Document Number: WG21/N1777=J16/05-0037
Date: 2005-03-04
Reply to: Hans Boehm
Hans.Boehm@hp.com
1501 Page Mill Rd., MS 1138
Palo Alto CA 94304 USA

Memory model for multithreaded C++: Issues

Andrei Alexandrescu Hans Boehm Kevlin Henney
Ben Hutchings 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 propose integrating a memory model suitable for multithreaded execution into the C++ Standard. This document is a continuation of N1680=04-0120, and outlines some of the more fundamental issues we have encountered. In particular, we desire initial feedback on whether we should prepare for a future type-safe subset of the language, or ignore type-safety considerations for now.

1 Introduction

Many of today's applications make use of multithreaded execution. We expect such use to grow as the increased use of hardware multithreading (a.k.a. "hyperthreading") and multi-core processors will force or entice more and more applications to become multithreaded. C++ is commonly used as part of multithreaded applications, either with direct calls into an OS-provided threading library (e.g. POSIX threads (pthreads) [5] or Win32 threads) or with the aid of an intervening layer that provides a platform-neutral interface (e.g. Boost Threads).

In N1680=04-0120 and in [3] we point out the difficulties with this approach and the need to clearly define a memory model, analogously to [4], which provides sufficient guarantees to the programmer to guard against unexpected transformations that are benign in the absence of threads but change the semantics of a concurrent program.

Here we list some fundamental issues on which we seek preliminary guidance. The first of these appears sufficiently fundamental that our group cannot make much progress without resolving it in one direction or the other. We suggest resolutions where there appears to be consensus within our group.

2 Do we define the semantics of Data Races?

A *data race* arises when one thread in a program can potentially write a shared memory location while another concurrently accesses it. Constructing a program that exhibits data races and is guaranteed to work correctly in spite of reorderings allowed by the memory system and by the compiler is very difficult, and data races are usually an indication of a programming error.[1]

Most existing multithreaded C++ applications operate in an environment in which the semantics of data races are left intentionally undefined. For example, the POSIX threads (pthreads) [5] standard takes this route. If the semantics of data races are left undefined, then the compiler can assume that within a synchronization-free region (a section of a program in which there are no synchronization, volatile or barrier operations):

- if a thread reads a memory location, no other thread can be modifying that location,
- if a thread writes a memory location, no other thread can be reading that location, and
- if a thread constructs an object, the object doesn't have to be made ready for access by other threads until immediately before the first synchronization operation after the object creation (e.g., the vtable doesn't need to be constructed until immediately before the first synchronization operation after the object is constructed).

(The meaning of "memory location" is not precisely defined.)

The compiler is allowed to make transformations that produce complete program failure, execute arbitrary code or result in other unexpected behavior if these rules are violated.

For example, in many cases it is acceptable to compile:

```
y = x; use y; ...; use y;
```

to

```
load x,r1; use r1; ...; load x,r2; use r2;
```

If the compiler runs out of registers in which to store `y`, it may reload the variable from its original shared source `x`, provided `x` cannot be written by any intervening statement and there are no intervening synchronization operations. This would mean, for example, that the program:

```
int i = x;
int j = i;
assert(i == j);
```

[1] In a few cases, concurrent accesses to shared variables without locking can lead to much faster programs. We plan to provide alternate mechanisms intended exclusively for such concurrent accesses in a way that is recognizable by the implementation. They are excluded from our definition of a data race, as are volatile accesses in Java.

can fail, since since the compiler is allowed to load `x` twice, which could return different results if `x` is being modified via a data race by another thread.

On the other side, the Java programming language carefully defines the semantics of programs with data races, and in this case outlaws this transformation, forcing `r1` to be spilled to a new location in the above example.

The Java specification is essentially forced to make this choice, since it tries to guard against malicious code running in the same address space. If the above sequence occurred in trusted code, the first use of `r1` were a safety check (raising an exception in the unsafe case), and the second use of `r1` were safe only when the check succeeded, it would be unsafe to reload `r1` from `x` in the interim. Malicious code could repeatedly call this function while changing `x`, eventually causing the trusted code to perform an unsafe action on a value of `x` different from the one it tested.

In Java, this transformation could also result in an unchecked out-of-bounds array access, and hence violate type-safety if, for example, the pointer to a shared array is reloaded between the subscript check and array access.

Some C/C++ compilers appear to spill to new locations anyway, and others might decide to now do so to avoid programmer surprises, so this particular issue might not be decisive. But there are potentially many additional cases along these lines (for example, caching statics). The need to provide a full semantics for data races in all such cases in the Java specification introduces some of the more challenging technical issues in [7], some of which we are not confident can be applied to C++, but which can be avoided if we leave their semantics undefined.

Some other implications of not defining the semantics for data races:

- Compilers would be allowed to perform tricks such as XORing pointer values, which could cause other threads reading such pointers to see illegal values.

- Compilers could perform writes non-atomically (e.g., writing a pointer as a sequence of two 16-bit writes), which could cause other threads to see corrupted pointer values.

- Even if a variable is only read once within a synchronization-free region, the compiler can transform the program to read that variable multiple times in ways that cause program faults if it sees different values.

- If a thread constructs an object and stores a reference to that object into shared memory without synchronization, then other threads that read the stored referance and invoke virtual member functions on that object are allowed to result in a segmentation fault or other erroneous behavior.

The consensus within our group (grudgingly for some of us) is that it would be preferable to leave the semantics of data races undefined, mostly because it is much more consistent with the current C++ specification, practice, and especially implementations. In the absence of objections from the committee, we plan to follow that path.

The disadvantage of this approach is that any future attempt to define a type-safe subset of C++, especially if it intends to support “sand-boxed” execution of untrusted code, would have to revisit this issue, and follow a more Java-like model.

3 Other Issues

We list here a number of other issues on which we would appreciate feedback. They are probably less time-critical. We also expect that we have overlooked some equally fundamental and controversial ones. We appreciate additions to this list.

3.1 Implicit writes restricted to bit-fields

Essentially all of the problems pointed to by our prior work are due to compiler transformations that introduce stores, and hence data-races, that did not exist in the original program.

A store to a bit-field requires reading and rewriting adjacent bit-fields of the same struct or class, since most architectures do not support storing, say, a single bit to memory. Some older architectures had similar restrictions for `char`-sized values, but we are not aware of current multiprocessors with such restrictions.

We are leaning towards allowing such compiler-introduced writes only for contiguous sequences of bit-fields in the same struct or class. (The detailed formulation would require more care.) Some restriction along these lines is required for reliable multithreaded programming. Weaker restrictions might be possible at some usability cost, but we currently see no reason to weaken the restriction.

This would greatly penalize multiprocessor architectures which do not support efficient atomic byte stores. However we are not aware of any such current architectures. It would force uniprocessors with such restrictions to use techniques such as those in [2].

This affects not only the way values are stored into a structure, but also some other compiler transformations, notably fully general speculative register promotion [8, 6]. However we see no way to simultaneously support this optimization and reliable concurrent programming. Further, we believe that not all high performance C++ compilers currently perform this transformation, so the cost of disallowing it is not huge. Java disallows all implicit writes, including those introduced by speculative register promotion.

3.2 The meaning of “volatile”

We are leaning towards strengthening the meaning of “volatile” to make it usable for (relatively) inexpensive inter-thread communication. As discussed in the original proposal N1680=04-0120, we also plan to define intrinsic library classes that provide more flexible and extensive control over barriers and atomic instructions.

The main attraction of this approach to `volatile` is that it makes it more likely for programmers to write correct code, by following the rule that any variable that is used for inter-thread communication without a lock should be declared `volatile`. (There are cases where this is unnecessary, but they require thought and care.) With the semantics of "volatile" used by Java, double-checked-locking [1], Dekker's algorithm, and other common constructions "just work" if the relevant variables are declared `volatile`.

However, there are a few concerns that make this choice controversial:

First, this sense of volatile requires read/write atomicity. This can be a problem for types wider than natively supported on a processor. On uniprocessors, this may entail some minor overhead to support "restartable atomic sequences" to keep together multiple read or write instructions in the face of interrupts or context switches. However, on some multiprocessors, there might not be any applicable techniques short of heavy solutions such as the insertion of otherwise inaccessible locks. Thus, there may be types for which the `volatile` qualifier either cannot be supported on a particular target platform, or would entail surprising time and space overhead. It's not clear to us how such limitations can be expressed in the specification, but we suspect that there is some way to do so. Guidance on this issue would be appreciated.

Second, these semantics for `volatile` impact performance, especially in the absence of new compiler optimizations. On multiprocessors, naive translation of Java `volatile`s requires an expensive "memory barrier" to be issued after each volatile assignment, in addition to an often much less expensive barrier preceding the store instruction. The current C++ `volatile` semantics on Itanium represent a slightly weaker variant that avoids this barrier and still seems to handle most practically important cases, but not Dekker's algorithm. And it may be more difficult to specify at the programming language level.

Third, it is possible that these semantics might conflict with some of the current implementation-defined effects of `volatile` in certain compilers. This does not appear likely though.

We believe that the advantages of this general approach outweigh the disadvantages, but would like to hear especially from compiler implementors about any additional concerns or constraints.

3.3 Function-scope statics

Construction of function-scope statics may require the compiler to introduce an "is this initialized" flag variable. In a multithreaded program access to this flag variable may introduce a data race not visible to the programmer. There appear to be three possible solutions:

1. Document the existence of the flag, and let the programmer add suitable synchronization code.

2. Let the compiler add the synchronization code.

3. Deprecate the construct, except for initializations to compile-time constants.

The first option may be very surprising to the programmer. The second option adds (potentially expensive, if done correctly) memory barriers, which we expect are usually redundant with programmer-provided synchronization. Thus it may surprise the programmer with unexpected performance problems. It also may cause the compiler to generate thread-library-dependent code, or it requires a standard API across thread libraries.

Based on a very limited sample, current practice seems to favor the second option. Our group currently seems to lean towards the third, though it does appear drastic.

We expect this is an issue that was previously debated by many compiler implementors. We would like to better understand the various outcomes. We would also like to know about the prevalence of such constructions in practice. If they are rare, a heavyweight solution may work out fine.

3.4 Atomic read-modify-write support

As described in the earlier document, we plan to add library access to the atomic read-modify-write operations provided by most modern hardware. The difficulty is that such operations are not implemented by all hardware, and even if they are implemented, operations such as *compare_and_swap* will be implemented for differently sized data on different hardware. Software emulations of these are only partially satisfactory.

We are discussing to what extent we should rely on emulations here, as opposed to directly exposing the hardware differences to the programmer. (Hybrids will clearly also be considered.)

It would be useful to be aware of any hardware that is likely to remain critical in the future, and provides a significantly different set of atomic update operations from the mainstream desktop or server processors that our group is familiar with.

3.5 The threading API

Our group has had interesting internal discussions about the desirability of defining a C++-standard threading API. Current practice appears to be split between many different APIs or language extensions, often with significantly different programming models. (Pthreads, Win32 threads, OpenMP, and Boost threads are probably among the most widely used, with many other very significant contenders.)

Real standardization would almost certainly be of benefit here, but since many of these are already widely established, it is unclear whether that is achievable, and there seems to be relatively little consensus as to what it should look like.

We plan to separate the memory model discussions from the threading API discussions as much as possible. It appears to be possible to define the memory model without reference to the details of the threading API, and in a way which applies across all of the above models.

If there is any consensus on the C++ committee as to whether we should pursue a threading API, or what it should include, we would like to learn about it.

References

[1] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Hahr, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "Double-Checked Locking Pattern is Broken" Declaration. Available at `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

[2] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS-V: Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.

[3] Hans Boehm. Threads cannot be implented as a library. `http://www.hpl.hp.com/techreports/2004/HPL-2004-209.html`.

[4] Tim Lindholm et al. Java Specification Request 133: Memory Model and Thread Specification Revision. Available at `http://www.jcp.org/jsr/detail/133.jsp`.

[5] IEEE Standard for Information Technology. *Portable Operating System Interface (POSIX) — System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. ANSI/IEEE 1003.1c-1995, 1995.

[6] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.

[7] Jeremy Manson, William Pugh, and Sarita Adve. The java memory model. In *Conference Record of the Thirty-Second Annual ACM Symposium on Principles of Programming Languages*, January 2005.

[8] A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on ssa form. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 15–25, 1998.