Foundations of the C++ Concurrency Memory Model

Hans-J. Boehm
HP Laboratories
Hans.Boehm@hp.com

Sarita V. Adve
University of Illinois at Urbana-Champaign
sadve@cs.uiuc.edu

Abstract

Currently multi-threaded C or C++ programs combine a single-threaded programming language with a separate threads library. This is not entirely sound [7].

We describe an effort, currently nearing completion, to address these issues by explicitly providing semantics for threads in the next revision of the C++ standard. Our approach is similar to that recently followed by Java [25], in that, at least for a well-defined and interesting subset of the language, we give sequentially consistent semantics to programs that do not contain data races. Nonetheless, a number of our decisions are often surprising even to those familiar with the Java effort:

- We (mostly) insist on sequential consistency for race-free programs, in spite of implementation issues that came to light after the Java work.
- We give *no* semantics to programs with data races. There are no benign C++ data races.
- We use weaker semantics for `trylock` than existing languages or libraries, allowing us to promise sequential consistency with an intuitive race definition, even for programs with `trylock`.

This paper describes the simple model we would like to be able to provide for C++ threads programmers, and explain how this, together with some practical, but often under-appreciated implementation constraints, drives us towards the above decisions.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent Programming Structures; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; B.3.2 [*Memory Structures*]: Design Styles—Shared Memory

General Terms languages, standardization, reliability

Keywords Memory consistency, memory model, sequential consistency, C++, trylock, data race

1. Introduction

As technological constraints increasingly limit the performance of individual processor cores, the computer industry is relying increasingly on larger core counts to provide future performance improvements. In many domains, it is expected that any substantial future performance gains will require explicitly parallel applications.

The most established way to write parallel applications in standard programming languages is as a collection of threads sharing an address space. In fact, a large fraction of existing desktop applications already use threads, though typically more to handle multiple event streams, than parallel processors. The majority of such applications are written in either C or C++, using threads provided by the operating system. We will use Pthreads [21] as the canonical example. The situation on Microsoft platforms is similar.

Both the C and C++ languages are currently defined as single-threaded languages, without reference to threads. Correspondingly, compilers themselves are largely thread-unaware, and generate code as for a single-threaded application. In the absence of further restrictions, this allows compilers to perform transformations, such as reordering two assignments to independent variables, that do not preserve the meaning of multithreaded programs [29, 2, 25].

The OS thread libraries attempt to informally address this problem by prohibiting concurrent accesses to normal variables or *data races*. To prevent such concurrent accesses, thread libraries provide a collection of synchronization primitives such as `pthread_mutex_lock()`, which can be used to restrict shared variable access to one thread at a time. Implementations are then prohibited from reordering synchronization operations with respect to ordinary memory operations in a given thread. This is enforced in a compiler by treating synchronization operations as opaque and potentially modifying any shared location. It is typically assumed that ordinary memory operations between synchronization operations may be freely reordered, subject only to single-thread constraints.

Unfortunately, informally describing multithreaded semantics as part of a threads library in the above way is insufficient for several reasons, as described in detail by Boehm [7]. Briefly, the key reasons are as follows.

- The informal specifications provided by current libraries are ambiguous at best; e.g., what is a data race and what precisely are the semantics of a program without a data race? Figure 1

```
   Initially X=Y=0
T1                T2
r1=X              r2=Y
if (r1==1)        if (r2==1)
    Y=1               X=1

Is outcome r1=r2=1 allowed?
```

Figure 1. Without a precise definition, it is unclear if this example is data-race-free. The outcome shown can occur in an execution where threads T1 and T2 speculate that the values of `X` and `Y` respectively are 1, then each thread writes 1, validating the other's speculation. Such an execution has a data race on `X` and `Y`, but most programmers would not envisage such executions when assessing whether the program is data-race-free.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

```
struct s { char a; char b; } x;

Thread 1:          Thread 2:
  x.a = 1;           x.b = 1;

Thread 1 is not equivalent to:
  struct s tmp = x;
  tmp.a = 1;
  x = tmp;
```

Figure 2. Compiler transformations must be aware of threads. Many compilers perform transformations similar to the one above when `a` is declared as a bit-field. The transformation may be visible to client code, since the update to `b` by thread 2 may be overwritten by the store to the complete structure `x`.

shows an example [1] where the answers to these questions are unclear from the current specification.

- Without precise semantics, it is hard to reason when a compiler transformation will violate these semantics. For example, Boehm [7] shows that reasonable (common) interpretations of the current specification do not preclude a class of compiler transformations that effectively introduce new writes to potentially shared variables. Examples include Figure 2 and speculative register promotion. As a result, conventional compilers can, and do, on rare occasions, introduce data races, producing completely unexpected (and unintended) results without violating the letter of the specification.
- There are many situations in which the overhead of preventing concurrent access to shared data is simply too high. Either vendors, or sometimes applications, generally provide facilities for atomic (indivisible, safe for unprotected concurrent use) data access, without the use of locks. For example, gcc provides a family of `__sync` intrinsics, Microsoft provides `Interlocked` operations, and the Linux kernel defines its own atomic operations (which have been regularly (ab)used in user-level code). These solutions have not been satisfactory, both because they are not portable and because their interactions with other shared memory variables are generally not carefully or clearly specified.

In order to address the above issues, an effort was begun several years ago to properly define the semantics of multi-threaded C++ programs, specifically, the memory model, in the next revision of the C++ language standard. Although this standard is not expected to be completely approved until 2009 or 2010, the core changes to support threads and the memory model described here were recently voted into the C++ working paper, and are now also under discussion in the C committee. This effort has been proceeding in parallel, and cooperatively with, a similar effort for Microsoft's native compilation platforms [31].

The key result of the above effort is a memory model for multithreaded C++ programs that supports a simple programming model. This paper describes that model and several new fundamental issues that had to be resolved along the way.

1.1 State-of-the-art for Memory Models

The memory model, or memory consistency model, specifies the values that a shared variable read in a multithreaded program is allowed to return. The memory model clearly affects programmability. It also affects performance and portability by constraining the transformations that any part of the system may perform. In practice, any part of the system (hardware or software) that transforms the program must specify a memory model, and the models at the different system levels must be compatible. For example, the C++ model constrains the transformations allowed by a C++ compiler. The memory model for the hardware that will run the produced binary must not allow results that would be illegal for the C++ model applied to the original program.

There has been extensive work in memory models over the last two decades. Sequential consistency, defined by Lamport [24], is considered to be the most intuitive model. It ensures that memory operations appear to occur in a single total order (i.e., atomically); further, within this total order, the memory operations of a given thread appear in the program order for that thread.

Unfortunately, sequential consistency restricts many common compiler and hardware optimizations [2]. There has been significant academic progress in compiler algorithms to determine when a transformation is safe for sequential consistency [29, 30, 23] and in hardware that speculatively performs traditional optimizations without violating sequential consistency [19, 28, 33, 15]. However, current commercial compilers and most current commercial hardware do not preserve sequential consistency.

To overcome the performance limitations of sequential consistency, hardware vendors and researchers have proposed several relaxed memory models [2]. These models allow various hardware optimizations, but most are specified at a low level and are generally difficult to reason with for high level language programmers. They also limit certain compiler optimizations [25].

An alternative approach, called the data-race-free [3, 1] or properly labeled [18, 17] models, has been proposed to achieve both the simple programmability of sequential consistency and the implementation flexibility of the relaxed models. This approach is based on the observation that good programming practice dictates programs be correctly synchronized or data-race-free. These models formalize correct programs as those that do not contain data races in any sequentially consistent execution. They guarantee sequential consistency to such data-race-free programs, and do not provide any guarantees whatsoever to others. Thus, the approach combines a simple programming model (sequential consistency) and high performance (by guaranteeing sequential consistency to only well-written programs). Different data-race-free models successively refine the notion of a race to provide increasing flexibility, but requiring increasing amounts of information from the programmer. The data-race-free-0 model uses the simplest definition of a data race; i.e., two concurrent conflicting accesses (formalized later).

Among high-level programming languages, Ada 83 [32] was perhaps the first to adopt the approach of requiring synchronized accesses and leaving semantics undefined otherwise. As mentioned above, Pthreads follows a similar approach, but neither Ada nor Pthreads formalized it sufficiently. Recently, the Java memory model underwent a major revision [25]. For programmers, for all practical purposes, the new Java model is data-race-free-0. However, the safety guarantees of Java preclude leaving the semantics with data races as undefined. Much of the Java effort therefore focused on defining these semantics in a way that preserves maximum implementation flexibility without violating the safety and security guarantees of Java. Nevertheless, the Java model does preclude some compiler optimizations and the full model is quite complex [25]. Since C++ is not type safe, neither the restrictions nor the complexity of the full Java model appear justified for C++.

1.2 The C++ Model and Contributions of this Paper

The model chosen for C++ is an adaptation of data-race-free-0; i.e., it guarantees sequential consistency for programs without data races and has undefined semantics in the presence of a data race. Given the recent work on the Java model described above, the data-race-free-0 model may seem an obvious choice. However, when this process began, there were several factors that prevented using data-race-free-0, some of which were exposed after conclusion of

the Java work and are also relevant to that work. This paper provides an understanding of those factors, shows how they were resolved to make the data-race-free-0 model acceptable to C++ developers and most hardware vendors, and provides the first published description of the C++ model in the full context of prior work.[1] Specifically, there are three main issues we discuss in this paper:

(1) Sequentially consistent atomics: The data-race-free models require that all operations that are not ordinary data operations (e.g., synchronization operations, atomics in C++, volatiles in Java[2]) appear sequentially consistent. The recent advent of multicore systems exposed important hardware optimizations that appear to conflict with this requirement, initially resulting in most hardware vendors and many software developers opposing it. We describe the conflict and show that unfettered exploitation of such hardware optimizations leads to models that are too hard to formalize and/or use. These arguments were largely responsible in convincing hardware vendors (e.g., AMD and Intel) to develop specifications that are now consistent with the use of sequentially consistent atomics.

(2) Trylock and impact on the definition of a data race: Synchronization primitives such as trylock can be used in non-intuitive ways that previously would have required more complex definition of data races and/or overly strict fences. We show a simple way to get around this problem.

(3) Semantics for data races: We do not provide any semantics for programs with data races. This was not uncontroversial; we explain the arguments for this position in Section 5.

We believe that sequentially consistent atomics provide a programming model that is both easier to use and describe than the alternatives, and is the model we should strive for. However, it became clear during the standardization process that exclusive support for sequentially consistent atomics was not viable, largely for two reasons:

- It is sufficiently expensive to implement on some existing processors that an "experts-only" alternative for performance-critical code was felt to be necessary. It is unclear to us whether this need will persist indefinitely.
- Existing code is often written in a way that assumes weak memory ordering semantics and relies on the programmer to explicitly provide the required platform-dependent hardware instructions to enforce the necessary ordering. The Linux kernel is a good example of this. It is often easier to migrate such code if we provide primitives closer to the semantics assumed (currently sometimes incorrectly) by such code.

As a result, the C++ working paper provides for both sequentially consistent atomics, and a mechanism to weaken memory ordering with explicit specifications. We refer to the latter as *low-level atomics*. This unfortunately requires a significantly more complicated memory model.

Here we first present the simpler model, as we would expect most programmers to use it. This is sufficient to understand the core points of this paper. We then return to a presentation closer to the standards working paper that does support the addition of low-level atomics, and outline the proofs that the two specifications are equivalent.

[1] Much of the work here, aside from the discussion of architectural issues, is described less formally in earlier unrefereed standards committee papers, notably [9, 10, 11, 13, 27] and their predecessors.

[2] See [6] for the reasons they are not called `volatile` in C++.

2. The C++ Model Without Low-Level Atomics

We try to specify memory models in this paper sufficiently precisely that if we had a completely formal sequential semantics, this could be translated into a completely formal description of the multi-threaded semantics. This section assumes only a single flavor of atomics; i.e., the default, sequentially consistent atomics in the standard.

Memory operations are viewed as operating on abstract *memory locations*. Each scalar value occupies a separate memory location, except that contiguous sequences of bit-fields inside the same innermost `struct` or `class` declaration are viewed as a single location.[3] Thus the fields in the `struct s` of Figure 2 are separately updateable locations unless we change both field declarations to bit-fields.

The remainder of the C++ standard was modified to define a *sequenced-before* relation on memory operations performed by a single thread [14]. This is analogous to the program order relation in Java and other work on memory models. Unlike prior work, this is only a partial order per thread, reflecting undefined argument evaluation order.

Define a *memory action* to consist of:

1. The type of action; i.e., lock, unlock, atomic load, atomic store, atomic read-modify-write, load, or store. All but the last two are customarily referred to as *synchronization operations*, since they are used to communicate between threads. The last two are referred to as *data operations.*
2. A label identifying the corresponding program point.
3. The values read and written.

Bit-field updates can be modeled as a load of the sequence of contiguous bit-fields, followed by a store to the entire sequence.

Define a thread execution to be a set of memory actions, together with a partial order corresponding to the sequenced-before ordering.

Define a *sequentially consistent execution* of a program to be a set of thread executions, together with a total order $<_T$ on all the memory actions, which satisfies the constraints:

1. Each thread execution is *internally consistent*, in that it corresponds to a correct sequential execution of that thread, given the values read from memory, and respects the ordering of operations implied by the sequenced-before relation.
2. T is consistent with the sequenced-before orders; i.e., if a is sequenced before b then $a <_T b$.
3. Each load, lock, and read-modify-write operation reads the value from the last preceding write to the same location according to $<_T$. The last operation on a given lock preceding an unlock must be a lock operation performed by the same thread.

Effectively this requires that $<_T$ is just an interleaving of the individual thread actions.

Two memory operations *conflict* if they access the same memory location, and at least one of them is a store, atomic store, or atomic read-modify-write operation. In a sequentially consistent execution, two memory operations from different threads form a *type 1 data race* if they conflict, at least one of them is a data operation, and they are adjacent in $<_T$ (i.e., they may be executed concurrently).

We can now specify the C++ memory model simply as:

- If a program (on a given input) has a sequentially consistent execution with a (type 1) data race, then its behavior is undefined.

[3] Zero-length bit-fields can be used to break such sequences. This extends the existing convention for the use of zero-length bit-fields.

```
  T1                        T2

x = 42;           while (trylock(l) == success)
lock(l);                unlock(l);
                  assert(x == 42);
```

Figure 3. Undesirable use of trylock.

- Otherwise, the program (on the same input) behaves according to one if its sequentially consistent executions.

2.1 Optimizations Allowed by the Model

Previous work [3, 18, 1, 12] shows that with the above model, hardware and compilers may freely reorder memory operation $M1$ sequenced before memory operation $M2$ if the reordering is allowed by intra-thread semantics and: (1) $M1$ is a data operation and $M2$ is a read synchronization operation, or (2) $M1$ is write synchronization and $M2$ is data, or (3) $M1$ and $M2$ are both data with no synchronization sequence-ordered between them.

Additionally, when locks and unlocks are used in "well-structured" ways, the following reorderings between $M1$ sequenced-before $M2$ are safe (assuming they are allowed by intra-thread semantics) [1]: $M1$ is data and $M2$ is the write of a lock operation; or $M1$ is unlock and $M2$ is either a read or write of a lock.

Finally, previous work also discusses the hardware optimization of executing a write non-atomically [2]; i.e., making the value of a data write visible to a thread before it becomes visible to all threads. It has been shown that for the model described above, data writes and writes from well-structured locks and unlocks can be executed non-atomically [1].

We note an unfortunate overload of the term *atomic* – (1) the above usage to describe how a write is executed in hardware and (2) the C++ qualifier to denote a special class of memory operations. The default atomic write in C++ (qualified as sequentially consistent and discussed here) needs to be executed atomically by hardware, but another class of low-level atomic writes (discussed later) need not be executed atomically in the above sense, though it is still atomic in the sense that no other individual thread can see a partially updated value.

The model imposes significant restrictions for synchronization operations. For all practical purposes, synchronization operations must appear sequentially consistent with respect to each other. This means that atomic operations must execute in sequenced-before order and atomic writes must execute atomically [2], referred to henceforth as the sequenced-order and write-atomicity requirements respectively. For locks and unlocks, their special usage restrictions enable some optimizations as mentioned above, without violating the appearance of sequential consistency.

We make some further observations on allowable optimizations in [10].

3. Making Trylock Efficient

The preceding gives undesirably strong semantics to programs using calls that try to acquire a lock without blocking, such as `pthread_mutex_trylock()`, or lock acquisitions with timeouts. Consider the example in Figure 3, equivalent to one found in [12]:

This program essentially inverts the sense of the lock `l` by having a thread wait for T1 to *acquire* the lock, instead of waiting for the lock to be released. This is clearly not a desirable programming idiom.[4]

Based on a conventional interpretation of `trylock()`, in a sequentially consistent execution, the assertion in T2 cannot be executed (i.e., appear in the $<_T$ ordering) until after T1 acquires the lock and until after it assigns `x` the value of 42. The program is therefore data-race-free and, by our current semantics, the assertion cannot fail.

The difficulty is that the assertion *can* fail if either the compiler or hardware reorders the two statements executed by T1, by moving the assignment after the lock. Prohibiting such reordering on many architectures requires a memory fence before the lock. As mentioned in Section 2.1, for well-structured uses of locks and unlocks, such a reordering is safe. Thus, this fence represents unnecessary overhead, possibly doubling the cost of lock acquisition, without benefit to reasonably written code. As pointed out in [12], many lock implementations as a result fail to enforce this requirement, even though it appears to already exist in Posix.

There is potentially a similar issue with reordering the final failing `trylock()` and assertion in thread T2. Even Posix tries to allow this reordering.

The fundamental difficulty here is that `trylock()` reads the value written by T1's lock to infer that the lock has been acquired. This communication from a lock write to a trylock read is used to synchronize the accesses on `x`. We wish to prevent such use to avoid paying the overhead of the additional fence on all locks. Previous work has achieved this by explicitly distinguishing between different types of synchronization operations and/or redefining data operations that are separated only by such synchronization as races. For example, the data-race-free-1 model distinguishes pairable and unpairable synchronization [1], allowing only the former to prevent a data race. The Java memory model requires conflicting ordinary (data) operations to be explicitly ordered by volatiles and/or pairs of locks/unlocks to avoid a data race [25]. A happens-before relation is used to formalize these notions.

Instead of introducing the complexity of different types of synchronization or a happens-before relationship to define a data race, we propose a simple solution. We change the specification of `trylock()`, though preferably not its implementation, to not guarantee that it will succeed if the lock is available. The C++0x specification is expected to allow `trylock` to "spuriously fail" in this manner. This effectively prevents a failed `trylock` from reliably revealing anything about the state of the lock (and hence the write of the lock operation has no means to convey any synchronization information). In particular, it is now clear that the assertion in the above example may fail. The execution in which the assertion fails is now sequentially consistent; it simply involved a "spurious" `trylock()` failure.

Although simple, to our knowledge, this is the first solution that eliminates the need to provide a fence before a lock, while still maintaining a simple definition of a race (i.e., conflicting accesses adjacent to each other in the total order).

For the remainder of this discussion, the reader may assume that a `trylock()` allowing spurious failures is included. A successful `trylock()` is treated as `lock()`. An unsuccessful `trylock()` is treated by the memory model as a no-op.

4. The Cost of Sequentially Consistent Atomics

As mentioned in Section 2.1, the model so far requires atomics to appear sequentially consistent. When this work began, there was concern from various hardware vendors and software developers

[4] One of the reviewers points out that `trylock` could be, and sometimes is, used with a lock that is never released to ensure that an action is performed by only a single thread. This fails in our model, reflecting the fact that many current implementations fail to provide sequential consistency for this use, while we want to insist on it. In our model, the equivalent can be accomplished with more straightforward semantics with an atomic test_and_set.

```
            Initially X=Y=0
T1          T2          T3          T4
X=1         Y=1         r1=X        r3=Y
                        fence       fence
                        r2=Y        r4=X

r1=1, r2=0, r3=1, r4=0 violates write atomicity
```

Figure 4. Write-atomicity may be too expensive for Independent-Reads-Independent-Writes (IRIW).

that this requirement was excessive and unnecessarily restricted performance.

In principle, the performance impact of these requirements should be restricted only to synchronization operations. Unfortunately, most current processor instruction sets do not directly distinguish synchronization operations (Itanium is a notable exception). The most common way for compilers to convey memory ordering requirements to hardware is through fence or memory barrier instructions.

Fences were designed primarily to convey the sequenced-order (program order) requirement. For example, the strongest fences ensure that all memory operations sequenced-before the fence will appear to execute before all memory operations sequenced after the fence. Other variants impose ordering between different subsets of memory operations; e.g., a Store|Load fence orders previous stores before later loads. Some processors ensure certain orderings by default, removing the need for fences for those cases; e.g., AMD64 and Intel 64 implicitly provide Load|Load and Load|Store fence semantics after each load and Store|Store fence semantics after each store.

When this work began, many specifications of fences were ambiguous about their impact on the hardware write-atomicity requirement as described in Section 2.1 (notable exceptions are Alpha and Sun). Some processor vendors argued that full write-atomicity was too expensive and software developers argued that weak versions of write-atomicity would suffice. Section 4.1 discusses the cost of enforcing write-atomicity in hardware and why it may be unnecessary for some programs. Section 4.2 then systematically shows how meaningful relaxations of write-atomicity result in unintended consequences for other programs. As a result, despite many attempts, it was difficult to formalize semantics for synchronization operations that were meaningfully weaker than sequential consistency for hardware and provided a simple enough interface for most programmers.

In this section, all variables in all examples are atomics. (The syntax does not reflect the current C++ working paper.)

4.1 Cost of Enforcing Write-Atomicity

Consider the example in Figure 4, referred to as the Independent-Reads-Independent-Writes (IRIW) example. The fences ensure that the reads will execute in program order, but this does not guarantee sequential consistency if the writes execute non-atomically. For example, if the writes to X and Y propagate in different orders to threads T3 and T4, the outcome in the figure violating sequential consistency can occur (T3 sees the new value of X and the old value of Y and vice versa for T4).

Systems with ownership-based invalidation protocols and single-core/single-threaded processors can avoid the non-sequentially consistent outcome in a straightforward way. Consider a typical such system employing a directory-based cache coherence protocol. If T1 does not already have ownership of X, then it must request it from the directory. The directory then sends invalidations for all cached copies of X and either it or T1 collect acknowledgements for these invalidations. To see the new value of X, T3 must first go to the directory which will forward the request to T1. To ensure writes appear atomic, the system simply needs to ensure that all copies of X are invalidated (i.e., all acknowledgements received) before T3 gets the updated copy of X, and analogously, all outstanding copies of Y are invalidated before T4 gets an updated copy of Y. Now it is no longer possible for both T3 and T4 to read the old values of Y and X respectively.

The key to ensuring sequential consistency for the above example is that a read is not allowed to return a new value for the accessed location until all older copies of that location are invalidated. This requirement can be somewhat relaxed further for processors that require explicit fences for ordering reads – the read can return a new value as long as a subsequent fence waits for all old copies of that location to be invalidated. Following [2], we refer to this as the "read-others'-write-early" restriction.

Note that as described above, the read-others'-write-early restriction does not require any global serialization point for all operations to all locations (e.g., a bus). It simply requires a serialization point for reads and writes to the *same* location, which is usually provided by the cache coherence protocol (the need for which is widely accepted). Further, this per-location serialization is technically only required for atomic operations. Unfortunately, the use of fences for memory ordering obfuscates which memory accesses are the atomic ones; therefore, without a mechanism to distinguish individual writes as atomic, our system must preserve write atomicity for all writes.

The advent of multicore and simultaneous multithreading (SMT), where threads may share a data cache or store queues, has previously unexplored implications for the read-others'-write-early restriction. These architectures offer tempting performance optimization opportunities that can violate sequential consistency for the IRIW example as follows. Suppose T1 and T3 share an L1 writethrough data cache. Suppose T3's read of X occurs shortly after T1's write to X. It is tempting to return the new value of X to T3, even if T1's ownership request for X has not yet made its way through the lower levels of the cache hierarchy. If T2 and T4 share a cache, an analogous situation can occur for their write and read of Y – T4 reads the new value of Y even before T2's ownership request reaches the rest of the memory system. It is now possible that both T3 and T4 read the old values of Y and X respectively (from their caches), violating sequential consistency.

Thus, while previously the read-others'-write-early restriction appeared to have (acceptable) implications only for the main memory system and the cache coherence protocol, new SMT and multicore architectures move this restriction all the way to the first level cache and even within the processor core (if there are shared store queues). At the same time, most programmers agree that the IRIW code does not represent a useful programming idiom, and imposing restrictions to provide sequential consistency for it appears excessive and unnecessary.

For this reason, some existing machines do not provide efficient methods for ensuring sequential consistency for IRIW and there was initial reluctance to adopting sequentially consistent semantics for atomics.

4.2 Unintended Consequences of Relaxing Write-Atomicity

We next show examples where relaxing the read-others'-write-early requirement in ways described for IRIW can give unacceptable results.

4.2.1 Write-to-Read Causality (WRC) Must be Upheld

Figure 5 illustrates a simple causal effect. Thread T1 writes a new value of X, T2 reads it, executes a fence, and writes a new value of Y. T3 reads the new Y, executes a fence, and then reads X. Sequential consistency requires that T3 return the new value for

```
          Initially X=Y=0
T1            T2            T3
X=1         r1=X          r2=Y
            fence         fence
            Y=1           r3=X

r1=1, r2=1, r3=0 violates write atomicity
```

Figure 5. Write-to-Read Causality (WRC) must be respected

```
          Initially X=Y=0
T1            T2            T3
X=1         r1=X          Y=1
            fence         fence
            r2=Y          r3=X

r1=1, r2=0, r3=0 violates write atomicity
```

Figure 6. Should Read-to-Write Causality (RWC) be respected?

X. We call this example Write-to-Read Causality (WRC) because a write of a new value by a thread followed by a read of the same value by another thread is used to establish a causal connection between the threads. Most programmers agree that this causality should be respected, and the violation of sequential consistency shown in Figure 5 should not be allowed.

Now consider applying the IRIW optimization to the WRC example. Suppose T1 and T2 share an L1 writethrough cache and T3 is on a separate system node. Suppose T2 is allowed to read T1's update to X early. Then it may be possible for T3 to receive the invalidation for Y before that for X, resulting in T3 reading the new value of Y and the old value of X.

One solution that preserves sequential consistency for WRC while retaining much of the impact of the IRIW optimization is to ensure that writes separated by a fence from a given system node are seen in the same order by all nodes. This solution continues to allow reads within the node to return a new value early. We next show that this solution unfortunately does not suffice for other programs.

4.2.2 Read-to-Write Causality (RWC)

Figure 6 illustrates an example similar to WRC, except that the causal relationship between T2 and T3 is established by T2's read of an old value followed by T3's write of a new value to Y. The IRIW optimization can violate sequential consistency for this example similar to WRC. Unfortunately, the WRC solution (ordering writes separated by fences from the same node) does not work in this case since each node (T1/T2 or T3) contains at most one write.

Unlike the WRC example, some programmers may find this violation of read-to-write causality acceptable; however, it is no longer as clear-cut as the IRIW code.

4.2.3 Subtle Interplay Between Coherence and Causality

Our final example illustrates how IRIW style optimizations can interfere in a non-intuitive way with a subtle interplay between cache coherence and write-to-read causality, referred to as CC. Cache coherence is a widely accepted property that guarantees that writes to the *same* location are seen in the same order by all threads. There is wide consensus that this property must be respected (for synchronization operations) to write meaningful code. There is also consensus that write-to-read causality must be respected. Figure 7 shows that IRIW style optimizations make it difficult to compose cache coherence and simple write-to-read causality interactions.

In Figure 7, assume again that T1 and T2 are on the same node with a shared writethrough L1 cache. Assume T3 and T4

```
           Initially X=Y=0
T1          T2          T3          T4
X=1       r1=X        Y=1         r3=X
          fence       fence       fence
          r2=Y        X=2         r4=X

r1=1, r2=0, r3=2, r4=1 violates write atomicity
```

Figure 7. CC: Subtle interplay between cache coherence and write-to-read causality.

are on separate nodes. As with RWC, T2 reads T1's new value (1) of X early, executes a fence, and reads an old value for Y. Now assume T3 writes the new value of Y, executes a fence, and then also updates X to 2. After all of T3's operations complete in the memory system, T4 reads X (returns 2) and executes a fence. At this point, assume T1's write permission request for X goes through the memory hierarchy and all its invalidations are done. Then when T4 reads X for the second time, it gets 1.

Thus, the IRIW optimization can result in the violation of sequential consistency shown in Figure 7. As with RWC, the WRC solution does not apply because no node has more than one write. No other simple solutions that would retain the advantage of the IRIW optimization are apparent.

A memory model that would violate sequential consistency for the above example seems difficult to formulate and reason with because such a model would not be able to easily compose cache coherence and write-to-read causality. Specifically, in the above example, T4 establishes that X=2 was serialized in the memory hierarchy before X=1. Cache coherence therefore allows reasoning that T2's read of 1 for X must have occurred after T3's write of 2 for X. Using a write-to-read causality argument leads to the inference that T3's write of Y must have occurred before T2's read of Y, which should return 1 and not 0.

Thus, the IRIW optimization and the consequent result in Figure 7 precludes inferences from a simple and intuitive composition of cache coherence and write-to-read causality.

4.3 Implications for Current Processors

The above examples show that a departure from sequential consistency for synchronization operations can lead to subtle non-intuitive behaviors that appear difficult to formalize in an intuitive fashion. We therefore retained sequential consistency semantics for default atomics and synchronization operations.

Influenced by our work, the new AMD64 [4] and Intel 64 [22] memory ordering specifications now provide a clear way to guarantee sequentially consistent semantics, although these specifications require atomic writes to be mapped to atomic `xchg` instructions (which are read-modify-write instructions). Hardware now need only ensure that `xchg` writes execute atomically. Additionally, with these specifications, `xchg` also implicitly ensures semantics of a Store|Load fence, removing the need for an explicit fence after an atomic store (such a fence would otherwise be required for the sequenced-before requirement).

While it may seem cumbersome and inefficient to convert atomic stores to read-modify-writes, the following observations make this a reasonable compromise: (1) it is better to pay a penalty on stores than on loads since the former are less frequent and (2) the Store|Load fence replaced by the read-modify-write is as expensive on many processors today.

There are three other approaches that could be used to achieve sequential consistency for atomics, without requiring converting all atomic stores to read-modify-writes. First, the processor ISA could provide a simple mechanism to distinguish atomic stores

(e.g., Itanium's st.rel) instead of having to use read-modify-writes. Hardware could then ensure that these stores execute atomically.

Second, write-atomicity could simply be provided for all writes, as with Sun's total store order (TSO) and the Alpha memory models. The mechanisms to implement this are similar to those for the read-modify-write and st.rel type solutions, except that they need to be invoked for all writes.

Third, many processors today use speculative approaches to reorder memory operations beyond that allowed by the memory model; e.g., AMD's and Intel's processors speculatively reorder loads even though the memory model disallows it [19]. A similar approach could be used to allow a read to return a value early from a shared cache. So far, the literature on such optimizations has focused on benefits for speculatively relaxing the sequenced-order requirement; we are not aware of prior work on understanding the impact of speculatively relaxing write-atomicity.

To the best of our knowledge, most existing machines today provide write-atomicity by default. A notable exception is some PowerPC machines which require a particularly expensive form of a fence instruction after atomic loads to achieve sequentially consistent results for the RWC and CC examples.

5. Semantics of Data Races

For languages like Java, it is critical to define the semantics of all programs, including those with data races. Java must support the execution of untrusted "sandboxed" code. Clearly such code can introduce data races, and the language must guarantee that at least basic security properties are not violated, even in the presence of such races. Hence the Java memory model [25] is careful to give reasonable semantics to programs with data races, even at the cost of significant complexity in the specification.

For C++, there is no such issue. Initially, there was still some concern that we should limit the allowable behavior for programs with races. However, in the end, we decided to leave the semantics of such programs completely undefined. In the current working paper for the C++ standard, in spite of discussions such as [26], *there are no benign data races*.

The basic arguments for undefined data race semantics in C++ are:

1. Although generally under-appreciated, it is effectively the status quo. Pthreads states [21] "Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it." As we mention in the introduction, Ada earlier took the same approach. The intent behind win32 threads appears to have been similar.
2. Since the C++ working paper provides low-level atomics with very weak, and hence cheaply implementable, ordering properties, there is little to be gained by allowing races, other than allowing code to be obfuscated. We effectively require only that such races be annotated by the programmer. Since the result is usually exceedingly subtle, we believe this should be required by any reasonable coding standard in any case.
3. Giving Java-like semantics to data races may greatly increase the cost of some C++ constructs. It would presumably require that we not expose uninitialized virtual function tables, even in the event of a race, since those could otherwise result in a wild branch. This in turn often requires fences on object construction. In Java, this is arguably less major, since object construction is always associated with memory allocation, which typically already carries some cost. This does not apply to C++.
4. Current compiler optimizations often assume that objects do not change unless there is an intervening assignment through a potential alias. Violating such a built-in assumption can cause very complicated effects that will be very hard to explain to a programmer, or to delimit in the standard. We believe this assumption is sufficiently ingrained in current optimizers that it would be very difficult to effectively remove it.

As an example of the last phenomenon, consider a relatively simple example, which does not include any synchronization code. Assume `x` is a shared global and everything else is local:

```
unsigned i = x;
if (i < 2) {
    foo: ...
    switch (i) {
        case 0: ...; break;
        case 1: ...; break;
        default: ...;
    }
}
```

Assume the code at label `foo` is fairly complex, and forces `i` to be spilled and that the `switch` implementation uses a branch table. (In reality, the second assumption might require a larger `switch` statement.)

The compiler now performs the following reasonable optimizations:

- It notices that `i` and `x` (in its view of the world) contain the same values. Hence when `i` is spilled, there is no need to store it; the value can just be reloaded from `x`.
- It notices, e.g. by value range analysis as in [20] that the `switch` expression is either 0 or 1, and hence eliminates the bounds check on the branch table access and the branch to the default case.

Now consider the case in which, unbeknownst to the compiler, there is actually a race on `x`, and its value changes to 5 during the execution of the code labeled by `foo`. The results are:

1. When `i` is reloaded for the evaluation of the `switch` expression, it gets the value 5 instead of its original value.
2. The branch table is accessed with an out-of-bounds index of 5, resulting in a garbage branch target.
3. We take a wild branch, to arbitrary code.

The result would only be slightly less surprising if the switch tested `x` instead of `i`. It seems difficult to explain either behavior in a language standard in any way other than to leave it completely undefined.[5]

Finally, even after the large effort to define semantics for data races in Java, recent work by David Aspinall and Jaroslav Sevcik has uncovered some bugs in that specification [5].

Thus we decided to leave the semantics of races undefined, in spite of some potential negative effects on debuggability.

5.1 Compiler-introduced Data Races

Since we expect the next C++ standard to completely prohibit data races at the source level, it will not be possible to write a

[5] It is of course not impossible for compilers to avoid this kind of behavior. Java compilers must avoid it, and C++ compilers rarely exhibit it in practice. But our experience has been that compiler writers are disinclined to promise it will not happen, particularly since current Posix and C or C++ standards clearly allow the fundamental assumption that ordinary variables do not change asynchronously.

portable C++-to-C++ source translator that introduces data races. In particular, source-to-source optimizers may never introduce a data race.

This requirement disallows some important optimizations; specifically, it prevents such an optimizer from introducing speculative loads as well as speculative stores. Optimizations such as Partial Redundancy Elimination commonly introduce speculative loads. For example, the requirement prevents a potentially shared but loop invariant variable `x` from being loaded into a register once outside the loop, if there is any possibility that it might not actually be referenced in the loop. The requirement also prevents a load from being scheduled before it was clear that the value was actually needed, though a prefetch of such a value would generally still be correct.

It is important to note that this restriction applies only if the *target* language of the optimizer prohibits data races. We are not aware of any hardware architectures that do so. The above optimizations introduce potentially racing loads of values whose results are not used. Machine architectures allow those without altering the meaning of the remaining code. Hence compilers that target a conventional hardware architecture may continue to insert speculative loads.

Such optimizers may generally still *not* insert speculative stores, since those could overwrite a legitimate store in another thread, thus changing the semantics of the program, even when the original program contained no race.

If a compiler were to translate to an architecture that gives undefined semantics for races at the machine level, then the compiler would be prohibited from introducing speculative loads as well. This arises for example, if the target is a virtual machine that itself detects races as in [16]. And that is as it should be. Any introduced speculative loads could race with a store in another thread, thus resulting in the diagnosis of a race where the original program had none.[6] By the same reasoning, a C++-to-C++ translator should not introduce races, since it is not aware of the final target.

6. C++ Model Supporting Low-Level Atomics

As mentioned earlier, enforcing sequential consistency is expensive on some platforms, and there are some frequently used idioms for which sequential consistency is not required.

For example, it is common to use counters that are frequently incremented by multiple threads, but are only read after all threads complete. Our semantics, since they guarantee sequential consistency, require that all memory updates performed prior to a counter update become visible after any later counter update in another thread. This property typically requires the addition of fences, which in some cases are substantially more expensive than the rest of the operation.

Similarly, in the absence of context information, sequentially consistent atomic stores often require two fences, one before and one after the store. The first ensures that memory operations sequenced before the store are visible to any thread that sees the result of the store. The second, often more expensive, fence ensures that the store is not reordered with respect to a later atomic load. The latter property is required when implementing Dekker's algorithm, for example, but is often unnecessary.

The actual C++ atomics library specification (see [13] or Chapter 29 of [14]) supports low-level atomic operations that allow the programmer to explicitly specify memory ordering constraints, allowing close to optimal implementations of these idioms by *very careful* programmers.

As presented so far, our memory model inherently does not support non-sequentially-consistent synchronization operations. Assume that `incr` increments a variable without enforcing any sort of visibility ordering. (Alpha, ARM, and PowerPC, for example, would allow a significantly cheaper implementation of `incr` than its sequentially consistent counterpart.) Now consider the following, where `x` and `y` are atomic, `z` is not, and all are initially zero:

```
Thread 1:        Thread 2:
incr(x);         incr(y);
r1 = y;          r2 = x;
if (r1 == 0)     if (r2 == 0)
  z = 1;           z = 2;
```

In a sequentially consistent execution, `r1` and `r2` cannot both be zero, and hence there is no data race in a sequentially consistent execution. However, any implementation that takes advantage of the lack of ordering between the atomic operations will allow both values to be zero, and hence potentially encounter a data race. Hence this should be given undefined semantics by all the arguments in the preceding section. But this requires that we define a notion of data race in terms of the actual language semantics, not in terms of a sequentially consistent execution.

Here we give an alternate memory model specification that allows extension to low-level atomics. This description is a more mathematical formulation of the C++ working paper (see [27, 8] or primarily section 1.10 of [14]). For the time being, we continue to omit low-level atomics. Thus the model presented here is intended to be equivalent to the earlier one (as demonstrated in the following sections). The last section of this paper then describes briefly how low-level atomics are actually incorporated into the C++ working paper.

This version parallels the simpler parts of the Java memory model [25] much more closely than our original version, though there are some differences.

Define a program execution to be

1. A set of thread executions
2. A mapping W from atomic loads and atomic read-modify-write operations to atomic stores and atomic-read-modify-write operations to the same location, and from lock acquisitions to lock releases of the same lock. W is intended to map each read-like operation to the corresponding write whose value it observes.
3. An irreflexive total order $<_S$ of atomic operations, intended to reflect the global order in which they are executed. [7]

We define an atomic load, atomic read-modify-write, or lock acquisition to be an *acquire* operation on the read locations. Conversely, an atomic store, atomic-read-modify-write, or lock release is a release operation on the affected locations. We define a memory action A to *synchronize with* a memory action B if B is an acquire operation on a location, A is a release operation on the same location, and $W(B) = A$.

We define *happens-before* ($<_{hb}$) to be the smallest relation on memory actions such that

- If a is sequenced before b, then a happens before b.
- If a synchronizes with b, then a happens before b.

[6] Arguably, the same holds for a physical machine target if the resulting execution is analyzed for races, as in [26]. Here the race semantics are well defined, but compiler-introduced races still result in undesirable output. Unfortunately [26] does not make as clear a distinction between source-level and machine-level races.

[7] This corresponds roughly to Java's synchronization order. In the C++ approach, the requirement for this total order is specified as part of the atomic library, in chapter 29 of [14].

- If a happens before b and b happens before c, then a happens before c.

We define a *visible side effect* with respect to a load or read-modify-write operation b to be an update (store or read-modify-write operation) a to the same location l such that a happens before b, but there is no intervening update c to l such that a happens before c and c happens before b. (The intent is that in the absence of races, ordinary loads see the unique visible side effect.)

We define a program execution to be *consistent* (a notion that is not explicit in the C++ working paper) if

1. Each thread execution is internally consistent, given the values read from memory.
2. The order $<_S$ is consistent with happens-before, i.e. if a happens before b, then $a <_S b$.
3. For each non-atomic load l, $W(a)$ is a visible side effect with respect to l. (If there are no data races, it is the unique visible side effect corresponding to l.)
4. For each atomic (i.e. synchronization) load or atomic-read-modify-write operation a, then $W(a)$ is the last preceding update in $<_S$ corresponding to the same location.
5. Lock and unlock operations on each individual lock are totally ordered by happens-before, and alternate in each such individual order. Furthermore each lock operation is sequenced before the next unlock operation in this total order, if there is one, i.e. locks are released only by the acquiring thread.

A consistent execution contains a `type 2 data race` if two data accesses to the same memory location are unordered by happens-before. We can now specify the C++ memory model simply as:

- If a program (on a given input) has a consistent execution with a (type 2) data race, then its behavior is undefined.
- Otherwise, the program (on the same input) behaves according to one if its consistent executions.

The following two sections show that this characterization is equivalent to the original one in section 2: First, data-race-free programs still have sequentially consistent semantics. Second, the two notions of data race are equivalent. We then conclude with an outline of the additional changes needed to actually include low-level atomics.

7. Sequential Consistency for Data-Race-Free Programs

THEOREM 7.1. *The model defined in Section 6 provides sequential consistency to programs whose consistent executions do not contain a type 2 data race.*

Proof Consider a type 2 data-race-free program and its consistent execution (on the given input). We need to show that this execution exhibits sequentially consistent behavior.

The corresponding happens-before relation ($<_{hb}$) and synchronization order $<_S$ are irreflexive and consistent, and hence can be extended to a strict total order $<_T$.

Clearly the actions of each thread appear in $<_T$ in sequenced-before order. Since lock operations are totally ordered by happens-before, they must occur in the same order in $<_T$.

We can say little about where a failed `trylock()` operation on a lock l appears in $<_T$. But, since we assume that `trylock()` may fail spuriously, it does not matter. A failure outcome is acceptable no matter what state the lock was left in by the last preceding operation (in $<_T$) on l. No matter where the failed `trylock()` appears in $<_T$, the operations on l could have been executed in that order, and produced the original results.

It remains to be shown that each load sees the last preceding store in $<_T$ that stores to the same location.

Clearly this is true for operations on atomic objects, since all such operations appear in the same order as in $<_S$, and each load in $<_S$ sees the preceding store in $<_S$.

From here on, we consider only ordinary, non-atomic memory operations.

Consider a store operation S_s seen by a load L.

S_s must be a visible side effect with respect to L, and hence must happen before L. Hence S_s precedes L in $<_T$.

Assume that another store S_b appears between S_s and L in $<_T$.

We know from the fact that $<_T$ is an extension of $<_{hb}$, that we cannot have either of $L <_{hb} S_b$ or $S_b <_{hb} S_s$ since that would be inconsistent with the reverse ordering in $<_T$.

However all three operations conflict and we have no data races. Hence they must all be ordered by $<_{hb}$, and S_b must also be happens-before ordered between the other two. But this contradicts the fact that S_s must be a visible side effect with respect to L, concluding the proof. •

Although this is a useful theorem, it does not (yet) ensure that our refined memory model is equivalent to the original one. We must also show that a program that is data-race-free by our original type 1 definition is also data-race-free by our revised (type 2) definition.

8. Equivalence of Race Definitions

THEOREM 8.1. *If a program allows a type 2 data race in a consistent execution, then there exists a sequentially consistent execution, with two conflicting actions, neither of which happens before the other.*[8]

In effect, we only need to look at sequentially consistent executions in order to determine whether there is a data race in a consistent execution. For example, a program such as the one in figure 1 cannot possibly contain a race, since in a sequentially consistent execution, each variable is accessed by only one thread.

Proof We show that any type 2 data race in a consistent execution corresponds to a data race (again defined in terms of happens-before, not simultaneous execution) in a sequentially consistent execution.

Consider a consistent execution with a data race. Let $<_T$ be the total extension of the happens-before and synchronization orders, as constructed above.

Consider the longest prefix P of $<_T$ that contains no data race. Note that each load in P must see a store that precedes it in either the synchronization or happens-before orders. Hence each load in P must see a store that is also in P. Similarly each lock operation must see the state produced by another lock operation also in P, or it must be a failed `trylock()` whose outcome could have occurred if it had seen such a state.

By the arguments of the preceding section, the original execution restricted to P is equivalent to the prefix of a sequentially consistent execution.

The next element N of $<_T$ following P must be an ordinary memory access that introduces a race. If N is a store operation, consider the original execution restricted to $P \cup \{N\}$. Otherwise consider the same execution except that N sees the value written by the last write to the same variable in P.

[8] The latter is essentially the condition used in [25] to define "correctly synchronized" for Java.

In either case, the resulting execution (of P plus the operation introducing the race) is a prefix of a sequentially consistent execution; if N was a write, it could not have been seen by any of the reads in P, since those reads were ordered before N in $<_T$; if N was a read, it was modified to ensure that it sees the last applicable write in P. In either case, we have a sequentially consistent execution of the program that exhibits the data race (a straightforward extension of $P \cup \{N\}$). •

THEOREM 8.2. *A program allows a type 2 data race on a given input if and only if there exists a sequentially consistent execution in which two unordered conflicting actions are adjacent in the sequential interleaving, i.e. it allows a type 1 data race.*

Proof Assume we have a sequentially consistent execution with total order $<_T$ and a type 1 data race. There is a straightforward mapping of this execution, up to the first such data race, to a consistent execution as defined in section 6 in which each load sees the corresponding preceding store in $<_T$, and the synchronization order is just a restriction of $<_T$. It is easy to see that the type 1 data race maps to a type 2 data race.

It remains to show the converse: If we have a type 2 data race, there must be a sequentially consistent execution with conflicting adjacent operations.

Start with the execution restricted to $P \cup \{N\}$, as above, with any value read by N adjusted as needed, also as above. We know that nothing in this partial execution depends on the value read by N. Let M be the other memory reference involved in the race.

We can further restrict the execution to those operations that happen before either M or N. This set still consists of prefixes of the sequences of operations performed by each thread. Since each load sees a store that happens before it, the omitted operations cannot impact the remaining execution.

Define a partial order (*race-order*) on $\{x \mid x <_{hb} M\} \cup \{x \mid x <_{hb} N\} \cup \{M\} \cup \{N\}$ which orders everything in the first two sets before the other two, but imposes no other order.

Race-order is consistent with happens-before and synchronization order. It imposes no additional order on the initial subset. Neither M nor N is ordered by the synchronization order, and neither is race ordered or happens before any of the elements in the first subset. If we had a cycle $A_0, A_1, ..., A_n = A_0$, where each element of the sequence happens before or is race-ordered or synchronization ordered before the next, neither M nor N could thus appear in the cycle. This is impossible, since happens-before and the synchronization order are required to be consistent.

We can thus construct the total order $<_T$ as a total extension of the reflexive transitive closure of the union of happens-before, synchronization order, and race order. By the preceding observation, this exists.

By the same arguments as in the proof of theorem 7.1, every memory read must see the preceding write in this sequence, except possibly N, since it is the only one that may see a value stored by a racing operation. But we can again simply adjust the value seen by N to obtain the property we desire, without affecting the rest of the execution. Thus $<_T$ gives us the desired sequentially consistent execution in which the last two operations, namely M and N, conflict. •

9. Model Adjustments for Low-Level Atomics

The C++ working paper provides low-level atomic operations that are explicitly parameterized with respect to a memory ordering constraint. The value of an atomic variable `x` can be retrieved, for example, with `x.load(memory_order_relaxed)` , allowing it to be reordered with other memory operations. In terms of the memory model, this specifies that the load is never an acquire operation, and hence does not contribute to the synchronizes-with ordering. For read-modify-write operations, the programmer can specify whether the operation acts as an acquire operation, a release operation, neither ("relaxed"), or both.

In order to include such low-level atomics in our memory model, as is done in the C++ working paper, we need several refinements to the model in section 6:

1. The total order S of synchronization operations contains only high-level (sequentially consistent) atomic operations. (These can also be specified with an explicit `memory_order_seq_cst` parameter.)
2. We do however still want updates to a *single* variable to occur in a total order, dubbed the *modification order* in the standard.
3. There was a feeling that, for example a weakly ordered atomic increment performed by another thread between an atomic store and load should not break the synchronizes-with relationship between the store and the load. Hence the definition of synchronizes-with was strengthened to cover this case (see "release sequence" in the C++ working paper). The precise definition in the working paper is a compromise between ease of use and implementability on common hardware.
4. Since S no longer includes all synchronization operations, the value seen by an atomic load is no longer uniquely determined by S. We define a *visible sequence* with respect to an atomic load or read-modify-write operation a of location l, to be the maximal subsequence V of l's modification order such that every element of V either is or occurs after a visible side effect of a, but no element of V happens after a. V then represents all possible updates that might be seen by a.

Although we can now accommodate low-level atomics in the memory model, it is important to remember that these are still very hard to use correctly, and very much an "experts-only" feature. Most users will benefit only indirectly from libraries written using these facilities.

10. Conclusions

We have outlined the foundations of the C++ threads memory model.

From the user's perspective, we provide a simple programming model. In return for avoiding data races or, equivalently, identifying variables and other objects involved in data races as `atomic`, most users can ignore the intricacies of hardware memory models and compiler optimizations; they are guaranteed sequentially consistent execution.

All of this can be based on the most intuitive definition of a data race: simultaneous execution of conflicting operations. The one place in which modern machine architectures do unavoidably show through slightly is that updates to adjacent bit-fields conflict; otherwise, operations conflict only when they touch the same object.

For those few users whose performance requirements cannot be satisfied with either locks or sequentially consistent atomic operations, we provide low-level, explicitly ordered, atomics, which trade simplicity for cross-platform performance.

From the compiler implementors perspective, we preserve the guarantee that ordinary variables do not appear to change asynchronously. Hence, standard program analyses remain valid, except for objects of `atomic` type, even in the presence of threads. In return, the implementation must refrain from introducing user visible data races, for example, as a result of rewriting adjacent structure fields or register promotion. More complete implementation guidelines are given in [10].

From the hardware implementors perspective, aside from now standard features, such as compare-and-swap and similar operations, we need a modest cost facility that allows us to implement sequentially consistent atomics, and particularly write atomicity. We do not need write atomicity for all store operations; but we do need it for `atomic` operations. Ideally, sequentially consistent atomics should be implementable with very small overhead for load operations.

Acknowledgments

This effort has benefitted greatly from contributions by many others, including especially Herb Sutter, Doug Lea, Paul McKenney, Bratin Saha, Jeremy Manson, Bill Pugh. We clearly would not have been successful at generating standards-appropriate text without the help of Clark Nelson and Lawrence Crowl, our coauthors for [13] and [27]. Lawrence Crowl is also responsible for most of the detailed design of the atomics interface. The anonymous reviewers provided a number of useful suggestions. Mark Hill gave comments on a previous version of the paper.

This work has made it this far through the standards process only with the the help of processor vendors including AMD, ARM, IBM, and Intel, who were helpful both in pointing out potential issues, and cooperating with the standardization process, even when significant effort was required.

References

[1] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. 17th Intl. Symp. Computer Architecture*, pages 2–14, 1990.

[4] AMD Corp. *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*, July 2007.

[5] D. Aspinall and J. Sevcik. Java memory model examples: Good, bad, and ugly. VAMP07 Proceedings `http://www.cs.ru.nl/~chaack/VAMP07/`, 2007.

[6] H. Boehm and N. Maclaren. Should volatile acquire atomicity and thread visibility semantics? C++ standards committee paper WG21/N2016 = J16/06-0086, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2016.html`, April 2006.

[7] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. Conf. on Programming Language Design and Implementation*, 2005.

[8] H.-J. Boehm. A less formal explanation of the proposed c++ concurrency memory model. C++ standards committee paper WG21/N2480 = J16/07-350, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2480.html`, December 2007.

[9] H.-J. Boehm. Memory model rationales. C++ standards committee paper WG21/N2176 = J16/07-0036, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2176.html`, March 2007.

[10] H.-J. Boehm. N2338: Concurrency memory model compiler consequences. C++ standards committee paper WG21/N2338=J16/07-198, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2338.htm`, August 2007.

[11] H.-J. Boehm. N2392: A memory model for c++: Sequential consistency for race-free programs. C++ standards committee paper WG21/N2392=J16/07-252, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2392.htm`, September 2007.

[12] H.-J. Boehm. Reordering constraints for pthread-style locks. In *Proc. 12th Symp. Principles and Practice of Parallel Programming*, pages 173–182, 2007.

[13] H.-J. Boehm and L. Crowl. C++ atomic types and operations. C++ standards committee paper WG21/N2427=J16/07-0297, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2427.htm`, October 2007.

[14] C++ Standards Committee, Pete Becker, ed. Working Draft, Standard for Programming Language C++. C++ standards committee paper WG21/N2461=J16/07-0331, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2461.pdf, October 2007.

[15] L. Ceze et al. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. Intl. Symp. on Computer Architecture*, 2007.

[16] T. Elmas, S. Qadeer, and S. Tasiran. A race and transaction-aware java runtime. In *Proc. Conf. on Programming Language Design and Implementation*, pages 245–255, 2007.

[17] K. Gharachorloo. *Memory Consistency Models for Shared Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[18] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 15–26, 1990.

[19] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. Intl. Conf. on Parallel Processing*, pages I355–I364, 1991.

[20] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Software Engineering*, 3(3), May 1977.

[21] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. IEEE, 2001.

[22] Intel Corp. *Intel 64 Architecture Memory Ordering White Paper*, August 2007. `http://www.intel.com/products/processor/manuals/318147.pdf`.

[23] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in titanium. In *Proceedings of the 2005 ACM/IEEE SC—05 Conference (SC'05)*, page November, 2005.

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

[25] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proc. Symp. on Principles of Programming Languages*, 2005.

[26] S. Narayanasamy et al. Automatically classifying benign and harmful data races using replay analysis. In *Proc. Conf. on Programming Language Design and Implementation*, pages 22–31, 2007.

[27] C. Nelson and H.-J. Boehm. Concurrency memory model (final revision). C++ standards committee paper WG21/N2429=J16/07-0299, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2429.htm`, October 2007.

[28] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proc. Symposium on Parallel Algorithms and Architectures*, 1997.

[29] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[30] Z. Sura et al. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Symp. Principles and Practice of Parallel Programming*, 2005.

[31] H. Sutter. Prism: A principle-based sequential memory model for microsoft native code platforms. C++ standards committee paper WG21/N2197 = J16/07-0057, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2197.pdf`, March 2007.

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

[33] T. Wenisch et al. Mechanisms for Store-wait-free Multiprocessors. In *Proc. Intl. Symp. on Computer Architecture*, 2007.