r1 == r3 == 1` and `r2 == 0` result is to be forbidden. The `herd7` tool reports this result.

In the execution described by the `exists` clause, `P0()` has a semantic dependency from line 9 to line 13 and `P1()` has a semantic dependency from line 18 to line 21. (There's also a semantic dependency from line 17 to line 21 but it doesn't enter into this OOTA cycle, which involves only `x` and `y`.) The `memory_order_seq_cst`

```
 1 C oota-causality-10
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5   [z] = 0;
 6 }
 7
 8 P0(atomic_int *x, atomic_int *y, atomic_int *z) {
 9   int r1 = atomic_load_explicit(x, memory_order_relaxed);
10   if (r1 == 1) {
11     atomic_store_explicit(y, 1, memory_order_relaxed);
12   }
13 }
14
15 P1(atomic_int *x, atomic_int *y, atomic_int *z) {
16   int r2 = atomic_load_explicit(y, memory_order_relaxed);
17   if (r2 == 1) {
18     atomic_store_explicit(x, 1, memory_order_relaxed);
19   }
20 }
21
22 P2(atomic_int *x, atomic_int *y, atomic_int *z) {
23   atomic_store_explicit(z, 1, memory_order_relaxed);
24 }
25
26 P3(atomic_int *x, atomic_int *y, atomic_int *z) {
27   int r3 = atomic_load_explicit(z, memory_order_relaxed);
28   if (r3 == 1) {
29     atomic_store_explicit(x, 1, memory_order_relaxed);
30   }
31 }
32
33 exists(0:r1=1 /\ 1:r2=1 /\ 3:r3=0)
```

Analysis by "`herd7 -c11 litmus/oota-causality-10.litmus`":

```
 1 Test oota-causality-10 Allowed
 2 States 5
 3 0:r1=0; 1:r2=0; 3:r3=0;
 4 0:r1=0; 1:r2=0; 3:r3=1;
 5 0:r1=1; 1:r2=0; 3:r3=1;
 6 0:r1=1; 1:r2=1; 3:r3=0;
 7 0:r1=1; 1:r2=1; 3:r3=1;
 8 Ok
 9 Witnesses
10 Positive: 1 Negative: 7
11 Condition exists (0:r1=1 /\ 1:r2=1 /\ 3:r3=0)
12 Observation oota-causality-10 Sometimes 1 7
13 Time oota-causality-10 0.01
14 Hash=5a1f0ffe94e24aca297e3a00a718798f
```

Listing 38: Causality Test Case 10

```
C oota-causality-11
{
  [w] = 0;
  [x] = 0;
  [y] = 0;
  [z] = 0;
}

P0(atomic_int *w, atomic_int *x, atomic_int *y, atomic_int *z) {
  int r1 = atomic_load_explicit(z, memory_order_relaxed);
  atomic_store_explicit(w, r1, memory_order_relaxed);
  int r2 = atomic_load_explicit(x, memory_order_relaxed);
  atomic_store_explicit(y, r2, memory_order_relaxed);
}

P1(atomic_int *w, atomic_int *x, atomic_int *y, atomic_int *z) {
  int r4 = atomic_load_explicit(w, memory_order_relaxed);
  int r3 = atomic_load_explicit(y, memory_order_relaxed);
  atomic_store_explicit(z, r3, memory_order_relaxed);
  atomic_store_explicit(x, 1, memory_order_relaxed);
}

locations [w;x;y;z]
exists(0:r1=1 /\ 0:r2=1 /\ 1:r3=1 /\ 1:r4=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-11.litmus`":

```
Test oota-causality-11 Allowed
States 5
0:r1=0; 0:r2=0; 1:r3=0; 1:r4=0; [w]=0; [x]=1; [y]=0; [z]=0;
0:r1=0; 0:r2=1; 1:r3=0; 1:r4=0; [w]=0; [x]=1; [y]=1; [z]=0;
0:r1=0; 0:r2=1; 1:r3=1; 1:r4=0; [w]=0; [x]=1; [y]=1; [z]=1;
0:r1=1; 0:r2=1; 1:r3=1; 1:r4=0; [w]=1; [x]=1; [y]=1; [z]=1;
0:r1=1; 0:r2=1; 1:r3=1; 1:r4=1; [w]=1; [x]=1; [y]=1; [z]=1;
Ok
Witnesses
Positive: 1 Negative: 15
Condition exists (0:r1=1 /\ 0:r2=1 /\ 1:r3=1 /\ 1:r4=1)
Observation oota-causality-11 Sometimes 1 15
Time oota-causality-11 0.02
Hash=e80e124e0701dd2726eb79ba97a4bf3c
```

Listing 39: Causality Test Case 11

```
C oota-causality-12
{
  [x] = 0;
  [y] = 0;
  [a[0]] = 1;
  [a[1]] = 2;
}

P0(atomic_int a[], atomic_int *x, atomic_int *y) {
  int r1 = atomic_load_explicit(x, memory_order_relaxed);
  atomic_store_explicit(a[r1], 0, memory_order_relaxed);
  int r2 = atomic_load_explicit(a[0], memory_order_relaxed);
  atomic_store_explicit(y, r2, memory_order_relaxed);
}

P1(atomic_int a[], atomic_int *x, atomic_int *y) {
  int r3 = atomic_load_explicit(y, memory_order_relaxed);
  atomic_store_explicit(x, r3, memory_order_relaxed);
}

exists(0:r1=1 /\ 0:r2=1 /\ 1:r3=1)
```

Listing 40: Causality Test Case 12

```
C oota-causality-13
{
  [x] = 0;
  [y] = 0;
}

P0(atomic_int *x, atomic_int *y) {
  int r1 = atomic_load_explicit(x, memory_order_relaxed);
  if (r1 == 1)
    atomic_store_explicit(y, 1, memory_order_relaxed);
}

P1(atomic_int *x, atomic_int *y) {
  int r2 = atomic_load_explicit(y, memory_order_relaxed);
  if (r2 == 1)
    atomic_store_explicit(x, 1, memory_order_relaxed);
}

exists(0:r1=1 /\ 1:r2=1)
```

Analysis by "`herd7 -c11 litmus/oota-causality-13.litmus`":

```
Test oota-causality-13 Allowed
States 2
0:r1=0; 1:r2=0;
0:r1=1; 1:r2=1;
Ok
Witnesses
Positive: 1 Negative: 1
Condition exists (0:r1=1 /\ 1:r2=1)
Observation oota-causality-13 Sometimes 1 1
Time oota-causality-13 0.00
Hash=82dfdeefc1038a6b032706834feeedd9
```

Listing 41: Causality Test Case 13

```
C oota-causality-14
{
  [x] = 0;
  [y] = 0;
  [z] = 0;
}

P0(atomic_int *x, atomic_int *y, atomic_int *z) {
  int r1 = atomic_load_explicit(x, memory_order_relaxed);
  if (r1 == 0)
    atomic_store_explicit(z, 1, memory_order_seq_cst);
  else
    atomic_store_explicit(y, 1, memory_order_relaxed);
}

P1(atomic_int *x, atomic_int *y, atomic_int *z) {
  int r2 = atomic_load_explicit(z, memory_order_seq_cst);
  int r3 = atomic_load_explicit(y, memory_order_relaxed);
  int r4 = r2 + r3;
  if (r4)
    atomic_store_explicit(x, 1, memory_order_relaxed);
}

exists(0:r1=1 /\ 1:r2=0 /\ 1:r3=1 /\ ~1:r4=0)
```

Analysis by "`herd7 -c11 litmus/oota-causality-14.litmus`":

```
Test oota-causality-14 Allowed
States 3
0:r1=0; 1:r2=0; 1:r3=0; 1:r4=0;
0:r1=0; 1:r2=1; 1:r3=0; 1:r4=1;
0:r1=1; 1:r2=0; 1:r3=1; 1:r4=1;
Ok
Witnesses
Positive: 1 Negative: 2
Condition exists (0:r1=1 /\ 1:r2=0 /\ 1:r3=1 /\ not (1:r4=0))
Observation oota-causality-14 Sometimes 1 2
Time oota-causality-14 0.00
Hash=a662e071a6b2bba313609f96082d5610
```

Listing 42: Causality Test Case 14

specification on line 17 has no important effect; hardware dependency ordering will prevent the cycle from being realized.

E.16 Causality Test Case 15

Listing 43 shows causality test case 15, for which the `r0 == r1 == r3 == 1` and `r2 == 0` result is to be forbidden. The `herd7` tool reports this result.

The execution described by the `exists` clause has an OOTA cycle involving `u` and `v`. The dependency in `P0()` from line 13 to line 19 and the dependency in `P1()` from line 24 to line 27 are both semantic. The resulting hardware dependency ordering prevents the cycle from being realized.

Having no load, `P2()` cannot participate in an OOTA cycle.

E.17 Causality Test Case 16

Listing 44 shows causality test case 16, for which the `r1 == 2` and `r2 == 1` result is to be allowed. The `herd7` tool does not report this result, which is to be expected because it would violate the C++ memory model's read-write coherence rule regarding the modification order of an atomic object (6.9.2.2p17 `[intro.races]`). Because no abstract execution can obtain `r1 == 2` and `r2 == 1`, no valid realization will either.

E.18 Causality Test Case 17

Listing 45 shows causality test case 17, for which the `r1 == r2 == r3 == 42` result is to be allowed. The `herd7` tool does not report this result, although it does report a different OOTA cycle involving lines 11, 12, 16, and 17 (the `S17` value on line 3 of the `herd7` output).

A compiler that treats atomics as quasi volatile could reason as follows: Suppose that `P0()`'s load from `x` on line 8 returns the value 42. In that case, the second load on line 11 can be merged with the first, so that it also returns the value 42. On the other hand, if the load on line 8 returns some other value then line 10 will store the value 42 to `x`, and then the load on line 11 can be omitted in favor of using the value 42 stored by line 10.

Either way, the value of `r1` will be 42. Therefore the loads from `x` cannot affect the store on line 17, so there is no semantic dependency. This allows the cycle to be realized, but because the dependency in `P0()` is not semantic, the cycle is not OOTA.

E.19 Causality Test Case 18

Listing 46 shows causality test case 18, for which the `r1 == r2 == r3 == 42` result is to be allowed. The `herd7` tool does not report this result.

This is the same as causality test case 17, even to the alternate OOTA cycle reported by `herd7`, except that line 9 tests `r3` for equality to zero rather than non-equality to 42. Perhaps surprisingly, this makes a difference.

A compiler that treats atomic objects as volatile or uses single-thread analysis cannot assume that `x` will necessarily be equal to 42 following lines 9 and 10; it might have

```
C oota-causality-15
{
  [u] = 0;
  [v] = 0;
  [x] = 0;
  [y] = 0;
}

P0(atomic_int *u, atomic_int *v, atomic_int *x, atomic_int *y) {
  int r0 = atomic_load_explicit(x, memory_order_seq_cst);
  int r1;
  if (r0 == 1)
    r1 = atomic_load_explicit(u, memory_order_relaxed);
  else
    r1 = 0;
  if (r1 == 0)
    atomic_store_explicit(y, 1, memory_order_seq_cst);
  else
    atomic_store_explicit(v, 1, memory_order_relaxed);
}

P1(atomic_int *u, atomic_int *v, atomic_int *x, atomic_int *y) {
  int r2 = atomic_load_explicit(y, memory_order_seq_cst);
  int r3 = atomic_load_explicit(v, memory_order_relaxed);
  int r4 = r2 + r3;
  if (r4)
    atomic_store_explicit(u, 1, memory_order_relaxed);
}

P2(atomic_int *u, atomic_int *v, atomic_int *x, atomic_int *y) {
  atomic_store_explicit(x, 1, memory_order_seq_cst);
}

exists(0:r0=1 /\ 0:r1=1 /\ 1:r2=0 /\ 1:r3=1 /\ ~1:r4=0)
```

Analysis by "`herd7 -c11 litmus/oota-causality-15.litmus`":

```
Test oota-causality-15 Allowed
States 5
0:r0=0; 0:r1=0; 1:r2=0; 1:r3=0; 1:r4=0;
0:r0=0; 0:r1=0; 1:r2=1; 1:r3=0; 1:r4=1;
0:r0=1; 0:r1=0; 1:r2=0; 1:r3=0; 1:r4=0;
0:r0=1; 0:r1=0; 1:r2=1; 1:r3=0; 1:r4=1;
0:r0=1; 0:r1=1; 1:r2=0; 1:r3=1; 1:r4=1;
Ok
Witnesses
Positive: 1 Negative: 4
Condition exists (0:r0=1 /\ 0:r1=1 /\ 1:r2=0 /\ 1:r3=1 /\ not (1:r4=0))
Observation oota-causality-15 Sometimes 1 4
Time oota-causality-15 0.01
Hash=40c8856db50fe44f850390851a61a4ba
```

Listing 43: Causality Test Case 15

```
 1 C oota-causality-16
 2 {
 3   [x] = 0;
 4 }
 5
 6 P0(atomic_int *x) {
 7   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 8   atomic_store_explicit(x, 1, memory_order_relaxed);
 9 }
10
11 P1(atomic_int *x) {
12   int r2 = atomic_load_explicit(x, memory_order_relaxed);
13   atomic_store_explicit(x, 2, memory_order_relaxed);
14 }
15
16 exists(0:r1=2 /\ 1:r2=1)
```

Analysis by "herd7 -c11 litmus/oota-causality-16.litmus":

```
 1 Test oota-causality-16 Allowed
 2 States 3
 3 0:r1=0; 1:r2=0;
 4 0:r1=0; 1:r2=1;
 5 0:r1=2; 1:r2=0;
 6 No
 7 Witnesses
 8 Positive: 0 Negative: 4
 9 Condition exists (0:r1=2 /\ 1:r2=1)
10 Observation oota-causality-16 Never 0 4
11 Time oota-causality-16 0.00
12 Hash=511e9d0dd5696cc530c9476d5a541cb7
```

Listing 44: Causality Test Case 16

some other value such as 1. Therefore the compiler cannot conclude that `r1` will always be 42 and must leave the semantic dependency between lines 11 and 12 intact. Of course, the resulting OOTA cycle will not be realized, because of hardware dependency ordering.

E.20 Causality Test Case 19

Listing 47 shows causality test case 19, for which the `r1 == r2 == r3 == 42` result is to be allowed. The `herd7` tool does not report this result but it does report the OOTA cycle in line 3 of its output.

This is nearly the same as the Simple OOTA cycle and causality test case 4, with the addition of `P2()`. Just as in those examples, the OOTA cycle cannot be realized by compilers that treat atomic objects as volatile or that do single-thread analysis and treat atomic objects as quasi volatile.

E.21 Causality Test Case 20

Listing 48 shows causality test case 20, for which the `r1 == r2 == r3 == 42` result is to be allowed. The `herd7` tool does not report this result.

This is the same as causality test case 19, even to the OOTA cycle reported by `herd7`, except that line 19 tests `r3` for equality to zero rather than non-equality to 42.

```
 1 C oota-causality-17
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r3 = atomic_load_explicit(x, memory_order_relaxed);
 9   if (r3 != 42)
10     atomic_store_explicit(x, 42, memory_order_relaxed);
11   int r1 = atomic_load_explicit(x, memory_order_relaxed);
12   atomic_store_explicit(y, r1, memory_order_relaxed);
13 }
14
15 P1(atomic_int *x, atomic_int *y) {
16   int r2 = atomic_load_explicit(y, memory_order_relaxed);
17   atomic_store_explicit(x, r2, memory_order_relaxed);
18 }
19
20 exists(0:r1=42 /\ 1:r2=42 /\ 0:r3=42)
```

Analysis by "herd7 -c11 litmus/oota-causality-17.litmus":

```
 1 Test oota-causality-17 Allowed
 2 States 4
 3 0:r1=S17; 0:r3=0; 1:r2=S17;
 4 0:r1=0; 0:r3=0; 1:r2=0;
 5 0:r1=42; 0:r3=0; 1:r2=0;
 6 0:r1=42; 0:r3=0; 1:r2=42;
 7 No
 8 Witnesses
 9 Positive: 0 Negative: 7
10 Condition exists (0:r1=42 /\ 1:r2=42 /\ 0:r3=42)
11 Observation oota-causality-17 Never 0 7
12 Time oota-causality-17 0.01
13 Hash=2a23fb056b6cef48296f421deb01bf20
```

Listing 45: Causality Test Case 17

```
 1 C oota-causality-18
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r3 = atomic_load_explicit(x, memory_order_relaxed);
 9   if (r3 == 0)
10     atomic_store_explicit(x, 42, memory_order_relaxed);
11   int r1 = atomic_load_explicit(x, memory_order_relaxed);
12   atomic_store_explicit(y, r1, memory_order_relaxed);
13 }
14
15 P1(atomic_int *x, atomic_int *y) {
16   int r2 = atomic_load_explicit(y, memory_order_relaxed);
17   atomic_store_explicit(x, r2, memory_order_relaxed);
18 }
19
20 exists(0:r1=42 /\ 1:r2=42 /\ 0:r3=42)
```

Analysis by "`herd7 -c11 litmus/oota-causality-18.litmus`":

```
 1 Test oota-causality-18 Allowed
 2 States 4
 3 0:r1=S17; 0:r3=0; 1:r2=S17;
 4 0:r1=0; 0:r3=0; 1:r2=0;
 5 0:r1=42; 0:r3=0; 1:r2=0;
 6 0:r1=42; 0:r3=0; 1:r2=42;
 7 No
 8 Witnesses
 9 Positive: 0 Negative: 7
10 Condition exists (0:r1=42 /\ 1:r2=42 /\ 0:r3=42)
11 Observation oota-causality-18 Never 0 7
12 Time oota-causality-18 0.01
13 Hash=84e6c90bdd080bb17f6fb08f893dfc56
```

Listing 46: Causality Test Case 18

```
 1 C oota-causality-19
 2 {
 3   [x] = 0;
 4   [y] = 0;
 5 }
 6
 7 P0(atomic_int *x, atomic_int *y) {
 8   int r1 = atomic_load_explicit(x, memory_order_relaxed);
 9   atomic_store_explicit(y, r1, memory_order_relaxed);
10 }
11
12 P1(atomic_int *x, atomic_int *y) {
13   int r2 = atomic_load_explicit(y, memory_order_relaxed);
14   atomic_store_explicit(x, r2, memory_order_relaxed);
15 }
16
17 P2(atomic_int *x, atomic_int *y) {
18   int r3 = atomic_load_explicit(x, memory_order_relaxed);
19   if (r3 != 42)
20     atomic_store_explicit(x, 42, memory_order_relaxed);
21 }
22
23 exists(0:r1=42 /\ 1:r2=42 /\ 2:r3=42)
```

Analysis by "`herd7 -c11 litmus/oota-causality-19.litmus`":

```
 1 Test oota-causality-19 Allowed
 2 States 4
 3 0:r1=S8; 1:r2=S8; 2:r3=0;
 4 0:r1=0; 1:r2=0; 2:r3=0;
 5 0:r1=42; 1:r2=0; 2:r3=0;
 6 0:r1=42; 1:r2=42; 2:r3=0;
 7 No
 8 Witnesses
 9 Positive: 0 Negative: 16
10 Condition exists (0:r1=42 /\ 1:r2=42 /\ 2:r3=42)
11 Observation oota-causality-19 Never 0 16
12 Time oota-causality-19 0.01
13 Hash=1931891a93f192d9a672e1d755611c9d
```

Listing 47: Causality Test Case 19

```
C oota-causality-20
{
  [x] = 0;
  [y] = 0;
}

P0(atomic_int *x, atomic_int *y) {
  int r1 = atomic_load_explicit(x, memory_order_relaxed);
  atomic_store_explicit(y, r1, memory_order_relaxed);
}

P1(atomic_int *x, atomic_int *y) {
  int r2 = atomic_load_explicit(y, memory_order_relaxed);
  atomic_store_explicit(x, r2, memory_order_relaxed);
}

P2(atomic_int *x, atomic_int *y) {
  int r3 = atomic_load_explicit(x, memory_order_relaxed);
  if (r3 == 0)
    atomic_store_explicit(x, 42, memory_order_relaxed);
}

exists(0:r1=42 /\ 1:r2=42 /\ 2:r3=42)
```

Analysis by "`herd7 -c11 litmus/oota-causality-20.litmus`":

```
Test oota-causality-20 Allowed
States 4
0:r1=S8; 1:r2=S8; 2:r3=0;
0:r1=0; 1:r2=0; 2:r3=0;
0:r1=42; 1:r2=0; 2:r3=0;
0:r1=42; 1:r2=42; 2:r3=0;
No
Witnesses
Positive: 0 Negative: 16
Condition exists (0:r1=42 /\ 1:r2=42 /\ 2:r3=42)
Observation oota-causality-20 Never 0 16
Time oota-causality-20 0.01
Hash=b08a75dbf03d1d9caa8c2830d1c4b235
```

Listing 48: Causality Test Case 20

The same analysis applies.

F Acknowledgments

We are grateful to David Goldblatt, Jade Alglave, and Peter Sewell for their careful review of an early draft of this paper and to John Wickerson for asking Paul for a rant and taking the proffered rant seriously. We also owe David Goldblatt a debt of gratitude for his having asked an insightful question at the right time, his insights on combinations of OOTA and UB, and for his "Deathstation 9000" demonic CPU. Martin Uecker contributed valuable insights on backwards-propagating UB. Gonzalo Brito Gadeschi shared an alternative way of handling nonvolatile atomic operations. Richard Grisenthwaite patiently explained the architectural constraints that prevent hardware OOTA. Mark Batty encouraged our work on new OOTA-related litmus tests.

Nonetheless, all errors and omissions in this paper are the sole property of the authors, and the appearance of a name in this appendix does not in any way constitute agreement with anything in this paper.

References

[1] Jade Alglave, Luc Maranget, Pankaj Pawan, Susmit Sarkar, Peter Sewell, Derek Williams, and Francesco Zappa Nardelli. PPCMEM/ARMMEM: A tool for exploring the POWER and ARM memory models. `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/pldi105-sarkar.pdf`, June 2011.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), jul 2014.

[3] Mark Batty, Simon Cooksey, Scott Owens, Anouk Paradis, Marco Paviotti, and Daniel Wright. D1780R0: Modular relaxed dependencies: A new approach to the out-of-thin-air problem. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1780r0.html`, June 2019.

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66, Austin, TX, January 2011. `http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/help.html`.

[5] Hans Boehm. "Undefined behavior" and the concurrency memory model. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2215r0.pdf`, August 2020.

[6] Hans-J. Boehm. P1217R2: Out-of-thin-air, revisited, again. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1217r2.html`, June 2019.

[7] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, New York, NY, USA, 2014. ACM.

[8] Alex Celeste. Strict order of expression evaluation. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3203.htm, December 2023.

[9] Luke Geeson. A proposal fix for c/c++ relaxed atomics in practice. http://lukegeeson.com/blog/2023-10-17-A-Proposal-For-Relaxed-Atomics/, November 2023.

[10] David Goldblatt. There might not be an elegant OOTA fix. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf, October 2019.

[11] Richard Grisenthwaite. Views on relaxed atomics in C++ from Arm's technical leadership team. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-technical-view-on-relaxed-atomics, November 2023.

[12] Paul Heidekrüger. Status report: Broken dependency orderings in the Linux kernel. https://lpc.events/event/16/contributions/1174/, September 2022.

[13] Thomas Köppe. Working draft, standard for programming language C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf, May 2023.

[14] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. ACM.

[15] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 362–376, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Daniel Lustig. P1239r0: Placed before. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1239r0.html, October 2018.

[17] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, October 2012.

[18] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (2018.12.08a Release)*. kernel.org, Corvallis, OR, USA, 2018.

[19] Paul E. McKenney and Hans Boehm. P2055R0: A relaxed guide to memory_order_relaxed. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2055r0.pdf`, January 2020.

[20] Paul E. McKenney, Alan Jeffrey, and Ali Sezgin. N4323: Out-of-thin-air execution is vacuous. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4323.html`, November 2014.

[21] Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. P0422r0: Out-of-thin-air execution is vacuous. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html`, July 2016.

[22] Paul E. McKenney, Ulrich Weigand, Andrea Parri, and Boqun Feng. Linux-kernel memory model. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p0124r8.html`, August 2023.

[23] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: Elaborating the de facto standards. *SIGPLAN Not.*, 51(6):1–15, jun 2016.

[24] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 1–15, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Gabriel Dos Reis, Herb Sutter, and Jonathan Caves. Refining expression evaluation order for idiomatic C++. Available: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0145r3.pdf` [Viewed: February 13, 2024], June 2016.

[26] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Chasing away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 161–174, New York, NY, USA, 2017. ACM.