Document Number: WG21/N1876=J16/05-0136
Date: 2005-08-26
Reply to: Hans Boehm
Hans.Boehm@hp.com
1501 Page Mill Rd., MS 1138
Palo Alto CA 94304 USA

Memory model for multithreaded C++: August 2005 status update

Hans Boehm Doug Lea Bill Pugh

Abstract

The C++ Standard defines single-threaded program execution. Fundamentally, multithreaded execution requires a much more refined memory and execution model. C++ threading libraries are in the awkward situation of specifying (implicitly or explicitly) an extended memory model for C++ in order to specify program execution. We have been working on integrating a memory model suitable for multithreaded execution into the C++ Standard.

This document is a continuation of N1680=04-0120 and N1777=05-0037. It describes some of the issues we have discussed and tried to address since the Lillehammer meeting and our current direction. Its main goal is to ensure that this direction remains consistent with the views of the committee.

It has turned out that the more challenging problems are imposed on us by our desire to go beyond pthread-like primitives, and support a general purpose atomic operations library, which allows the construction of lock-free programs which efficiently utilize primitives most hardware provides for that purpose. Hence we will begin with our current thinking about this library.

1 Atomic Operations Library

We had assumed from the start that thread support for C++ would have to include access to low level atomic operations.

These operations allow the development of lock-free code, a notoriously difficult task, but one that is often necessary for high performance implementations of some low level functionality. These are primitives aimed at expert programmers. Most of their uses would be hidden in other libraries.

It has become increasingly clear that the diversity of operations in this library has a profound effect on the memory model. The relatively simple approaches we were considering earlier do not allow us to describe some of the primitives we would like to provide here.[1]

[1] The simplest approaches also don't allow us to describe certain lock semantics. We would

1.1 Atomics library approach

We have been discussing a library design in which primitives atomically read and/or update a memory location, and may optionally provide some memory ordering guarantees.

Nearly all lock-free algorithms require both atomic, i.e. indivisible, operations on certain pieces of data, as well as a mechanism for specifying ordering constraints. Allowing them to be combined in a single operation (as opposed to providing separate atomic operations and "memory barriers") makes it possible to cleanly and precisely express the programmer's intent, avoids some unnecessary constraints on particularly compiler reordering, and makes it easy to take advantage of hardware primitives that often combine them.[2]

For example, atomically updated integers would provide operations such as

load Atomically read the value of the integer.

store Atomically replace the value of the integer.

fetch and add Atomically add a value to the integer.

store with release ordering semantics Atomically replace the value of the integer, and ensure that all prior memory operations executed by this thread become visible to other threads before the update.

load with acquire ordering semantics Atomically load the value of the integer, and ensure that all later memory operations performed by this thread become visible after the load.

The last two together can be used to correctly implement a flag denoting, for example, that an object has been properly initialized. The ordering semantics ensure that the initialization operations performed by one thread before it sets the flag (using release ordering) are visible to another thread which reads a set flag (using acquire semantics), and then expects to see an initialized object.

On the vast majority of existing X86 processors, the "load" and "load with acquire ordering semantics" primitives could use the same hardware implementation, and would merely impose different compiler constraints. Other processor architectures are split as to whether they require separate implementations. The same applies to "store" and "store with release semantics".

We expect that in the final version of the library, operations such as load and store will be parameterized in some form with an ordering constraint (such as "release", "acquire", or "none").

A current, still very rough, draft of the atomics library interface is now available [2].

like to allow locking primitives that allow memory operations to be moved into, but not out of, critical sections, as in Java. A simple specification along the lines of the current Pthread spec would impose additional restrictions and cost.

[2]For example, many architectures provide an atomic exchange or compare-and-swap primitive that simultaneously enforces memory ordering. Itanium provides load and store instructions that ensure ordering similar to what we describe below.

1.2 Ordering constraints

Perhaps the most contentious aspect of the design of the atomics library has turned out to be the set of ordering constraints. This matters for several reasons:

1. Lock-free algorithms often require very limited and specific constraints on the order in which memory operations become visible to other threads. Even relatively limited constraints such as "release", may be too broad, and impose unneeded constraints on both the compiler and hardware.

2. Different processors often provide certain very limited constraints at small or no additional cost, where the cost to enforce something like an "acquire" constraint may be more major.[3] In particular, the vast majority of, but not all, processors ensure that dependent operations, e.g. loading a pointer, and then loading a value referenced by the pointer, occur in order, even when observed by other threads. But different processor architecture rarely, if ever, agree on what exactly constitutes dependent operations.

3. As discussed briefly below, adding further constraints appears to significantly complicate the memory model.

At the moment, there is no clear consensus within our group, but we are leaning towards a small set of ordering constraints (acquire, release, both, or none), largely because we have been unable to find a way to precisely specify the rest. In particular, the other implicit guarantees provided "for free" by many processor architectures are usually such that they can be easily negated by routine compiler transformations, sometimes even by transformations made to code in a separately translated source file. Thus it appears hard to take advantage of these without resorting to essentially assembly code.

We do expect to diverge somewhat from most primitves defined by [3] in allowing atomic operations themselves to be reordered with each other, if they have no explicit ordering constraint to the contrary. There is no assumption that atomic operations by default enforce sequential consistency among themselves, as volatiles do in Java.[4]

We expect that limiting the ordering constraints to a small set will result in small performance losses (on the order of a few percent) for some lock-free code relative to the assembler equivalent. But this should remain minor compared to the benefit that may be achieved by lock-free code in many circumstances.

2 Implications on the Memory Model

Our original plan was to largely preserve what might be termed the "pthreads memory model". Any code that contained a "data race", i.e. an assignment

[3] Note that this does not apply to current X86 processors and others, where even an "acquire" constraint imposes no additional cost.

[4] Our "release" semantics correspond roughly to `java.util.concurrent.atomic lazySet` semantics.

to a shared location that took place concurrently with another access to that location, would have undefined semantics. Any other code would have "sequentially consistent"[6] semantics, i.e. could be understood as a simple interleaving of the actions of the individual threads.

We concentrated on defining precisely what was meant by a "data race". This is currently only very incompletely defined, which gives rise to the issues discussed in [1]. This definition was itself based on sequential consistency.

This approach has the nice property that the programmer only needs to understand sequential consistency in order to reason about programs. At the same time, the actual guarantees that are provided are *much* weaker than sequential consistency, since concurrent writes are simply disallowed.

This approach unfortunately does not work in the presence of an atomic operations library, which explicitly allows concurrent accesses to special atomic variables, while providing few memory ordering guarantees. Consider having thread A execute the following, where initially `x`, `y` and `z` are all zero:[5]

```
atomic_store(&x, 1);
r1 = atomic_load(&y);
if (!r1) ++z;
```

while thread B executes:

```
atomic_store(&y, 1);
r2 = atomic_load(&x);
if (!r2) ++z;
```

Under a sequentially consistent interpretation, one of the `atomic_store` operations must execute first. Hence `r1` and `r2` cannot both be zero, and hence there is no data race involving `z`. There are data races involving `x` and `y`, but those accesses are made through the atomic operations library, and hence must be allowed. Atomic accesses are not meaningful if there is no data race.

The difficulty is that there are strong reasons to support variants of `atomic_store` and `atomic_load` that allow them to be reordered, i.e. that allow the `atomic_load` to become visible to other threads before the `atomic_store`. For example, preventing this reordering on some common X86 processors incurs a penalty of over 100 processor cycles in each thread. Both the ordinary load and store operations, as well as the acquire and release versions from the preceding section, will allow this reordering.

For variants that allow reordering, the above program should really invoke undefined semantics, since `r1` and `r2` can both be zero, and hence there is a data race on the ordinary variable accesses to `z`.

This means that we need to use a definition of data race that incorporates the reordering constraints imposed by atomic operations.

Our current (still very rough) draft proposal [4] requires sequential consistency for ordinary variable accesses, but effectively allows reordering of atomic

[5]For brevity of explanation, we use over-simplified syntax, not the one from our current draft proposal.

variable accesses *when determining the existence of a data race*, i.e. it tries to adapt the original solution. (The basic approach we use for atomics in the definition of a data race is similar to the notion of happens-before consistency[5].)

So far, this approach seems to have the right properties. It has the disadvantage that we are essentially exploring new ground, and hence have relatively little confidence that there are no unforeseen glitches.

3 Alternative: The Java Memory Model

An alternate approach to many of these issues would be to essentially adopt the Java memory model. This option is still discussed regularly, and has not been completely dismissed, particularly if we run into further technical difficulties with our current approach.

There are however good reasons that we have not committed to that route, in spite of the overlap among the participants in the two efforts. These were largely discussed in an earlier document (N1777=05-0037), and informally agreed to by the committee.

Both for completeness, and because they have been explored a bit deeper by our group in the meantime, we again summarize them here:

- The Java approach is required to ensure type-safety, and more particularly, to ensure that no load of a shared pointer can ever result in a pointer to an object that was constructed with a different type. C++ does not have this property to start with, and hence we are free to continue to violate it.
- This would be different from the approach taken by pthreads (and probably implicitly by other thread libraries) which give undefined semantics to any code involving a data race.
- The Java approach requires ordinary reads to be atomic for certain data types, including pointers. We expect this may be problematic on some embedded hardware with narrow memory buses or weak alignment constraints.
- The Java approach requires a complex model to deal with causality. We are hoping to make that unnecessary for the core language, and to specify the atomics library less formally. This is discussed in the next section.
- The current pthreads approach allows some compiler transformations that are not legal in Java, but probably moderately common in the context of C++. These produce unexpected results in the presence of data races. But given the pthreads approach of outlawing data races, they appear benign, and can improve performance. In particular, assuming we have no intervening synchronization actions, compilers currently may:

1. "Rematerialize" the contents of a spilled register by reloading its value from a global, if single-threaded analysis shows that the global may not have been modified in the interim. This is only detectable by programs that involve a data race. (This means that r1 == r1 may return false if r1 is a local integer loaded via a data race. That's OK since the program has undefined behavior.)
2. Fail to provide explicit guarantees about visibility of vtable pointer initialization. Current rules do not allow an object to be communicated to another thread without explicit synchronization. We expect that on some architectures, this avoids an expensive memory barrier.
3. Introduce redundant writes to shared globals. For example, if we are optimizing a large synchronization-free switch statement for space, all branches but one contain the assignment `x = 1;`, and the last one contains `x = 2;` we can, under suitable additional assumptions, put `x = 1;` before the switch statement, and reassign x in the one exceptional case.
4. Insert code to do the equivalent of `if (x != x)` <*start rogue-o-matic*> at any point for any global x that is already read without intervening synchronization. That's perhaps not very interesting. But it does point out that analysis in current compilers is based on the assumption that there are no concurrent modifications to shared variables. We really have no idea what the generated code will do if that assumption is violated. And the behavior cannot necessarily be understood by modeling reads of globals to return some sort of nondeterministic value.

None of these transformations are legal in Java. Variants of all of them appear potentially profitable and/or useful as part of a debugging facility.

4 Causality issues

We are hoping to largely avoid the rather complex treatment of causality which forms the basis of the Java memory model. It currently appears that we can do so for the core language, but that this is not entirely possible in the presence of the atomics library.

Consider a modification of the example from section 2. Again all variables are initially zero. Thread *A* executes:

```
r1 = atomic_load(&y);
atomic_store(&x, r1);  // Effectively assigns y to x
if (r1 == 42) ++z;
```

while thread *B* executes:

```
r2 = atomic_load(&x);
atomic_store(&y, r2);  // Effectively assigns x to y
if (r2 == 42) ++z;
```

This contains a data race, and thus has undefined semantics, if and only if `r1` = `r2` = 42. Most programmers would be surprised by this result. However, since the `atomic_load` and `atomic_store` operations performed by each thread may become visible to the other thread out of order, in the absence of some causality requirement this becomes possible. Each thread's `atomic_load` operation sees a value of 42, which it then stores in the other variable, justifying the result read by the other thread.

In fact, ignoring the atomicity requirements, there are sequentially correct compiler transformations that could produce this outcome. Me might have profiled this code under different circumstances and discovered that x and y are usually 42. The compiler may then have transformed what is effectively the assignment `x = y;` in thread A to

```
x = 42; if (y != 42) x = y;
```

(In isolation, this is clearly silly. In more complicated contexts, this might conceivably make sense for instruction scheduling reasons, e.g. because `y` is being prefetched, and this gives the prefetch more time to complete.)

We would like to prohibit this kind of transformation for atomics. (It is not an issue for ordinary variable assignments, since it can only arise in the presence of a data race, and that is effectively disallowed.)

Stating precisely what it means to disallow this would require something functionally similar to the Java treatment of causality[5], which we would like to avoid for reasons of simplicity.

Since we expect that this kind of aggressive transformation on calls to the atomics library is unlikely in any case, we instead propose to address it with a less precise statement, to the effect that speculative execution of primitives in the atomics library is disallowed.[6]

5 Conclusions

We believe that we are making progress on the specification of both a memory model and an atomic operations library for C++.

It has become increasingly clear that any specification will have an unavoidable impact on compiler optimization.[7] Some currently common compiler optimizations need to be adapted to ensure thread safety. But this also reinforces the urgency for thread support in C++: Current implementations make it much harder than it should be to write correct multithreaded code.

[6]This would probably not be acceptable for ordinary variables, since it gives rise to some borderline cases, which really need to be resolved. But we feel that in the case of atomics, it is perfectly acceptable to resolve any such questions conservatively, since any performance penalty should be negligible. In the very unlikely case that serious questions arises in this area with atomics, the Java spec could serve as a guide for resolving them.

[7]We have some new examples of this. See paper N1131 which is being submitted concurrently to WG14.

Our progress has been slowed by both the technical difficulties of defining a memory model that is compatible with a high performance atomics library, and by disagreements about the atomics library itself.

We believe that our current approach is based on sound reasoning, and that there is no easy way to sidestep the technical difficulties. It does still appear that we are heading for a solution whose complexity is effectively limited to the atomics library, and hence will not affect a programmer writing data-race-free programs that use locks for synchronization. We expect most programmers to fall into this category.

Feedback on our direction is greatly appreciated.

6 Acknowledgement

Peter Dimov, Alexander Terekhov, and others contributed significantly to the discussions that led up to this document, and to the documents that preceded this.

References

[1] Hans Boehm. Threads cannot be implented as a library. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 26–37, 2005.

[2] Boehm et al. A very preliminary proposal for the atomics package interface. `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/atomics.h.txt`.

[3] Doug Lea et al. Java Specification Request 166: Concurrency Utilities. Available at `http://www.jcp.org/jsr/detail/166.jsp`.

[4] C++ Memory Model Working Group. A memory model for c++: Strawman proposal. `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/mm.html`.

[5] JSR 133 Expert Group. Jsr-133: Java memory model and thread specification. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf`, August 2004.

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