ISSN:0975-9646

An Abstract memory model describing the interaction between thread and memory with debugger tools

Shruti Sandal[#1], Dr.Raghuraj Singh[*2], Abdul Jabbar Khilji[#3], Shashi Shekhar Ranga[#4], sanjay tejasvee[#5]

#Assistant professor

Department Of Computer Application

Engineering College Bikaner

**Head of Department of Computer Science & Engineering Department,*

H.B.T.I., Kanpur

#1shrutisandal@yahoo.com,#3khiljisania746@gmail.com,#4ranga.ssr@gmail.com,
#5sanjaytejasvee@gmail.com

**raghurajsingh@rediffmail.com*

***Abstract:*-This paper describe the multithreaded execution and data race detectors which are commonly viewed as debugging tools.The C++ Standard defines single-threaded program execution. Basically, multithreaded execution requires a much more refined memory and execution model. C++ threading libraries are in the awkward situation of specifying an extended memory model for C++ in order to specify program execution. We suggest integrating a memory model suitable for multithreaded execution in the C++ Standard. We wants to make fast and error free program .but ideally it is not possible To overcome this problem we give first concept threading and ssecond concept in this paper is data race detector. They would allow us to give precise, simple, and safe semantics to shared variables in multithreaded programs, a problem that has so far defied a complete solution.**

***Keyword*:-atomicity,data race.**

I INTRODUCTION

multithreaded execution. is use in most of today's programming. C++ is commonly used as part of multithreaded applications, sometimes with either direct calls into an OS-provided threading library or with the aid of an intervening layer that provides a platform-neutral interface. Properties critical for reliable, efficient, and correct multithreaded execution are left unspecified The C++ Standard specifies program execution in terms of observable behavior, which in turn describes sequential execution on an implicitly single-threaded abstract machine. The main sketch of attack is:

1. Specification of an abstract memory model describing the interactions between threads and memory.
2. Application of this model to existing aspects of the C++ specification to replace the current implicitly sequential semantics. This will entail new constraints on how compilers can emit and optimize code. In particular, this will entail a reworking of the specification of volatile to provide useful multithreaded semantics.
3. Introduction of a small number of standard library classes providing standardized access to atomic update operations (such as compare_and_set). These classes will have multithreaded semantics integrated with the above specifications for other memory operations. Thus, compilers will need to treat these as intrinsic. These operations form the low-level basis for modern multithreaded synchronization constructs such as locks, and are also required in the construction of efficient non-blocking data structures.
4. Definition of a standard thread library that provides similar functionality to threads and Win32 threads, but meshes with the rest of the C++ standard.

Secondly Data races are well-recognized as a common source of particularly difficult-to-diagnose bugs in parallel programs. As a result many tools have been built to explicitly detect data races, either at compile time, or as the program is executing).Although code correctness typically requires stronger properties, such as atomicity[7] or even determinacy, data-race-freedom remains interesting since it is a well-defined condition that is easy to check, even in the absence of any additional programmer supplied specifications. This is particularly true for a number of important language specifications, notably the expected upcoming revisions of the C and C++ language standards [14, 11, 15] and the much earlier Posix threads [10] and Ada [17] standards, that explicitly treat all data races as programmer errors [1]. In these languages, an accurate (no false positives) data race detector, such as [6] or [8], by definition diagnoses only actual errors

II WHY DATA RACE IS VIEWED AS DEBUGGER TOOL

There are two additional reasons we would really like to see an accurate mechanism for detecting and avoiding data races, e.g. by throwing an exception as in [6]:

1. Data-race-free programs are independent of the granularity at which memory accesses are performed. They exhibit the same behavior on a machine that accesses memory a byte-at-a-time as it does if memory is accessed 64 bits at a time. Similarly, accesses to library or user-defined synchronization-free data structures behave atomically. In both cases, a half-updated data structure can't be observed by another thread, since the observer thread would introduce a data race.Note that this property is orthogonal to sequential consistency.

2. It has proven to be very difficult to define the meaning of programs with data races in a way that both disallows behavior that can result in blatant security holes, and allows simple meaning preserving
compiler transformations on source programs.

We can assurance such ordering by avoiding data races. There are two common approaches to doing so:

1. Data races result in "undefined behavior". This solves the problem in C++0x [4, 11], except in the presence of the previously mentioned esoteric library calls.
2. Statically enforce the absence of data races.

III THE BEHAVIOR OF THERADS TOWARDS BASIC MEMORY OPERATIONS

A memory model explore the behavior of threads with respect to basic memory operations – mainly reads and writes of variables potentially accessible across multiple threads. The main questions are raised by a memory model nclude as follow:

Atomicity: Which memory operations have indivisible effects?

Visibility: Under what conditions will the effects of a write action by one thread be seen by a read by another thread?

Ordering: Under what conditions are sequences of memory operations by one or more threads guaranteed to be visible in the same order by other threads?

As such, the process of defining a sound memory model for C++ can reuse the years-long effort that was invested in defining, peer-reviewing, refining, and debugging the mentioned formal model. However all above questions are related to hardware but still work is done on it.In other word ,work on progress.

IV MAPING OF MEMORY MODLE

There are various kind of memory actions. and after that next for a language specification is to map these notions to all of the memory-related constructions in the language. This process entails nailing down a large set of "small issues" that are necessary for programmers to be able to predict and control effects. Areas that we have so far identified include:

Atomicity A given platform may guarantee atomicity only for reads and writes of certain bit widths and alignments. The spec must permit these to vary, and must therefore provide some means for programs to query these properties.

Extra writes There are several cases in C++ in which compilers and machines have historically been permitted to issue writes that are not obvious from inspection of source code. The most notable examples involve structures with small fields. For example, given:

```
struct S { short a; char b; char c; } s;
```

an assignment such as s.a = 0 might be executed as if the code were *(int*)&s = 0 if a compiler infers from context that b and c are zero as well, as in the following example:

```
void Fun(S& s) {
if (s.b == 0 && s.c == 0) {
s.a = 0;
}
}
```

It is not expected at best in a multithreaded context in which the other fields were also being assigned concurrently. A spec must clearly define whether and when such compiler transformations remain legal.

Volatile data In the current language spec, the volatile qualifier is mainly used to indicate guaranteed order of reads and writes within single threaded semantics—for example for device control registers, memory mapped I/O, or opaque flow (as in setjmp or interrupts). In a multithreaded language, it may be useful for volatile to take on the extra burden of constraining inter-thread visibility and ordering properties. There are a few options for the detailed semantics. In the simplest, volatile reads act as acquire and writes as release. This has the virtue of being
relatively easy to use by programmers who are not intimately familiar with memory models. For example, the infamous "double-checked locking" idiom [3] works as expected under these rules if references are declared as volatile (and other lock-based rules below are followed). This has the disadvantage of imposing "heavier" constraints on the compiler and processor than necessary in very performance-sensitive applications. However, optimizers can often eliminate unnecessary operations (such as consolidating several consecutive acquire and release operations into
one).

Opaque calls One anxiety about moving to multithreaded specifications is that compilers may become excessively conservative when compiling code with opaque function calls—flushing and reloading registers and/or issuing memory barriers in case the called function's effects depend on this. It may be desirable to allow programmers to control this using some kind of qualifier. Options include those with defaults in both directions; for illustration, assuming lack of effects unless a function is qualified as, say, mutable; versus assuming effects unless qualified with some extended form of const. Alternatively, or in addition, the spec could include a means for programmers to tell compilers that a certain program is either definitely single-threaded or definitely multithreaded, as a way of controlling certain
optimizations. Further exploration of options and their consequences is needed. These considerations are very related to an existing C++ standardization proposal [2].

V MULTITHERADING SYNCHRONIZATION AND COORDINATION

Atomic update operations (since linked memory barrier instructions, which impose memory ordering constraints on the processor) form the basis for fundamentally all modern multithreaded synchronization and coordination. While there is some variety across architectures in the nature and style of these instructions, there is adequate commonality in current and medium-term-future systems to define a small set of intrinsic that can be used for moveable concurrent programming. There are several stylistic options here. One approach is to define three small intrinsified inclinable classes, one each holding a single value of type int, long, and (templated) pointer, and supporting operations such as:

```
namespace std {
```

```
class atomic_int {
public:
int get();
int set(int v);
bool compare_and_set(int expected_value, int new_value);
int weak_get();
int weak_set(int v);
bool weak_compare_and_set(int expected_value,
int new_value);
// other minor convenience functions, including:
int get_and_increment();
int get_and_add(int v);
// ...
};
}
```

The major magnetism of this approach is that it appears to be implement able on essentially any platform. Even those machines without such primitives can emulate them using private locks. And even though some machines (such as PowerPC) support LL/SC (load-linked, store-conditional) instead of CAS (compare-and-set), in live out, nearly all usages of LL/SC are to perform CAS (the reverse is impossible), so there would rarely be incentive to resort to non-standardized, non-portable constructions even on these platforms. The thought of the "weak" versions is to permit finer control of atomics and barriers than otherwise available using volatile or other constructions. For example, a weak set need only perform a store ordering barrier, not a full release, which may be cheaper on some machines.

VI LIBRARIES FOR THERADING SUPPORT

A this time, multithreaded C++ programs tend to rely first and foremost on one of a fairly small set of libraries for threading support: POSIX threads, Win32, ACE, and Boost. These hold many more similarities than differences. The chance arises to provide a standard library that unadventurously abstracts over such packages. Even if this is not done, such libraries must, to conform to the rest of this proposal, spell out their basic locking primitives in terms of the memory model. All basic locks should and do make available semantics in accord with the basic gain and release actions specified by the Standard. Compilers in turn must respect these semantics. The technicalities to ensure this would rely on how the opaque call issue mentioned above is resolved. In this draft we do not even sketch out the APIs of this library.

VII. THE C++ MODLE WITHOUT LOW LEVEL ATOMICS

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 The remainder of the C++ standard was modified to define a sequenced-before relation on memory operations performed by a single thread [5]. 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 aset 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. Otherwise, the program (on the same input) behaves according to one if its sequentially consistent executions.

VIII SEMANTICS OF DATA RACES

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 [9] 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 [10] 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 [12] "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 implement able, 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.

IX CONSULATION

In this paper ,we give the outline about the foundation of multithreading I and data race. We try to represent the memory model for multithreading and also map it .Data race is debugger tool in this respect of threading . 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 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 [13].

REFERENCES

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

[2] Walter E. Brown et al. Toward Improved Optimization Opportunities in C++0X. DocumentWG21/N1664 = J16/04-0104; available at http://www. open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1664.pdf, Jul 2004

[3] Scott Meyers and Andrei Alexander's. C++ and The Perils of Double-Checked Locking. Doctor Dobb's Journal, Jul 2004.

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

[5] 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.

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pages 245–255, 2007.

[7] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. Science of Computer Programming, 71:89–109, 2008.

[8] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, 2009.

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

[10] 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.

[11] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, programming language - C++ (committee draft). http://www.open-std.org/jtc1/ sc22/wg21/docs/papers/2008/n2800.pdf, 2008.

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

[13] 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.

[14] 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.

[15] C. Nelson, H.-J. Boehm, and L. Crowl. Parallel memory sequencing model proposal. C standards committee paper WG14/N1349, http: //www.open- std.org/JTC1/sc22/wg14/www/docs/n1349. htm, February 2009.

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

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

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

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

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

[21] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overby, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. Technical Report UIUCDCS-R-2009-3032, UIUC, 2009.

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

[23] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. A case for system support for concurrency exceptions. In HotPar, 2009.

[24] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pages 245–255, 2007.

[25] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. Science of Computer Programming, 71:89–109, 2008.
[26] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, 2009.
[27] J. Gosling, B. Joy, G. Steele, and G. Bracha. Java Language Specification, 3rd edition. Addison Wesley, 2005. [28] IEEE and The Open Group. IEEE Standard 1003.1-2001. IEEE, 2001.
[29] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, programming language - C++ (committee draft). http://www.open-std.org/jtc1/ sc22/wg21/docs/papers/2008/n2800.pdf, 2008.
[30] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C-28(9):690–691, 1979.
[31] J. Manson, W. Pugh, and S. Adve. The Java memory model. In Proc. Symp. on Principles of Programming Languages, 2005
[32] 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.
[33] C. Nelson, H.-J. Boehm, and L. Crowl. Parallel memory sequencing model proposal. C standards committee paper WG14/N1349, http: //www.open-std.org/JTC1/sc22/wg14/www/docs/n1349. htm, February 2009.
[34] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In PPoPP 07, March 2007.
[35] United States Department of Defense. Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001, 1983. Springer.2