P3692R3: How to Avoid OOTA Without Really Trying

Alan Stern (stern@rowland.harvard.edu),
Paul E. McKenney (paulmckrcu@gmail.com),
Michael Wong, Maged Michael, Gonzalo Brito, Simon Cooksey, and Mark Batty
Other contributors: Hans Boehm

Abstract

Attempts to create memory models that forbid out-of-thin-air (OOTA) behaviors of C++ `memory_order_relaxed` accesses have been either non-executable, complex, or unloved by implementers. 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 hardware, including CPUs and GPGPUs, thus enabling analysis of semantic dependencies and OOTA behaviors by means of existing hardware models. We show that these models' constraints prevent OOTA cycles from occurring in undefined-behavior-free C++ programs running on such implementations, provided the cycles involve only volatile atomics. We accommodate nonvolatile atomics by defining "quasi-volatile" behavior and show that for existing implementations doing per-thread analysis and optimization, this behavior not only avoids OOTA but also is necessary to avoid incorrectly evaluating arithmetic expressions involving atomic accesses.

It follows that enforcing OOTA avoidance requires no special action from implementations and no changes to user code. As a consequence, we are asking for only for a small non-normative change to the standard.

Audience: SG1, EWG.

History

- P3692R3 fixes an "and and" typo, updates the proposed note to be added to the standard, and adds authors..
- P3692R2 applies feedback from the 2025 Sophia meeting, including reworking the requested change to the standard in terms of volatile semantics (courtesy of Olivier Giroux), states this paper's benefits to other OOTA researchers, and updates the author list.
- P3692R1 adds contact information, defines "preserved dependencies" that are a subset of syntactic and a superset of semantic dependencies, and makes numerous changes in the name of wordsmithing and clarification.
- This paper is a condensation of P3064R2 ("How to Avoid OOTA Without Really Trying") [26], which was criticized as being overly long. Voluminous off-list discussions are reflected. It also adds citations and clarifications.

1 Introduction and Background

We begin with a brief overview of the OOTA problem followed by an equally brief summary of prior OOTA work and of our approach.

1.1 Brief OOTA Overview

In broad terms, OOTA occurs 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" as shown in Listing 1 (JMM TC4).[1] Here, all of `x`, `y`, `r1`, and `r2` might have final values of 42, despite there being no instances of 42 in the default-zero initial values or in the executable code [23].[2] First, thread T0's line 1 does a load from `x` into `r1`, reading the value of T1's line-2 store rather than `x`'s initial value of zero and somehow obtaining 42. Second, T0's line 2 stores `r1`, and thus 42, to `y`. Third, T1's line 1 loads 42 from `y` to `r2`. Finally, T1's line 2 stores 42 to `x`, justifying the value loaded by T0's line 1.

Because nothing in the C++ memory model rules out such OOTA cycles (as indicated by the "sometimes satisfied" in the figure), the C++ standard explicitly recommends against them in 33.5.4p8 (`[atomics.order]`) [17]:

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

This prohibition prevents misapplication of the as-if rule in oracular C++ implementations. (The as-if rule permits optimizations for which the optimized code's observable behavior is the same as if it were unoptimized.)

	T0	T1
1.	`int r1 = x`	`int r2 = y`
2.	`y = r1`	`x = r2`

Condition (0:r1=42 ∧ 1:r2=42) sometimes satisfied.

Listing 1. Simple OOTA

[1] Java Memory Model Test Case 4 from http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html.

[2] Throughout this paper, we use names beginning with "r" for private per-thread variables ("registers"); other names denote shared variables. All shared-variable accesses are relaxed atomic.

We will use *full C++* to denote the standard including this prohibition of OOTA and *loose C++* to denote a hypothetical standard that excludes the prohibition but is otherwise identical to full C++. Unqualified *C++* means full C++.

Unfortunately, the standard does not explain what the phrase "depend on" in the quotation above really means. It is generally taken to mean *semantic dependency*, a somewhat vague concept. Roughly speaking, there is a semantic dependency from a given load to a given store in the same thread when *all other things being equal, a change in the value loaded can change the value stored, change the target address of the store, or prevent the store from occurring at all.* In Listing 1, for example, there is a semantic dependency from line 1 to line 2 in both T0 and T1. The semantic dependencies here are trivial, because the values stored simply *are* the values that were loaded.

A *syntactic* dependency, by contrast, exists whenever the value of the load appears in the source code as part of the computation leading up to the store, whether it affects the store or not. Although we will not give a precise definition of syntactic dependency here, doing so presents no fundamental difficulties. In fact, the C++ standard covers much of the same ground in its (now deprecated) definition of "carries a dependency" in 6.9.2.2p7 (`[Data races]`). Of course, all semantic dependencies are also syntactic. Note, however, that only semantic dependencies contribute to OOTA.

Since real-world CPUs cannot store something until they have determined its value,[3] the stores on line 2 of Listing 1 cannot take place until T0's and T1's CPUs know what values are returned by the loads on line 1. Thus the hardware orders these stores after their corresponding loads, and this ordering prevents the OOTA result.

Simple reordering can cause non-OOTA, but OOTA-like, behavior, as exemplified by Listing 2 [24]. Either the compiler or the CPU might reorder T1's lines 1 and 2, so that all of `x`, `y`, `r1`, and `r2` might end up with the value 42, as indicated by the "sometimes satisfied" in the figure.

On the other hand, OOTA cycles need not involve nonsensical or unmotivated values, as shown in Listing 3, adapted from [24]. Line 3 of both threads stores the value 42 only when line 2 determines that the value loaded was 42. As with Listing 1, hardware ordering prevents the OOTA result, as discussed further in Section 5.

	T0	T1
1.	`int r1 = x`	`int r2 = y`
2.	`y = r1`	`x = 42`

Condition (0:r1=42 ∧ 1:r2=42) sometimes satisfied.

Listing 2. Simple Reordering

	T0	T1
1.	`int r0 = y`	`int r1 = x`
2.	`if (r0 == 42)`	`if (r1 == 42)`
3.	`    x = 42`	`    y = 42`
4.	`else`	`else`
5.	`    x = r0`	`    y = r1`

Condition (0:r0=42 ∧ 1:r1=42) sometimes satisfied.

Listing 3. OOTA Cycle

1.2 Prior Work

All OOTA researchers owe a debt of gratitude to the foundational work led by William Pugh, represented by the infamous Java-language "Causality Test Cases"[4] [21]. The OOTA problem has been open for more than 20 years and has been shown to be quite difficult [3, 26].

Some executable C++ memory models correctly flag some OOTA cycles [1].[5] 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") [25] provided a decision procedure for distinguishing between reordering and OOTA, using a perturbation method based on the insight that OOTA cycles are fixed-point computations (in other words, an OOTA cycle must produce the same value that it consumes).

Some researchers recommend avoiding OOTA by forcing prior relaxed loads to be ordered before subsequent relaxed stores [5, 6, 11, 18, 19, 30], but this can require executing real instructions [13, 22, Section 7.1], consuming real time and real electrical power to solve a strictly theoretical problem. Furthermore, as discussed in Section 4.2, cross-thread optimizations can defeat this ordering constraint.

Other researchers recommend various procedures to identify and avoid OOTA cycles [2, 8, 9, 16, 18, 20, 31, 34], but none of these have been accepted by C++ implementers, in part due to these models requiring consideration of multiple executions. As of 2024, Mark Batty is working on modular relaxed dependencies, which may have the added benefit of checking compiler optimizations [33], a portion of the OOTA problem that is out of scope for this paper.[6]

Moiseenko et al. avoid cycles in po ∪ rf [29], which might permit more optimizations than forcing prior relaxed loads to be ordered before subsequent relaxed stores, but which the authors characterize as "(almost) a per-execution model". It remains to be seen whether this is close enough to a per-execution model to be accepted by C++ implementers.

[3] 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. A similar restriction applies to compiler-based speculation.

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

[5] Others avoid OOTA by forbidding atomic stores of nonconstant values [4].

[6] But please see Section 7.4.

Jagadeesan et al. combine preconditions and pomsets [15], putting forward a memory model with attractive properties, but which does not support ARMv7 or PowerPC.

Goldblatt looked at interactions between OOTA and undefined behavior (UB) [12]. We consider only UB-free examples.

All this work focused on either identifying OOTA or helping C++ implementations to avoid it.

1.3 Our Approach

Our approach applies real-world hardware constraints (including some imposed by long-standing laws of physics) to the problem of avoiding OOTA cycles. We do this by relating the C++ abstract machine to hardware executions, which of course limits the applicability of our results to C++ programs compiled and run on real-world computer systems. Given that a quarter century of OOTA research based on pure unaugmented C++ abstractions has resulted in (at best) incremental progress on the one hand, and that compiler-based real-world computer systems comprise an exceedingly important part of the overall OOTA problem on the other, our results are valuable despite these limitations. In the wise words of Peter J. Denning, “Maybe we need to occasionally descend from the high clouds of our abstractions to the concrete earthy concerns of everyday life” [10].

The key constraints mentioned above are: (1) Communicating data within a computer system takes time, and (2) Executing instructions that compute an output value that semantically depends on given input values also takes time. Because each link in an OOTA cycle consists of either data communication or computation of an output that semantically depends on an input, and because closing such a cycle requires the sequence of links to end at the same time as it starts, OOTA cycles cannot form in those C++ implementations that are within the scope of this paper. A great advantage of our approach is that we do *not* need to define semantic dependency purely at the level of C++ source code; rather, we are able to utilize information already provided by existing hardware execution models.

These simple-seeming notions conceal considerable complexity, and much of the remainder of this paper deals with corner cases that arise when carefully considering semantic dependencies, executions, C++ compilers, and hardware properties.

In short, this paper’s contribution is to show how these constraints allow real-world C++ implementations doing per-thread analysis to avoid OOTA behavior easily, in many cases with no cost at all [26]. In addition, our results simplify the job of OOTA researchers seeking practical impact by allowing them to ignore C++ implementers’ need for extreme simplicity and performance. This in turn permits such researchers to focus on advanced topics such as formal-verification tooling and evaluation of optimizations.

2 OOTA, Semantic Dependencies, and Reads-From

This section analyzes the main components of OOTA cycles, namely semantic dependencies and reads-from links.

2.1 OOTA: rf vs. 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. In principle, the time required by each of these links should prevent OOTA cycles from occurring. This idea has been formalized by defining an OOTA cycle as a cycle in sdep ∪ rf [24], where sdep is the set of semantic dependencies within each thread and rf is the set of store-to-load “reads-from” links, whether internally within a thread (rfi) or externally between threads (rfe).[7]

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 = y;
2. x = r2;
3. int r3 = x;
4. z = r3;
```

This is an elaboration of T1 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 x in between. Such an implementation might therefore behave as if line 3 had been removed from the source code and line 4’s r3 had been replaced by r2, causing line 4 always to store the same value to z as was stored to x by line 2. And since the load from x has been removed, it cannot possibly act as a temporal constraint.

To avoid this issue, we will substitute rfe for rf, defining an OOTA cycle—for now—as a cycle in sdep ∪ 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 naive argument against OOTA cycles. The primary difficulty lies in the fact that the code transformations performed by optimizing compilers can destroy syntactic dependencies, possibly even semantic ones (depending on one’s definition). That is, even when a syntactic 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 than the load it supposedly depends on, and thus no

[7] Additional background on temporal distinctions between coe, fre, and rfe maybe found elsewhere [26, Appendix A].

impediment to the occurrence of an OOTA cycle. We will see examples of this destruction later on.

2.2 Properties of Semantic Dependencies

Here we consider some of the complexities inherent to semantic dependencies.

2.2.1 Semantic Dependencies and Source Code. Some discussions of semantic dependencies assume that they are strictly properties of the source code. This assumption is not always valid; in fact, many instances of semantic dependency vary according to the details of particular executions. Consider for example:

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

Expanding on earlier discussions [6], as long as y is zero, changes in the value of x will not cause a change in the value stored to z. As a result, the semantic dependency from x to z exists only in executions where y 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 x and y are zero. Then a change to exactly one of x or y will not cause a change in the value stored to z. In other words, in this execution there is no semantic dependency from either x or y to z. But there *is* a semantic dependency from the *pair* {x, y} to z, because changes to both x and y can cause the value stored in z to change. This means that, prior work [25] notwithstanding, accurate definitions of sdep cannot always rely on single-variable perturbations; they must at times consider changes to multiple variables.

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 OOTA again; we will say that an execution is an instance of OOTA if in that execution:

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

This can make 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[8] 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 only the value of a given store, but also whether or not that store is executed at all.

2.2.5 Semantic Dependencies Not Affected by `if` Statements. Compare this example [14] 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.

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:

```
1. if (x > 0) {
2.     y = 42;
3. } else {
4.     y = 53;
5.     y = 42;
6. }
```

Consider an execution in which x is greater than zero, so line 2 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 on line 2?

One way to cut the Gordian knot is to match up the stores by the order they occur: Since line 2 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

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

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 line 2, and so there would not be a semantic dependency. Once again, the decision is up to the implementation.

2.2.7 Semantic Dependencies and Function Calls.

Boehm and Dempsky [6, Section 5] point out that arbitrary function calls can pose challenges for any attempt to determine semantic dependencies based solely on analysis of the source code, using the following example:

```
y = f(x);
```

And it is indeed impossible to determine whether or not there is a semantic dependency from x to y without reference to the definition of f(). In addition, f might well be a pointer to a function, in which case the existence of a semantic dependency from x to y might well vary at runtime as the value of function pointer f changes.

2.2.8 Semantic Dependencies and Free Choices.

Consider the following example, with variable 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.[9]

(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++ [32] and C [7]. 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.)

2.2.9 Semantic Dependencies and Architectural Variations.

In some cases, the standard permits architectures to differ as to whether an execution contains a semantic dependency. For 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 architecturally a signed type, but not those for which it is unsigned.

We have seen several examples showing that semantic dependencies can be more complicated than might first appear, varying according to the execution and even the implementation. This raises several questions, of which the first is…

3 What is an Execution?

This section looks more carefully at executions from the viewpoints of an abstract machine and the computer hardware, and then reconciles 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 (see Section 2.2.9).
- 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 (see Section 2.2.8).
- 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. In short, the standard grants C++ implementations considerable freedom, as detailed in Section 4.

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

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

ordered series of steps to be carried out by the abstract machine. However, with a few exceptions[10] 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. (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.)

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 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 its memory-load instructions.[11]

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 CPU may execute a thread's instructions out of order. The architecture specifies the extent to which this may happen, and it also specifies circumstances under which 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 source code given the same input (in the absence of any abstract executions containing undefined behavior). This requirement is the standard's "as-if" rule. It means: (1) The program's output must be the same as that of the abstract execution; (2) Volatile accesses "are evaluated strictly according to the rules of the abstract machine" (4.1.2p6.1 `[intro.abstract]`); and (3) [A condition on the timing and interleaving of input and output, which does not matter for our purposes]. We will say that the hardware execution *realizes* the abstract execution, or, equivalently, 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

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

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

of the C++ memory model,[12] 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 volatile objects as a form of I/O, and it may not invent, omit, merge, or reorder these accesses, as we will discuss in Sections 4.3 and 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

In addition to a compiler, a complete C++ implementation might include `.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 “compiler” and “implementation” interchangeably.

This section discusses influences and constraints on C++ compilers and then uses this information to define necessary properties of volatile and *quasi-volatile* atomic accesses.

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 their own influence [27, 28], 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 (see Section 2.2.9). Many more examples of user control over language semantics may be found through use of the command `make V=1` in a Linux-kernel source tree [26, Appendix B]. and by the discussion in [27, 28].

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 1, in which T0 loads the value of `x` and stores it in `y` while T1 does the reverse. A sufficiently perverse globally optimizing loose C++ compiler might transform that program to the following before translating it into machine code:

```
1. │ int r1 = 42 │ int r2 = 42
2. │ y = r1      │ x = r2
```

The loads previously on line 1 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, even if relaxed loads are ordered before relaxed stores! (In fact, the transformed program has no loads at all, relaxed or otherwise.)

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.[13] 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 Cause Errors and Destroy Semantic Dependencies

Invented atomic loads are problematic. Consider this code [26]:

```
int r0 = x;
int r1 = r0 * r0 + 2 * r0 + 1;
```

For values of `x` small enough to avoid overflow, the abstract machine is guaranteed to produce a perfect square in `r1`.

But if the compiler is permitted to invent atomic loads then the compiler might transform this code as follows:

```
int r0 = x;
int rinvented = x;
int r1 = r0 * r0 + 2 * rinvented + 1;
```

If the load into `r0` happened when the value of `x` was zero and the load into `rinvented` happened when the value of

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

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

x was 10, then the value of r1 would be 21, which is not a perfect square. This fails to satisfy the abstract machine's guarantee and therefore is an invalid transformation.

Inventing atomic loads can also destroy dependencies. For example, consider this code [26]:

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

It seems 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:

```
 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 = r1a;
10.     z = (r1 == r2);
11. }
```

Here r1a represents the original load from x while the r1b load is invented by the compiler.

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 simply be one, with no dependence on the value loaded from y.

Because invented loads can cause errors and break dependencies, as we have just seen, we should insist that 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 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.

These requirements prohibit invention or duplication of atomic loads and stores. However, they do permit omitting redundant non-volatile atomic stores and fusing of non-volatile atomic accesses to adjacent objects.

5 Hardware Dependencies, Instruction Ordering, and the Fundamental Property

This section examines dependencies at the level of machine instructions and then uses the mapping from the previous section to define the Fundamental Property of semantic dependencies.

5.1 Dependencies at the Hardware Level

Dependencies between machine instructions are determined by the flow of information within a CPU. Each instruction takes a set of inputs and provides a set of outputs, some of which flow to inputs of later instructions. The inputs determine what an instruction will do.

For example, an add instruction might have two inputs (the values to be summed) and two outputs (the sum and some condition-code bits—e.g., Zero, Carry, and Overflow). For another example, a memory-load instruction's input is the address to load from, and its output is the value obtained by the load. For a final example, a memory-store instruction's inputs are the value to store and the address at which to store it; there are no outputs.

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 via intermediate instructions. Tracing back, you can see that any hardware dependency ultimately emanates from initial register values, load instructions, immediate values, or I/O accesses. We are setting I/O aside, which leaves load instructions as the only sources of truly new information in a thread.

5.2 Instruction Ordering

A CPU may start executing an instruction speculatively, but at some point it must either abandon that instruction or *commit* it with a set of specific outputs based on well-defined inputs. A CPU is clearly not permitted to commit an instruction until its inputs have been committed, unless the remaining uncommitted inputs cannot affect that instruction's outputs. For instance, a conditional-move instruction needn't wait for the source input to commit once it knows that the condition is definitely false.

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 the affected instructions to commit in order, even on weakly ordered architectures. We will say that one instruction is *ordered after* another to mean that it must commit after the other one commits.[14]

Dependencies aren't the only ordering mechanism. As mentioned in Section 1.1, a CPU does not make the value of a store instruction available to other CPUs until after the store commits. It then takes additional time for the stored value to travel to other CPUs, owing to the finite speed of light and the non-zero size of processor hardware. Since a load instruction cannot commit until its output (the value read) is fully determined, it must commit after the store it reads from.

Conditional or computed branches also provide ordering. 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; until then the CPU cannot know whether the later instruction should be executed at all. Therefore instructions following a conditional- or computed-branch instruction must commit after the branch, and hence after the source for the branch's condition and/or destination input.[15]

5.3 The Fundamental Property of Semantic Dependencies

We can now formulate the Fundamental Property that we want all semantic dependencies to satisfy in implementations that treat all atomic objects 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.

No implementation in which semantic dependencies satisfy the Fundamental Property can realize a nontrivial OOTA cycle. To see this, suppose that abstract execution A having an OOTA cycle is realized by hardware execution H. 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: (1) W_0' commits after $R_{0,0}'$; (2) $R_{0,0}'$ commits after the store instruction W_k' it reads from; and finally (3) W_k' commits no earlier than W_0'.

[14] Note that this implies nothing about when a load instruction retrieves its value from memory; it may do so long before it commits.

[15] A branch that conditionally jumps to the next instruction would be an exception. We ignore this possibility by treating such branches as no-ops.

If all the stores in the OOTA cycle are omitted then all the reads must also be omitted, meaning that the entire cycle has been optimized out of existence. 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 or not there was an OOTA cycle. We therefore declare OOTA cycles in which all accesses are omitted to be *trivial*.

6 Preservation of Dependencies

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 bypass the difficulty by instead defining when an execution of compiled code *preserves a syntactic dependency.*

In addition, we propose the following Dependency Preservation (DP) thesis:

> If a syntactic dependency is semantic then it is preserved by all compilers for which the definition below applies.

As with the Church-Turing thesis in computability theory, our DP thesis is not susceptible of formal proof because the main concept it deals with (semantic dependency) does not have a formal definition. Nevertheless, we hope that readers will agree our definition of dependency preservation captures the informal notion of semantic dependency sufficiently well to justify the thesis. It will be all we need to rule out the possibility of OOTA.

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 per-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 dependencies at the level of the machine code, where they are much more tractable.

6.1 For Compilers Using Per-Thread Analysis

In this section we consider implementations whose compilers perform only per-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 preservation of dependencies is relative to a particular execution, as described in Section 2.2. A syntactic dependency present in the source code may or may not be preserved in the machine code produced by a compiler, according to the details of the execution in question. For this reason we will characterize preservation of dependencies in a given abstract execution realized by a given hardware execution. (While it is possible to argue about preservation of 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 has a syntactic dependency. We can dispose of one case immediately: If W is omitted in H then the issue of dependency preservation is moot. You can give either answer since it will have no effect. Therefore we'll assume that W is not omitted. Then:

> The dependency from $\{R_i\}$ to W in A is preserved in H if there is another abstract execution B realized by hardware execution G (of machine code produced by the same compiler as H) that together *witness the dependency.*

Informally, a *witness* is another execution in which at least one of the R_i loads obtains a different value but all else is the same as far as possible, and yet the store W acts differently.

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 must either obtain the same value in H and G, or itself be one of the R'_i loads, or 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 dependency preservation 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 whenever a dependency is preserved, it 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 that the dependency of the store W on the loads $\{R_i\}$ in A is preserved in 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 T in 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 preservation of the dependency.

Second case: W' lies in the common period of T in 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 above 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 T in 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 (1) uses per-thread analysis and treats atomic objects as quasi volatile, or (2) uses global analysis and treats atomic objects as volatile, then programs produced by that implementation will never exhibit an OOTA cycle in which the semantic dependencies are preserved. And since by the DP thesis, programs produced by the implementation always preserve semantic dependencies, these programs will never exhibit any OOTA cycles at all. Thus the implementation will automatically comply with full C++, even if it was designed only to comply with loose C++.

6.4 Exercising the Definition

This section exercises the definition of "semantic dependency" on selected examples. These exercises use the following steps:

1. Identify the potential semantic dependencies of interest.
2. Compile the code to assembly language, either using a real compiler or conceptually.
3. Check for an assembly level syntactic dependency between the loads and stores making of the potential semantic dependency.
   a. If there is no assembly level syntactic dependency, then there can be no semantic dependency.
   b. Otherwise, if there is no execution having a witness, then there can be no semantic dependency.
   c. Otherwise, there might be a semantic dependency.

In cases 3a and 3b there is no semantic dependency. In case 3c, there might or might not be a semantic dependency, but time will definitely be consumed executing the resulting code. Either way, the candidate dependency cannot be part of an OOTA cycle.

The following sections apply these steps to two of the examples in this paper.

6.4.1 Exercising Function Calls.

Section 2.2.7 provides the following example:

```
    y = f(x);
```

If the compiler knows nothing about the function `f()` and the variable `x`, it must emit code that loads from `x`, passes this value to `f()`, and finally stores the return value to `y`. Is there a semantic dependency from the load of `x` to the store of `y`? To determine this, it is necessary to apply this same analysis to the definition of `f()`, treating its argument as a load and each of its `return` statements as a store. If and only if this analysis shows that, for a given execution, there is an assembly level syntactic dependency within `f()`, then there must also be a semantic dependency from the load of `x` to the store of `y`. As always, if the compiler cannot prove that there is no such semantic dependency, it must assume than there is one.

If `f` is a pointer to a function, then the compiler must also emit a load from `f`, so that there might also be a semantic dependency from the load from `f` to the store to `y`. In the context of C++, this is straightforward: A given execution will load a pointer to a particular function from `f`, so if there is another execution that loads a pointer to a different function, and the two functions return different values for the value loaded from `x`, then there is a semantic dependency from `f` to `y`. Languages that permit functions to be generated at runtime must use more complicated analysis.

Situations where the compiler knows about `f` often result in inlining, in which case there is no function call, allowing the semantic dependencies (or lack thereof) to be determined with respect to the inlined code.

6.4.2 Exercising Multiplication.

Sections 2.2.1 and 2.2.2 provide the following example:

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

This has potential semantic dependencies between the loads from `x` and `y` to the store to `z`. The corresponding assembly code can vary depending on what the compiler knows about the values of `x` and `y`.

If it knows nothing, then it must load from both, multiply the values loaded, and then store the resulting product to `z`. There is an assembly-level syntactic dependency, and if we start with the execution in which the value 1 is loaded from both variables (resulting in the value 1 being stored to `z`), then the execution in which the value 2 is loaded from `x` witnesses a semantic dependency from `x` to `z`. Similarly, the execution in which the value 2 is loaded from `y` witnesses a semantic dependency from `y` to `z`.

But suppose we instead start with the execution where the value 0 is loaded from `x` and the value 1 is loaded from `y`. In this case, there is no witness execution that varies only the value of `y`, as all such executions will result in the value 0 being stored to `z`. However, the execution where the value -1 is loaded from `x` witnesses a semantic dependency from `x` to `z`, resulting in the value -1 being stored to `z`.

But suppose we instead start with the execution where the value 0 is loaded from both `x` and `y`. There is again no witness execution that varies only the value of `y`, but there is also no witness execution that varies only the value of `x`, all such executions still resulting in the value 0 being stored to `z`. However, the execution where the value 2 is loaded from `x` and the value 3 is loaded from `y` witnesses a semantic dependency from the set {`x`, `y`} to `z`, resulting in the value 6 being stored to `z`.

Alternatively, if the compiler knows that the value loaded from `x` is guaranteed to be zero, then it can omit the loads from both `x` and `y` (assuming that these values are not otherwise used), and then simply store the constant 0 to `z`. There clearly can be no semantic dependencies from these now-nonexistent loads to the store to `z`.

6.4.3 Discussion.

Allowing the compiler to transform the source code to assembly code prior to analysis provides great simplifications. This enables us to show the main result, namely that current compiler-based C++ implementations do not produce OOTA cycles. Note well that these compilers' current analysis suffices.

7 Issues and Refinements

Here we consider some general questions related to semantic dependency and our definition of dependency preservation. Additional information is available in P3064R2 ("How to Avoid OOTA Without Really Trying") [26].

7.1 Defining Semantic Dependency

Some readers may object that our approach does not include any rigorous definition of semantic dependency itself. One could address this drawback by turning our DP thesis around and using it instead as a *definition* of semantic dependency. In other words, a syntactic dependency could be considered to be semantic if it is preserved by 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, see for example Section 7.4. However it may be the best we can do with our current understanding of computing systems.

Either way, regardless of how one wants to define semantic dependency, we have shown that as long as the DP thesis continues to hold, programs produced by an implementation of the right sort will never exihibit OOTA.

7.2 Global Analysis and Volatile vs. Quasi Volatile

One can ask if we really need to require global-analysis compilers to treat atomic objects as volatile; would our results still hold if they merely treated them as quasi volatile? Answer: The proof that preserved dependencies obey the Fundamental Property would go through as before, but the DP thesis would no longer be valid. Here's an example showing how it would fail.

Consider the Simple OOTA program shown in Listing 1. 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", as shown in Section 4.2. This would be a valid transformation, but it would destroy the dependencies in T0 and T1 rather than preserving them, even though those dependencies are considered to be semantic.

This fact is intuitively obvious, but let us describe in detail exactly how the preservation definition would fail. Recall that according to the definition, in order to be preserved in an execution a syntactic 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 the transformed program. In fact, the definition of dependency preservation might remain strong enough to justify the DP thesis if the requirement for global-analysis compilers were weakened, if the compiler were allowed to treat atomic objects as quasi volatile except that it was not allowed to omit accesses to them. This is a possible topic for future research.

7.3 Effect of Memory Layout

Part of our demonstration of the Fundamental Property for preserved 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 preserved 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.

7.4 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.

Although many optimizing compilers would omit the loads into r1 and r3, or merge them with r2 if they are used later, one could imagine a perverse compiler instead using per-thread analysis to 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 that the dependency in the original code is preserved, a suitable witness would have to include a hardware realization of an abstract execution in which r1 and r2 are zero and r3 and r4 are one. But there are no such hardware realizations 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 the dependency is not preserved, which leads one to question whether it really was semantic.

Although this result is unexpected, and while we don't recommend this sort of code transformation, we cannot say that this conclusion is definitely wrong, because one's intuitive notions of semantic dependency are not always clear in cases involving multiple loads from the same variable. In the unlikely event that this transformation proves useful, there will be a community process that determines whether the value of this transformation is worth the added conceptual load on developers. Until that time, pure mathematics cannot help us because it is impossible to say whether or not there is a semantic dependency from y to z without reference to a specific implementation.[16]

7.5 Other Potential Refinements

Additional future work could: (1) Extend these results to some classes of interpreters and just-in-time compilers (JITs) on one hand, and to link-time optimization (LTO) on the other; (2) Relax quasi-volatile semantics to permit limited load speculation (for example, hoisting loads out of the bodies of conditionals); (3) Consider various classes of global analysis, such as demonstrating that a given expression will always evaluate to a constant; (4) Examine the possibility of "flattening" optimizations that combine multiple threads into one; (5) Explore more detailed justifications for the Dependency Preservation thesis; (6) Include the effects of input and output or other operations which can vary from one execution to another; and (7) Delineate more precisely the limits of permissible behavior for quasi-volatile object accesses. This last item may well shed additional light on the dangers of optimizations involving non-volatile atomic operations beyond the cautions outlined in Section 4.3.

7.6 Possible Change to the Standard

As noted in Section 1.1, the C++ standard recommends against OOTA cycles in 33.5.4p8 (`[atomics.order]`) [17]:

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

Sensible though this recommendation might be, it can unnecessarily raise anxiety levels of C++ implementers. After all, what exactly do they need to do in order to follow that recommendation? We therefore advocate adding the following two non-normative notes to this recommendation, but following the notes giving examples of OOTA cycles:

> [Note N — A compiler-based implementation that runs on real-world hardware and treats non-volatile atomic acesses as though they were volatile will not produce out-of-thin-air values for a program free of undefined behavior. An implementation that performs only thread-at-a-time analysis/optimization and that omits and merges non-volatile atomic accesses to the same object and reorders non-volatile atomic accesses to different objects (when permitted by the as-if rule), but that otherwise treats non-volatile atomic accesses as though they were volatile, will not produce out-of-thin-air values for a program free of undefined behavior. Implementations not meeting these restrictions require extra care to ensure that they do not compute out-of-thin-air values. — End note.]
> [Note N+1 — In this context, real-world hardware is subject to physical limitations on instruction-execution and information-transfer latencies, as opposed to abstract platforms with no such limits. Real-world hardware must also carefully manage any speculative execution so as to avoid generating out-of-thin-air values at the hardware level. — End note.]

8 Summary and Conclusion

This paper focused solely on compilers, primarily those that do only per-thread analysis and optimization. It introduced the notion of quasi-volatile object accesses and gave a definition for dependency preservation by a C++ implementation. It formulated a Fundamental Property of semantic dependency along with a Dependency Preservation thesis, and using the thesis proved that implementations subject to some very mild restrictions on how they treat atomic object accesses do have this property. Finally, it demonstrated that satisfying the Fundamental Property guarantees an implementation cannot give rise to OOTA executions.

The conclusion to be drawn from this work is that in this context, avoiding OOTA cycles requires no changes to the standard or to user practices for portable code, and only minimal changes to implementations.

References

[1] 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.

[2] 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.

[3] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In Jan Vitek, editor, *Programming Languages and Systems*, pages 283–307, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

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

[6] 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.

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

[8] Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[16] At the 2025 C++ Standards Committee meeting in Sofia, it was pointed out that the added "`if (r1 != r3)`" can be argued to be repurposing the first and third relaxed atomic load from x. At first glance, this is a very reasonable argument that would unambigously rule out this optimization.

[9] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. Modular data-race-freedom guarantees in the promising semantics. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 867–882, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Peter J. Denning. Abstractions. *Commun. ACM*, 68(3):21–23, February 2025.

[11] 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.

[12] 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.

[13] 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.

[14] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. Linux kernel memory barriers. https://www.kernel.org/doc/Documentation/memory-barriers.txt, March 2006.

[15] Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[16] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *SIGPLAN Not.*, 52(1):175–189, January 2017.

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

[18] 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.

[19] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. Putting weak memory in order via a promising intermediate representation. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[20] 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.

[21] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. *SIGPLAN Not.*, 40(1):378–391, January 2005.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Paul E. McKenney, Alan Stern, Michael Wong, and Maged Michael. How to avoid OOTA without really trying. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3064r2.pdf, July 2024.

[27] 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.

[28] 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.

[29] Evgenii Moiseenko, Matteo Meluzzi, Innokentii Meleshchenko, Ivan Kabashnyi, Anton Podkopaev, and Soham Chakraborty. Relaxed memory concurrency re-executed. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[30] Peizhao Ou and Brian Demsky. Towards understanding the costs of avoiding out-of-thin-air results. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[31] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. *SIGPLAN Not.*, 51(1):622–633, January 2016.

[32] 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.

[33] Jay Richards, Daniel Wright, Simon Cooksey, and Mark Batty. Symbolic mrder. https:www.cs.kent.ac.uk/people/staff/mjb211/smrder.htm, July 2024.

[34] 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.