Outlawing Ghosts: Avoiding Out-of-Thin-Air Results

Hans-J. Boehm
Google, Inc. *
hboehm@google.com

Brian Demsky
UC Irvine
bdemsky@uci.edu

Abstract

It is very difficult to define a programming language memory model for shared variables that both

- allows programmers to take full advantage of weakly-ordered memory operations, but still
- correctly disallows so-called "out-of-thin-air" results, i.e. results that can be justified only via reasoning that is in some sense circular.

Real programming language implementations do not produce out-of-thin-air results. Architectural specifications successfully disallow them. Nonetheless, the difficulty of disallowing them in language specifications causes real, and serious, problems. In the absence of such a specification, essentially all precise reasoning about non-trivial programs becomes impractical. This remains a critical open problem in the specifications of Java, C, and C++, among others.

We argue that there are plausible and relatively straight-forward solutions, but their performance impact requires further study. In the long run, they are likely to require strengthening of some hardware guarantees, so that they translate properly to guarantees at the programming language source level.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent Programming Structures

Keywords relaxed memory models, atomic operations, C++, Java

1. Introduction

A programming language memory model specifies the values read when an object is accessed in a multithreaded program. Most recent programming languages have adopted memory models that at their core guarantee sequential consistency (cf. [11]) for data-race-free programs.[1, 2, 5, 9, 10, 12, 19] Ordinary data variables should not be concurrently accessed by multiple threads, unless all of the accesses are read accesses. If that rule is followed, sequentially consistent (interleaving of thread operations) semantics are guaranteed. There is a way to declare special synchronization variables (e.g. `volatile` variables in Java, `atomic<T>` variables in C++) that do support concurrent accesses, and which can be used, along with locks, to prevent concurrent accesses to data variables.

However, C, C++, and Java have all found it necessary to extend this core guarantee in ways that give the user direct access to weakly ordered memory operations. Motivations for this have varied.

Java was designed to allow the inclusion of untrusted code in trusted applications. Since we have not found a practical way to detect data races in untrusted applications on current hardware, Java must provide reasonable semantics for code that includes data races, so that we can reason about the possible effects of untrusted code. These semantics provide very weak ordering guarantees in order to minimize performance costs.

C[9] and C++[10] are perfectly happy to leave the semantics of programs with data races undefined, but found the cost of enforcing sequential consistency for `atomic` (synchronization) variables high enough to provide programmers an escape to more relaxed memory consistency for such synchronization variables. Programmers may, for example, annotate individual accesses to these synchronization variables with `memory_order_relaxed` to indicate that such accesses do not enforce sequentially consistent memory ordering, and cannot be used to order data accesses. Java is also moving in this direction.

Thus, in C++, if we declare

```
int data;
atomic<bool> sync;
```

then the following two threads running concurrently do not have a data race, and we are guaranteed that `r1 = 17` when thread 2 finishes:

Thread 1	Thread 2

`data = 17;`
`sync = true;`
	`while (!sync) {}`
	`r1 = data;`

Here we have assumed the customary conventions for memory model examples: All variables are initially zero/false, the variables r*i* are local to a thread ("registers"), and all other variables are potentially shared.

However, if we change either or both of the references to `sync` to mention `memory_order_relaxed`, as in, for example

Thread 1	Thread 2

`data = 17;`
`sync.store(true,`
`    memory_order_relaxed);`
	`while (!sync) {}`
	`r1 = data;`

then that is no longer true. The stores to `data` and `sync` can become visible to Thread 2 out of order. Hence we now have a race on `data`, and the program may produce any result whatsoever. If `data` were declared as `atomic<int>` and all accesses to `data` used `memory_order_relaxed` then the program would no longer

* Closely related content previously appeared in an unreviewed C++ standards committee paper [6] produced while the first author was at HP Labs.

MSPC 14, June 13, 2014, Edinburgh, United Kingdom.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2917-0/14/06.
http://dx.doi.org/10.1145/2618128.2618134

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = 42;`

Figure 1. "Real" example: `r1 = r2 = 42` expected

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = r2;`

Figure 2. "Ghostly" example: `r1 = r2 = 42` unexpected

produce undefined behavior, since there are no longer any *data* races. However, `r1` may still be set to zero since, for example, the two stores in thread 1 may still become visible to thread 2 out of order.

For our purposes, though not in general, accesses to ordinary Java variables behave "close enough" to C++ (or C) `memory_order_relaxed` accesses that we will no longer distinguish between the two. From here on, we will generically use ordinary C/Java assignment notation to denote either ordinary Java assignments or C/C++ `memory_order_relaxed` operations.

Unfortunately, it is well-known that such hopefully-well-defined relaxed memory references are exceedingly difficult to specify correctly. The rest of this paper provides some new insights into the character and severity of the problem as well as the space of possible solutions.

Although this topic is one that requires careful formal treatment, our primary goal in this paper is to present an informal intuitive explanation for a problem that most people find highly counter-intuitive.

2. The Out-of-thin-air problem

Consider the example in Figure 1, recalling that all assignments are intended to have well-defined semantics with relaxed memory ordering.

In this example, we expect `r1 = r2 = 42` to be a legal execution as the store to `x` in thread 2 may be reordered by either the compiler or the hardware to before the load into `r2`. Thus the store of 42 may immediately become visible to thread 1, which can then store `42` to `y`, and then finally thread 2 reads `42` from `y`.

In Figure 1, two threads each execute a load followed by a store, and each load sees the other thread's store. If we apply the same reasoning to the similar example in Figure 2, we can get `r1 = r2 = 42` or for that matter, any value whatsoever:

An execution in which both stores store `42` is entirely consistent: both loads can see these stored values, indeed causing both stores to store a value of `42`; the value `42` is essentially generated "out of thin air".

Both Java and C++ memory models are defined by specifying when a particular mapping of loads to corresponding stores ("seen by" the load) is allowed. The natural formulations of the memory model in both cases is to simply allow a load l to see any store s that is not ordered after it by synchronization or program order ("l does not *happen-before* s"), and is not hidden by an intervening store similarly ordered between them. This allows weakly ordered operations to either observe racing operations or not. This naturally allows both the expected result in Figure 1, and the out-of-thin-air behavior in Figure 2, since the loads in both examples race with the store in the other thread, and are thus allowed to see it.

One can conceive of wildly speculative (and correspondingly dubious) execution mechanisms that in fact generate such out-of-thin-air results: The language implementation might remember that the last time this code was executed, `x` and `y` both had a value of `42`. It might speculatively assume that they will again this time, tentatively assigning 42 to both `r1` and `r2`. The stores would go ahead and speculatively store the values of `r1` and `r2` (both `42`) back to, say a shared cache. It would then confirm that the initial guesses for the loads were correct, which they now are, and thus conclude that it was safe to commit the speculative stores.

Such cross-thread speculation in fact does not occur in practice. As a result, out-of-thin-air results do not occur in practice. The problem is that it is exceedingly difficult to specify semantics that preclude them, while not precluding the analogous result in the "real" example in Figure 1. In fact, it is extremely difficult to precisely define an "out-of-thin-air" result; if we could define what it is, we could explicitly disallow it.

Both Java and C++ (and C) have tried and failed to effectively prohibit out-of-thin-air results in their specifications. Java [12] introduced complicated causality rules, which turned out not to have the intended effect [21]. The C++11 standard includes rules intended to prohibit certain kinds of out-of-thin-air values, and informally discourages the rest. Those initial rules were again subsequently found not to specify what was intended, and were subsequently replaced, in the current C++14 draft, by vague encouragement to do the right thing. [4] We consider this unsatisfactory, since the current specification is too imprecise to support formal reasoning.

In the next sections we analyze these difficulties, and propose a solution, though at some performance cost.

3. Allowing out-of-thin-air results is disastrous

During the Java memory model effort that resulted in [12], it was widely believed that the primary problems with allowing out-of-thin-air values were:

1. If the semantics allowed out-of-thin-air results, there is no way to prove that untrusted, potentially malicious, code will not be able to generate out-of-thin-air values that compromise security, e.g. secret passwords. The semantics would allow untrusted code to generate an out-of-thin-air value that happens to be the secret password. Such attacks are not entirely implausible: Implementation techniques that could result in out-of-thin-air values generally rely on speculating/guessing values, typically values that have arisen before when the code was last executed. By invoking a function that had previously been used for password computation in such a context, an attacker might be able to coax the implementation into generating that password again. Certainly there would be no way to prove that this couldn't happen, and hence it would be difficult to prove properties of applications including untrusted code.

2. Certain kinds of out-of-thin-air computations can induce race-free computations to produce non-sequentially consistent results. The canonical example (repeatedly pointed out by Sarita Adve) is

Thread 1	Thread 2
`if (x) y = 1;`	`if (y) x = 1;`

   If each condition sees the assignment from the other thread, then (with `x` and `y` initially zero, as usual), this can result in both variables set to one, in spite of the fact that there is no sequentially consistent execution in which either assignment is executed, and hence, if we interpret `x` and `y` as ordinary data variables, there is no data race.

3. There was concern that out-of-thin-air pointers might violate type safety by pointing to objects of the wrong type.

```
struct foo {
  atomic<struct foo *> next;
};
struct foo *a, *b;
// a & b initially reference
// disjoint data structures
```

Thread 1	Thread 2
`r1 = a->next;`	`r2 = b->next;`
`r1->next = a;`	`r2->next = b;`

Figure 3. Ghostly linking of data structures

```
int a[2];
int a_is_big = 0;
int ok_to_write_a2 = 0;
```

Thread 1	Thread 2
`if (a_is_big)`	`if (ok_to_write_a2)`
`  ok_to_write_a2 = 1;`	`  a[2] = 17;`

Figure 4. Ghostly results from control dependence

It seems at least plausible that the last two could be explicitly prohibited without addressing the general problem. And there was some debate about the importance of being able to run untrusted code as part of a Java application.

Perhaps more importantly, none of these are completely convincing in the context of C++ memory_order_relaxed atomic operations: It is never safe to allow untrusted code into a C++ application, for many reasons. Memory_order_relaxed operations are unlikely to be used in such non-racing contexts. Type-safety guarantees in C++ are already weak.

Thus the original C++11 specification took a somewhat cavalier attitude towards this problem, knowingly including only a partial solution.[1]

Here we argue that the situation is in fact *much* worse: If we allow out-of-thin-air results, we break the most basic forms of reasoning about very ordinary code, be it C++ code using memory_order_relaxed or Java code.

We show this by assuming that out-of-thin-air results are allowed and looking at the surprising consequences for even tiny code examples.

Consider the example in Figure 3 in which a and b initially reference two disjoint lists. The two threads in this example then create a reference from the second objects in their respective disjoint lists to the first objects.

Surprisingly, out-of-thin-air behavior allows an execution of this example in which r1=b, r2=a. Assuming r1=b, the store r1->next = a in Thread 1 then justifies the load r2 = b->next in Thread 2 assigning r2=a. The same reasoning can be applied to r2=a to justify r1=b. The end result is that two independent threads spontaneously link disjoint data structures. Clearly, allowing such behavior is unacceptable as it makes it impossible to reason about the behavior of almost any code.

Like the first example in this section, this one produces a non-sequentially-consistent result for what is really a data-race-free program. But it even more profoundly violates our intuitions about parallel program behavior.

[1] As previously mentioned, this solution was also later discovered to be incorrect.[6]

Out-of-thin-air behavior involving control dependences can also lead to similarly disastrous behavior. Consider the example in Figure 4. If thread 2 speculates ok_to_write_a2, and writes a[2] (out of bounds, updating a_is_big), then thread 1 speculatively writes ok_to_write_a2 after seeing the speculative write to a_is_big, then thread 2's speculative hypothesis is satisfied.

This of course involves C++ undefined behavior, but this is likely to be the problem in practice; speculation can lead to undefined behavior, which can then satisfy the speculation.

In practice, we expect such issues to arise where a programmer calls a library function f() that expects a certain precondition to hold. The library implementer does not specify the behavior of f() when the precondition fails to hold. In order to prove this program correct without absence-of-out-of-thin-air guarantees, the programmer has to ensure that such a bad call to f() cannot result in bad behavior that sets a variable seen by another thread, and then indirectly justifies the bad call to f(). Without understanding the implementation of f(), we have no way to reason about whether or not this may yield such a self-satisfying misspeculation.

For a developer to reason about the correctness of code, the developer must postulate all of the different ways code might interact to generate out-of-thin-air behaviors. This is complicated by the fact that these interactions may not be localized and could easily cross API boundaries. In fact even supposedly dead code can potentially generate out-of-thin-air behaviors.

Many of the concurrent algorithms that stand to benefit from relaxed memory operations have complex enough bugs that they stand to benefit greatly from formal methods. Out-of-thin-air behaviors have been recognized by many experts[3, 14, 20] to make verifying code infeasible.

4. Out-of-thin-air results at hardware level

The fundamental difference between the "real" and "ghostly" examples above (Figures 1 and 2) is that the latter involves dependences between each load and the subsequent store. Informally, the out-of-thin-air results form a cycle of dependences: The thread 1 store depends on the thread 1 load, which depends on the thread 2 store, which depends on the thread 2 load, which depends on the thread 1 store.

If we can define a precise notion of "dependence", we can precisely define "out-of-thin-air" results, and simply state that they are disallowed. The difficulty is defining the notion of "dependence".

At the hardware level, it is entirely feasible, and common, to do so. Weakly-ordered architectures, including ARM, Power, and Itanium, specify when one instruction in an execution trace is dependent on another. This notion of dependence is defined essentially in terms of which instruction mentions the result produced by another. For example, these architectures view the xor instruction, and hence the subsequent store instruction, in

```
load r1 = [a]
xor r2 = r1 ^ r1
store [b] = r2
```

as depending on the prior load, in spite of the fact that r2 is always set to zero, independent of the contents or r1. Similarly, they generally consider all instructions as depending on any prior conditional branch, and transitively on any preceding instructions producing results that fed into the branch condition.[2]

In doing so, hardware, in some sense, uses a conservative approximation to "actual dependences". The hardware notion is easy to define, but not all "hardware dependences" reflect cases in which the final outcome can actually be affected.

[2] Itanium is the exception here, in that the specification appears to use a more restrictive rule for branch dependences.

Thread 1	Thread 2

`r3 = x;`
`if (r3 != 42)`
`  x = 42;`
`r1 = x;`
`y = r1;`
	`r2 = y;`
	`x = r2;`

Figure 5. Ghosts are real!

Unfortunately, as we will see in the next section, this hardware approximation to "dependences" does not make sense at higher levels of the software stack. The core problem is that we have not been able to find one that does.

With such a hardware definition of dependence, the surprising results in Figure 2 are easily disallowed. The architecture specification disallows stores from being made visible before the loads on which they depend. This effectively disallows cyclic dependences as we saw in Figure 2.

5. Out-of-thin-air results at language level

Standard compiler optimizations do not introduce the most problematic forms of out-of-thin-air behavior in programs. As precisely defining what constitutes out-of-thin-air behavior is the core problem, it is difficult to make a blanket statement that there are no issues.

Indeed, compilers can introduce behaviors that yield executions that arguably contain causal cycles. Figure 5, a variation on the "ghostly" example, is taken from the JMM causality tests[15]. Compiler optimizations can actually transform this example to allow executions where `r1 = r2 = r3 = 42`. (The value of `r3` is the most surprising one.) A compiler could determine that it must be legal for `r1 = x;` to see the value 42. (In the absence of interference from another thread `x` is always 42 at this point, and it is certainly allowable for no other thread to be scheduled while the first three lines of Thread 1 are executing.) It could then transform `y = r1;` to `y = 42;` and perform it earlier. At this point, the transformed program allows the behavior in question.

Compiler optimizations on vanilla sequential code routinely break dependences. And the ability to do so is often important.

Consider the following code excerpt:

```
if (x) {
  a[i++] = 1;
} else {
  a[i++] = 2;
}
```

Does the store to `i` depend on the load of `x`? Requiring compilers to preserve such dependences in vanilla code would prevent compilers from hoisting the expression `i++` out of the conditional, since that would remove the dependence on `x`. There are numerous other examples of this. Optimizing a multiplication by zero potentially eliminates a dependence.

In general dependences between two shared variable operations may pass through arbitrary other functions. Even in C++, where `memory_order_relaxed` operations are easily identifiable, we can write:

```
y.store(f(x.load(memory_order_relaxed)),
          memory_order_relaxed);
```

If the language specification required dependence preservation, this would require `f()` to preserve dependences, even if `f()` involved no atomic accesses and was compiled separately. Even functions that appear entirely single-threaded, but might be called from a multithreaded program using `memory_order_relaxed` would be affected. In C++, this seems entirely indefensible. Even in Java, such constraints are extremely undesirable. In addition to the negative performance consequences, all the world's compilers would have to be substantially rewritten.

Since we can't easily preserve a naïve notion of dependence, can we define a more sophisticated one that respects compiler optimizations? So far, such definitions have been elusive, nor is it really clear what that should mean. Consider

```
y = n * x;
```

Is the store of `y` dependent on the load of `x`? If $n \neq 0$, presumably yes. But what if the expression occurs inside a function with parameter `n` and the function was invoked with an argument of zero for `n`, and then inlined by the compiler? In that case again there are execution paths on which the load of `x` and store of `y` are not ordered.

Again, such transformations don't actually introduce anything that looks like an out-of-thin-air result. But this example demonstrates the difficulty of defining a reasonable notion of dependence.

The C++ and Java memory models essentially define when a particular "candidate" execution is valid, i.e. reflects correct cross-thread visibility of updates. But there appears to be no reasonable way to define a notion of dependence on such candidate executions.[3]

6. A solution: preserve load-to-store ordering

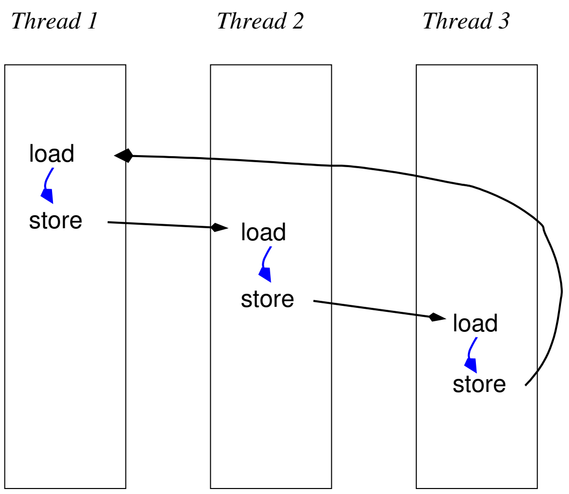

Figure 6. cyclic dependence

Our goal is to prevent "dependence" cycles that involve multiple threads. Such cycles must be composed of intra-thread segments, each starting with a load and ending with a store dependent on that load. This is illustrated in Figure 6, where the blue edges are intra-thread dependences and the black edges represent the fact that a store is seen by a load in another thread, i.e. the cross-thread reads-from relation.

Very informally speaking, we can always prevent such cycles if we make sure that the result of a store instruction seen by another thread never becomes visible to that thread before a load by the same thread on which it depends. One way to ensure this is to have both the compiler and hardware ensure that no store ever

[3] Peter Sewell and Mark Batty have an unpublished, but much more precise, version of such a result.

becomes visible to other threads before any prior load by that thread, dependence or not.

Prior work [7, 22] has observed that this can be achieved by requiring the hardware to enforce a TSO [17, 18] memory model, and having the compiler preserve it. However TSO is impractically expensive to enforce on common weakly ordered architectures like ARMv7 and Power.[4]

But TSO is a much stronger memory model than we need for present purposes. We simply need to prohibit all reordering of load instructions with respect to later store instructions executed by the same thread. This property is enforced by TSO, but it can also be enforced on more weakly ordered architectures, at much less cost than TSO enforcement.

This property is easy to state as an addition to the existing C++ memory model (as a replacement for the current vague "no out-of-thin-air" wording) or within the framework of the current Java memory model (as a replacement for the current discussion of "Causality Requirements" in [8]). We simply require that the existing happens-before relation is consistent with the "reads-from" relation, relating stores to the loads that retrieve the stored value. More formally:

$$(\textit{happens-before} \cup \textit{reads-from})^*$$

is irreflexive. This is a simple constraint on a single execution that fits well with the existing model.

However, it is still expensive enough to implement on some architectures to remain highly controversial. It prohibits the undesirable results from Figures 2, 3 and 4. But it also disallows the $\texttt{r1} = \texttt{r2} = 42$ outcome in Figure 1, which we previously argued should be expected, is commonly allowed by existing implementations, and is generally accepted as benign.

6.1 Implementation cost

The implementation cost requires much more careful empirical study before we could adopt such a solution for Java. Here we briefly outline our preliminary observations.

In general, the costs fall into two categories: prohibited compiler optimizations and the cost of additional instructions required to enforce ordering at the hardware level.

Prior research has shown that in many cases, even the cost of having the compiler preserve sequential consistency (i.e., ensure sequential consistency on sequentially consistent hardware) is modest. [13] Here we impose *much* weaker constraints. In particular, it remains acceptable to advance loads and to delay stores. Moving loop invariant loads out of loops is legal where it previously was. Heroic loop-nest optimizations for cache locality are a problem, but we do not believe that these are commonly performed for either Java loops, or for C++ loops containing `memory_order_relaxed` accesses.

For Java, the precise compiler costs warrant further study, but our expectation is that the added restriction adds only modest overhead. For C++, it is likely to be minuscule, since only `memory_order_relaxed` operations are affected. These are currently extremely rare, since compiler support for them is new. We expect them to remain reasonably rare.

We briefly outline the expected cost of adding hardware ordering instructions on some common architectures:

[4] Among other issues, TSO requires that stores be made visible to all other threads in the system at the same time. Enforcing this on ARM or Power requires a heavy-weight fence (`dmb` or `sync`) between every pair of shared memory accesses.[16] Such fences typically incur costs on the order of dozens of cycles, so we expect this to be impractical in the absence of heroic whole-program analysis (which, in turn, has also proven impractical in most contexts).

6.1.1 X86

X86 and other TSO memory models already forbid load-to-store reordering for normal loads and stores. Thus no additional instructions are needed and no additional cost is incurred.

6.1.2 Power

The Power architectural specification allows loads to be reordered with subsequent stores. There is some evidence that current implementations do not take advantage of this [16].

If necessary, spec-compliant ordering can be enforced with a relatively light-weight fence between every shared and potentially racing load and the following such store, or probably preferably, by inserting a bogus conditional branch (e.g. to the next instruction), which "depends" (by the conservative hardware definition) on each load. The architecture guarantees that stores are not speculatively executed, so the branch is sufficient to preserve ordering.

The compiler has several opportunities to further reduce the cost of these branches. They can be delayed until the next store to a potentially shared location, or omitted if other constructs along the path (e.g. a branch on the load appearing in the code, or a fence inserted for other reasons) already enforce the ordering.

Thus the cost to enforce ordering either involves a conditional branch for a significant fraction of potentially shared loads, or possibly a change to the hardware specification, restricting future implementations.

6.1.3 ARMv7

The situation is similar to Power, except that current implementations are known to perform load-to-store reordering, and thus it is not an option to simply change the specification. A fence or conditional branch is required after shared loads. However, a fairly widespread processor erratum already appears to require this fence for C++.[5] Thus the cost of enforcing this ordering for C++ may actually be similar to the x86 case. However the cost of enforcing this ordering for Java, where the erratum is irrelevant, remains a major concern.

6.1.4 ARMv8

ARMv8 provides yet another option to enforce ordering: One could use the newly provided acquire and release load and store operations, most likely by replacing all affected stores by release stores. On the other hand, we expect that added overhead is required even for C++; there is presumably not a similar processor erratum to mitigate the restriction.

6.2 A variation on preserving load-to-store ordering

Preserving load-to-store ordering implicitly preserves all dependence ordering between such pairs of atomic operations. Alternatively, this requirement could be weakened to require that syntactic dependences be preserved, as for hardware, but to enforce it in many cases by again fully preserving load-to-store ordering. This is somewhat more complex to specify as it requires that the language memory model define a syntactic notion of dependence. However, this more complex approach has the potential to provide compiler writers with some more room to implement optimizations.

The straightforward implementation strategy would then be to simply preserve load-to-store ordering as above. The dependence based specification approach opens up an alternative implementation strategy of preserving dependences in code that contains atomics and requiring that load-store fences be inserted at any point that carries a dependence on a relaxed load to code outside of the com-

[5] See `http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf` for details.

pilation unit. In some cases, it may be possible to avoid the fence altogether if existing operations already provide the necessary fence or if no dependence on a relaxed load can leave the compilation unit.

For small compilation units, the dependence preservation implementation approach is unlikely to yield improvements and thus the preferred implementation strategy is likely to be preserving load-to-store dependences as above. For larger compilation units, the freedom to avoid stronger atomics or fences within the compilation unit by preserving dependences may enable better code generation. Further investigation is necessary to see whether the potential performance benefits from this approach merit the additional specification and compiler complexity. Either variation of these approaches would benefit from better future hardware support for constraining load-to-store reordering.

7. Conclusions

Our inability to specify the absence of out-of-thin-air results for concurrent programming languages is a serious obstacle to reasoning about very conventional multi-threaded programs. After a decade of trying to address this problem in the context of current hardware, we believe it is time to consider solutions that incur modest performance costs on current hardware, and are likely to require long-term hardware changes. It appears more and more that a core contributor to the problem are hardware specifications, such as those involving dependences, that we do not know how to translate to the programming language level.

Acknowledgments

This paper relies significantly on insights from discussions involving Mark Batty, Peter Sewell, Richard Grisenthwaite, Paul McKenney, Jeremy Manson and others, who are unlikely to fully agree with all of our conclusions. This project was partly supported by a Google Research Award and by the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786.

References

[1] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8): 90–101, August 2010.

[2] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, 1990.

[3] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[4] H.-J. Boehm. N3786: Prohibiting "out of thin air" results in C++14. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm`, September 2013.

[5] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 68–78, 2008.

[6] H.-J. Boehm et al. N3710: Specifying the absence of "out of thin air" results (LWG2265). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html`, August 2013.

[7] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A buffered memory model for Java. In *POPL*, pages 329–342, 2013.

[8] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java language specification: Java SE 8 edition. http://docs.oracle.com/javase/specs/jls/se8/html/index.html, 2014.

[9] ISO JTC1/SC22/WG14. ISO/IEC 9899:2011, information technology — programming languages — C. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853` or an approximation at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf`, 2011.

[10] ISO JTC1/SC22/WG21. ISO/IEC 14882:2011, information technology — programming languages — C++. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372` or a close approximation at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf`, 2011.

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[12] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proceedings of the Symposium on Principles of Programming Languages*, 2005.

[13] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an SC-preserving compiler. In *PLDI*, pages 199–210, 2011.

[14] B. Norris and B. Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *"Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications"*, 2013.

[15] B. Pugh et al. JMM causality test cases. `http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html`, retrieved March, 2014.

[16] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[17] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocesors. *Communications of the ACM*, 53(7):89–97, July 2010.

[18] I. SPARC International. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc, 1994.

[19] *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001*. United States Department of Defense, 1983. Springer.

[20] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

[21] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 27–51, 2008.

[22] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *JACM*, 60, 2013.