Research on and Comparison of the Java and C++ Memory Models

Nathan C. Padoin
Supervisor: Dr Julian Rathke
Department of Electronics and Computer Science
University of Southampton
ncp1g09@ecs.soton.ac.uk

*Abstract–***This document seeks to examine the memory models of Java and C++. Java's, which was specified before the concept was fully understood, and C++'s which had the concept added to a language which had been written and used for years under the assumption that there wasn't one. This paper then goes on to compare the two models, in their specification, implementation and intended use.**

*Index Terms–***C++, Concurrency, Java, Memory Model, Multithreading**

I. Introduction

In the current realm of computing the dominant processing paradigm is that of a constantly increasing core count. This is because computer processors have increasingly hit a wall of limiting per-processor performance. As such, programming paradigms have therefore been moving in the direction of explicit multithreading in order to take advantage of these parallel architectures.

In doing so, developers and programmers must make use of either a programming language with threads as a core feature, or else a language with a threading library implemented elsewhere in order to achieve an explicitly multithreaded program.

In doing so, some measure of synchronisation is required in order to ensure that the semantics of the multithreaded program are correct; else there may be concurrent variable accesses and stores

Modern compilers and processors can reorder the instructions in the written source code in order to achieve faster program speeds, and while this is acceptable in a single threaded system as long as the semantics of the resulting machine code are the same, these assumptions fall apart is in a multithreaded system where this instruction reordering can cause a thread to observe another thread performing the actions out of order. This effect can cause subtle behaviour modification which can change the semantics of the program.

In order to alleviate this, we can use a memory model to enforce orders on operations in the source code thus preserving sequential semantics. I.E., the memory model defines the behaviours which are legal in correctly synchronised (see also) legal code, and how threads may interact through memory. As such, another problem encountered with modern processors is that of cache, as a variable write may be missed due to cache, and also compiler optimisations, where since a shared variable isn't written to by a particular thread, a compiler might optimise the variable out, but the variable is changed from another thread, thus program semantics are again changed. Example code where this might happen is shown in Figure 3, where thread 1 might have x optimised to true, as x is only changed in thread 2.

```
  Before             After
LD (r31,X,r1)    LD (r31,X,r1)
LD (r31,Y,r2)    LD (r31,Y,r2)
ADD(r1,r2,r3)    LD (r31,Z,r5)
MUL(r3,r1,r4)    LD (r31,W,r6)
ST (r4,P,r31)    ADD(r1,r2,r3)
LD (r31,Z,r5)    MUL(r3,r1,r4)
LD (r31,W,r6)    SUB(r5,r6,r7)
SUB(r5,r6,r7)    ST (r4,P,r31)
ST (r7,Q,r31)    ADD(r4,r7,r8)
ADD(r4,r7,r8)    ST (r7,Q,r31)
ST (r8,T,r31)    ST (r8,T,r31)
```

Figure 1 - Code Reordering

```
x=0;y=0;
if (x == 1) ++y;
if (y == 1) ++x;
can be transformed to:
x=0;y=0;
++y; if (x != 1) --y;
++x; if (y != 1) --x;
```

Figure 2 - Code Transformation

Other reasons for a memory model – processors have also experienced a large increase in cache and available memory. Under a strong processor memory model, all processors and cores will see the same values in the same variables at all times. However, most modern processor architectures have weaker memory models which require special operations to cause cache coherence. This is because each processor/core has its own cache, so not only does internal cache have to be taken into account, but differences between caches in the system.

A memory model also allows semantics of a correctly synchronised program to be fixed with regards to the source code, and as such, operations within a program can be reordered by both a compiler and by the processor itself.

In the field of memory models, the definition of correctly synchronised is that there are no variables for which a write in one thread and a read in another thread are not ordered by synchronisation. If this is violated, this is defined as a **data race** and the semantics of such a program are undefined if a programming language level memory model is in effect. With a weaker memory model come memory barriers, also known as memory fences, in which when a memory barrier is passed, all processors will see the same variable in memory.

This paper shall now define several terms common to the field of memory models, as they form the core of the topic:

- ***Code Reordering*** – Code reordering refers to the process of switching the order of complied assembly instructions in order to increase the efficiency of program execution. Figure 1 shows the common idiom of pushing loads early and stores late (to prevent pipeline stalling), using common platform independent assembly syntax
- ***Code Transformation*** – Code transformation is the process of changing the program, as written, to another, semantically equivalent program. Figure 2, taken from [1], shows a transformation, using simplified C syntax, which keeps semantics in a single threaded context, but does not if another thread changes one of the variables.
- ***Program Order*** – The order of memory operations, as written in the source. Necessary to define the semantics of a program, as under this relation, all operations in the program are completely ordered, with no ambiguities.
- ***Sequenced Before Order*** – The sequenced-before ordering of a program is the intra-thread ordering roughly corresponding to program order. Except where the language specification does not specify an order.
- ***Total Order*** – The total order ($<_T$) is the final ordering, comprising all threads, of a program written under the constraints of sequenced-before. With the total order defined, a compiler is then able to tell how code can be transformed and reordered without violating sequential consistency or synchronisation rules.
- ***Happens-Before Order*** ($<_{HB}$) – A weaker ordering of a program, which only orders synchronisation effects with respects to loads and stores. A more full definition as used by Java and in some ways by C++ is given later.
- ***Data Race –*** A data race is typically defined as unordered accesses on the same variable. However, other definitions can be given, or to give semantics on what it means to avoid them, and, if the language demands it, any semantics of the 'flawed' program that apply.
- ***Correctly Synchronised –*** Each memory model has a definition of a program which correctly utilises memory barriers, the program order and any other ordering to prevent data races from occurring in a program. This concept is known as correct synchronisation and this definition is central to the behaviour of a memory model.
- ***Sequentially Correct*** – The notion of sequential correctness refers to the fact that in a sequentially correct program, the semantics of a total ordering can be derived as a direct interleaving of given memory operations in the program order [22].

```
thread 1                  thread 2
x = true;                 x=false;
while(x) foo();
```

Figure 3 – Thread Signalling

```
Thread 1                  Thread 2
volatile int x = 0        int y = 0
while (x == 0) {}         y=42
print "y is 42"           x=1
```

Figure 4 – Undermining Volatile

This paper will now describe a short history of the Java and C++ memory models, their current specifications, and any likely short-term developments.

II. The Java Memory Model

A. History

Java was the first major programming language to attempt implementation of a memory model across all processor architectures which supported the JVM (Java Virtual Machine) ***(citation needed),*** but as usage of the language progressed and the language was extended it has been found that the implementation of the Java Memory Model was not sufficient or intuitive, and as such a document with the changes was drafted and presented.

As with many compilers, code can be reordered in order to improve speed of execution. In Java, this can be done by the bytecode compiler, the JVM JIT (just in time) recompiler (which compiles bytecode to assembly code on-the-fly) and finally by processor that the JVM is running on. This reordering without changing single-threaded semantics is described as as-if-serial, where the result is the same as the serialisation of the written source code.

The initial Java memory model had several problems, with many parts working in a non-intuitive manner, and certain implementations of the JVM used code reordering which ran contrary to the specifications. An example of this lies in final fields. Under the old model, final fields were not treated differently from other fields, in terms of reordering or synchronisation. So a variable being marked as final only meant that the value could not be directly changed after initialisation, and not that the variable would always be seen as having that value.

This means that under certain circumstances, a final value could be perceived to change including immutable variables (an immutable variable is a final variable, or an object whose fields are all final) such as Strings. Volatile reads/writes could also be reordered with non-volatile operations of the same, as can be seen in Figure 4. In this example (using simplified Java syntax), after x has been set to be non-zero, it is expected that y is 42, as x is volatile and that is how the program has been written. However, since the compiler can reorder the statements in thread 2, y can legally be not 42, undermining the validity of volatile as a thread signalling mechanism.

Final objects being seen as changing their values can be seen under the old memory model, by using a possible representation of Strings in Java. A string can be implemented as a character array (`char[]`), an offset and a length, all declared final. The length and offset are used to make String and StringBuffer objects share the same class for efficiency.

Since operations on final fields can be reordered, a string with offset 4, length 4 and character array value "/usr/var", has the intended value "/var". However, it can be seen before being fully initialised. So if this implementation of String used offset 0 as its default value, if the length has already been set at value 4, the value of the overall String is "/usr". This can then cause successive reads to give that values "/usr", followed by "/var", once the String has been fully constructed. Thus, an immutable object (which by definition is final), has been seen to have changed its value.

B. JSR133 and Fixing the Old Model

JSR133 [2, 3] proposed changes to the memory model which were implemented as part of Java 5. It was written due to mistakes in the original specifications and implementations of the Java synchronisation keywords, consisting of final, volatile and synchronised. The primary goal of JSR133 being for these keywords to work in both a correct and specifically intuitive manner. Following is a shortlist of goals taken from [3]:

- Preserving existing safety guarantees, like type-safety, and strengthening others. For example, variable values may not be created "out of thin air": each value for a variable observed by some thread must be a value that can reasonably be placed there by some thread.
- The semantics of correctly synchronized programs should be as simple and intuitive as possible.
- The semantics of incompletely or incorrectly synchronized programs should be defined so that potential security hazards are minimized.
- Programmers should be able to reason confidently about how multithreaded programs interact with memory.
- It should be possible to design correct, high performance JVM implementations across a wide range of popular hardware architectures.
- A new guarantee of *initialization safety* should be provided. If an object is properly constructed (which means that references to it do not escape during construction), then all threads which see a reference to that object will also see the values for its final fields that were set in the constructor, without the need for synchronization.
- There should be minimal impact on existing code.

In order to achieve these it is important to define the core of memory models and synchronisation; the formal definition of a **data race**. In JSR133, and thus the new Java memory model, a data race is where, for a given variable, there is a write in one thread, a read in another thread and there is no synchronising order between them.

This lack of synchronisation and resulting data race is what causes inconsistent program behaviour. The basics of Java synchronisation show us that using synchronisation keywords prevents certain reorderings from taking place. As with most multithreading systems, Java relies on a mutual-exclusion paradigm, henceforth referred to as a **mutex**. Releasing a mutex lock causes cache coherence to be updated, and all things which are written in the program order before releasing the lock are definitely done before the lock is released. In processor terms, a **memory fence** (also known as a memory barrier) is employed.

When discussing these effects in terms of caches, it may sound as if these concepts only affect multiprocessor machines. However, the reordering effects can be easily seen on a single processor [find example]. It is not possible, for example, for the compiler to reorder operations around the mutex operations **lock** and **unlock** (also called acquire and release, respectively), such that code in a lock-unlock block cannot be moved before the lock or after the unlock. The implication that code can be reordered *into* the block is discussed later.

When it is said that acquires and releases act on caches, this is using shorthand for a number of possible effects.[3] The overall effect on this, even in systems which are not multiprocessor, are that of defining a partial order on the program, called the 'happens-before' ordering based on the fact that synchronisation causes operations to 'happen before' other operations. [3]

- Each action in a thread happens before every action in that thread that comes later in the program's order.
- An unlock on a monitor happens before every subsequent lock on **that same** monitor.
- A write to a volatile field happens before every subsequent read of **that same** volatile.
- A call to `start()` on a thread happens before any actions in the started thread.
- All actions in a thread happen before any other thread successfully returns from a `join()` on that thread.

This ordering is on the basic memory operations read, write, lock and unlock, and thread operations start and join (join is when child threads terminate and 'rejoin' their parent thread). This means that any memory operations which were visible to a thread before exiting a synchronized block are visible to any thread after it enters a synchronized block protected by the same monitor, since all the memory operations happen before the release, and the release happens before the acquire.

An implication of this is that one cannot enforce a synchronisation event with the code: `synchronized (new Object()) {}`, as this is semantically a no-op and is therefore compiled out, as no other thread attempts to do anything on that lock. The operations (release and acquire) must happen to set up a happens-before order, as it must be the same monitor, else data races ensue.

During object construction, the new memory model specifies that final and static values will never be seen to change, assuming that no references to that object 'escape' during the constructor (by passing a reference to 'this' outside of 'this') [15]. As the ability to create final references to objects and arrays is also desired, then a reference which is declared final will construct the correct values into the array or object before the end of the constructor.

Although, in order to get this behaviour, once again no references to the constructing object are allowed to escape the constructor. Also, ‘correct’ here means ‘up to date as of the end of the object's constructor’, and not ‘the latest value available’.

After all of this, though, after a thread constructs an immutable object (that is, an object that only contains final fields), in order to ensure that it is seen correctly by all of the other threads, synchronization is still typically needed. There is no other way to ensure, for example, that the reference to the immutable object will be seen by the second thread. The guarantees the program gets from final fields should still be used with the knowledge that thread visibility and other such problems addressed by synchronisation are still present.

Volatile in Java means a value which is always flushed to main memory, and as such stale reads (an old cache value) should not be able to have happen. Before a read, a cache flush must happen in order to get the value into memory, and on write to volatile then causes a cache flush because visibility. Under the old memory model volatile operations could not be reordered with respect to each other yet could be reordered with respect to non-volatile operations, and as such volatile operated in a non-intuitive manner, and could cause bad synchronisation, and was bad for signalling changes of condition between threads, one of the primary concerns behind the keyword. JSR133 states that volatile operations cannot be reordered at all, significantly strengthening the keyword nearly to the point of synchronised. All things that thread A can see when it writes to volatile f are made visible to thread B when it reads f. Note the f must be the same field for both A and B otherwise no happens-before ordering is established.

One particular example about how the old memory model did not specify synchronisation correctly is a common implementation of the lazy initialisation object creation paradigm, **double-checked locking** [6]. In which an object is only constructed when a program calls for it, and not on class initialisation. [16]

In lazy initialisation, an object is only instantiated when another object specifically calls for it, with the desired behaviour, written in Java, shown in Figure 6, taken from [6]. Since this is obviously incorrect from a multithreaded point of view, the correct and simple fix is to synchronise `getHelper()`.

However, synchronising `getHelper()` means that synchronisation is performed every time it is accessed when, theoretically, it only needs to be performed once. As such, the double-checked locking paradigm only synchronises the object creation, as shown in Figure 6, taken from [6].

Now, this does not work, either in practice or going by the specification. The compiler is free to reorder the object initialisation and return calls in such a way that a reference to the object that a reference to the object is returned before the object has finished initialising, exposing default field values. This is the case as long as the object does not synchronise and cannot throw an exception.

'Intuitive' methods of solving the problem also do not work, such as including a second synchronisation block inside the first, only containing the `helper = new Helper();` statement. The idea behind this is that all

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      helper = new Helper();
    return helper;
  }
}
```

Figure 5 – Lazy Initialisation

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      synchronized (this) {
        if (helper == null)
          helper = new Helper();
      }
    return helper;
  }
}
```

Figure 6 – Double-Checked Locking

statements in a synchronisation block have to be performed before exiting the block.

However, statements can still be moved inside the block, so moving `return helper;` into the second block is still valid, and then the same problem is faced as before.

The changes to volatile as part of JSR133, being that volatile reads and writes cannot be reordered at all with other reads and writes, means that the same code as Figure 5, except declaring helper as volatile. Double-checked locking will work with that code. It will also work if helper is immutable, due to changes to how immutables are constructed, as described earlier.

Parts of JSR133 were further clarified in JSR166 [4, 5] (a separate change to be implemented in Java 5, centred on concurrency in general), with confirmation of:

- Writing to a volatile having the same basic memory effects as lock, reading a volatile as unlock,
- Thread `start()` and `join()` also having the same basic memory effects as lock and unlock respectively.
- Final fields are guaranteed their actual final values if they are written in such a way that a reference to an object does not escape before the constructor terminates.
- Unsynchronised code is still guaranteed to be type safe, invisible and reorderable, in accordance with the basic security guarantees of Java, yet it does not allow for out-of-thin-air results or 'speculative' reads/writes, as these, as discussed later, are not sequentially consistent.

C. *Current Status*

Work on the Java memory model has not been an important task since the introduction of JSR133. Work on concurrency in general has been done in the form of JSR166, but this has largely been work on improving packages such as Java.util.concurrent, but not on the memory model.

As primary work on the C++ memory model has been completed and implemented as this paper will shortly discuss, a second look at re-specifying the Java memory model is possible, but not likely soon.

However, there has been research into whether JSR133 still over-specifies restrictions on compiler optimisations. The discussion [21] over whether disallowing speculative stores and out-of-thin-air variable is feasible for performance, and can be done without violating the other consistency rules but in place by JSR133.

III. The C++ Memory Model

A. History

C++, being based on C, did not have a defined memory model as part of the core specification in any standard C++03 or earlier. As the language also did not specify an implementation of threading, this was not seen as a large problem as threading libraries (this paper shall focus on pthreads (Posix [17] threads)) had also been written. In these libraries, as part of their existence, had to specify some level of memory model in order for any software programmer to reason over whether a given program would be correct.

An important point to be made about a separate threading library is that correctness is basically implementation dependant. The lack of ability to properly specify a memory model at the program language level results in a need to rely on the permissible transformations and reorderings in the compiler and underlying processor.

The simple pthread specific memory model is that any shared accesses must take place between mutex lock and unlock (or equivalent) operations, with the semantics of all data races going undefined. As compilers treat the synchronisation operations as opaque, it cannot reorder them. Also, as they map to atomic processor instructions, the processor cannot reorder the operations either. Thus, the model usually works.

Although, as shown in [1], there are a few ways in which threading, and as such the memory model not being part of the core language makes it very difficult to reason whether a given program is correct.

Some of these issues follow: Compilers allow intra-thread transformations which introduce speculative operations, particularly stores (an example is in Figure 2), will introduce data races as part of that transformation. Adjacent bit-fields in structs can be optimised into a single word, such that a write operation on the first bit-field will cause a write on the second, co accessing both concurrently can cause writes to be 'lost'.

Finally, there's the issue of performance. Since a interfacing with library functions can only involve going through the library, fine-grain speed is difficult to achieve. Indeed, by the results in [1] (partially brought in as Figure 7), by using synchronisation, it is very difficult to achieve the same sort of performance as unsynchronised code running on one core.

Each of these issues points to the largest issue being that the language itself does not specify a memory model. So, between C++03 and C++11 a memory model, largely based on the work done on JSR133 was specified, and is implemented as of C++11.

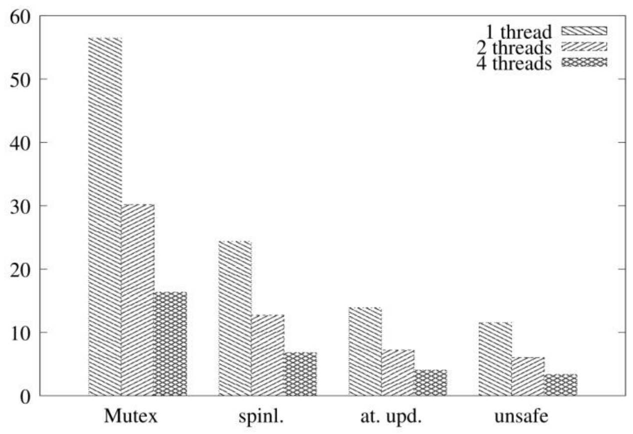

Figure 7 – Speed Comparison of Sieve of Eratosthenes algorithm. Published in [1].

B. Creating a Model

The C++ memory model, designed after the 2003 C++ standard (C++03) for what would become the 2011 C++ standard (C++11), was written in order to address concerns with the pthreads approach above, and to keep C++ current and portable for the modern approach to computer performance. It was based on the work done on the Java model, especially the research which went into JSR133 and keeps close to the programming ideas presented in the combination of C++ with pthreads.

The new memory model in C++11 calls for universal sequential consistency in its operations, with some exceptions that shall be delved into later, and no formal semantics are given to programs which are not correctly synchronised, as C++ operates with the assumption that any error could lead to arbitrary code execution.

Compilers are also no longer allowed certain code transformations, as the compiler cannot be expected to know which variable would be used for signalling, so the assumption is that any variable can be used for signalling between threads. However, later it will be shown that some transformation and reordering can take place regardless, and further restrictions can be applied to special synchronisation keywords.

As shown above in III.A, the primary concerns of moving the memory model from a library to language are:

1. The informal specifications in the libraries are ambiguous and do not define precisely the semantics of either what a data race is or how a data race-free program behaves.
2. Without these precise semantics it is difficult to determine if a given transformation will violate them.
3. There are many situations in which the act of synchronisation is unnecessarily expensive, and less expensive solutions are not portable across multiple systems, or are again, improperly specified.

In section I, the notion of sequential consistency is introduced, a model in which there is a total within-thread program order in which every operation is atomic and memory effects cannot be reordered with respect to each other or to the program order. This is intuitive and easy to synchronise, but many compiler optimisations cannot be applied, thus reducing the final speed of the application. It can be said that most compilers do not abide by sequential consistency.

Relaxed models have been proposed, such as data-race-free, which only guarantee sequential consistency to any program which is data race-free, and give no guarantee otherwise. Different uses of data-race-free define the notion of a data race. Data-race-free-0, the memory model currently used by Java, defines a data race as two concurrent conflicting accesses.

The C++ memory model is an adaptation of data-race-free-0, as it is known that it can be successfully implemented under the already restrictive environment of Java. However, certain problems came to light both as part of that work and following that work as regarding its ability to be implanted as part of C++. The largest three problems found in the implementation are:

- Semantically consistent atomics.
- Trylock() and its impact on data races.
- The semantics for data races.

1) Sequentially Consistent Atomics

In order to be data-race-free, all operations that are not normal data operations (so synchronisation, C++ atomics, Java volatiles etc.) must also appear sequentially consistent. Hardware considerations must then be taken into account, as many processors (prior to current systems, generally prior to AMD64 etc.) did not take this into account when performing its own optimisations on these instructions. It is now the case that processors perform optimisations in line with the C++ memory model in synchronisation and atomics.

Furthermore, exclusively having sequentially-correct-atomics is not universally suitable because:

- It is very expensive on some platforms, so non-sequentially-correct atomics need to be present for a faster alternative.
- A lot of code written prior to C++11, such as the Linux kernel, is written in such a way that assumes a much weaker memory model, and as such relies on the programmer to provide atomics which perform correctly on the target hardware. If the target hardware does not optimise correctly for sequentially-correct-atomics, then atomics which aren't sequentially correct can both provide greater performance, and an easier method for migrating pre-existing code which already accounts for the non-sequential correctness.

So C++11 allows for sequentially-correct as well as non-sequentially-correct atomics which are referred to as *low-level-atomics.* A mechanism to weaken the memory ordering and allow for these low-level-atomics is then required, and is much more complicated. As it is not expected for most programmers to use this system, a more basic model is described, which is equivalent to the model without these. Model: memory operations are viewed as operating on abstract memory locations and each scalar value occupies separate memory locations except contiguous bitfields in an innermost class/struct definition. C++ also defines a sequenced-before relation ($<_T$) on memory operations performed by a single thread, similar to the program order relation in Java. This is only partial order as C++ has undefined argument evaluation order.

Taken straight from [11], with work from [7, 8, 9, 10], is a formal definition of allowable code reordering optimisations follows:

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 modelled as a sequence of: load all fields, update, store all fields. In the execution of a thread: sequence of memory actions and sequenced-before order.

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).

The C++ memory model can now be defined simply as:

- If a program (on a given input) has a sequentially consistent execution with a (type 1) data race, then its behaviour is undefined.
- Otherwise, the program (on the same input) behaves according to one if its sequentially consistent executions.

Having defined this, the optimisations allowed are: 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): 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.

Non-atomic operations can still be operated on only to thread cache and such happen non-atomically, but synchronisation instructions are half ordered, will cause cache updates and can be reordered in some ways. The above still ensures that atomics are still executed atomically, and in accordance with program order.

2) Trylock()

Trylock() is a command which, if successful, results in the targeted mutex being locked, otherwise it is treated as a no-op. Trylock() can be used in ways which would either require a more complex definition of a data race and/or more strict memory fences, but this can be worked around.

Locking without blocking or lock acquisition with timeout have undesirably strong semantics from the above (i.e. we want to make trylock() efficient). Shown in Figure 8 is a situation (from [1]) where assert is not allowed to fail by sequential rules as $<_T$ means x=42 must be performed before trylock(). Except a valid hardware reordering is moving lock() before x=42, so a memory-fence need to be used in order to give the correct ordering (as by our rules the assert cannot be allowed to fail), but this is inefficient.

The problem is needing to read the state of the lock, so instead of introducing anything else to our order we make it so that trylock() can randomly fail, making our reorders correct, as a failed assert means that trylock() just failed.

```
T1          T2
x = 42;     while (trylock(l) ==
            success)
lock(l);    unlock(l);
            assert(x == 42);
```

Figure 8 - Problematic Trylock()

3) Semantics of Data Races

As mentioned earlier, C++11 gives ***no*** semantics to programs which contain data races, as it is assumed that no data race is benign. Certain analysis, such as that described in [12] can give some idea of whether certain data races are benign or not, but this is not taken into account in the C++11 model. Further Implications of this decision are discussed later.

4) Extensions to the Model

It can be shown that the model presented above is equivalent to a much more complicated model, which can then be extended to implement low-level-atomics. Memory ordering instructions using the AMD64/Intel64 architectures onwards [13, 14] specify that the mapping of atomic writes (in x86) is the xchg assembly operation. This is an atomic read-modify-write, which also implicitly provides a Store|Load fence (Store|Load means that all store operations before the fence are guaranteed to happen before all loads after the fence), which is just as efficient as enforcing a manual fence anyway. Other ways of solving this are discussed, but are largely platform dependent [23]. At this time, the largest target of compiled-to-assembly code is x86(-64) anyway.

The model as discussed here is the model expected to be used by C++ programmers in general. However, this is not the complete model. In order to aid in code porting to C++11, and as a performance tool to be used if the implications are understood, there is a mechanism to weaken the C++ memory model for atomic variables.

This extension specifically allows atomic variables to ignore sequential consistency. The specifics are out of scope for this discussion, but the basics of the mechanism are that a happens-before ordering (much like Java's) is introduced into the model, with a data race being redefined as two concurrent accesses with respects to happens-before.

Low-level atomics can then be implemented by selectively removing synchronisation operations that count for happens-before. Further details and proof of equivalence exists in the following: [11, 18, 19]

C. Future

In order for the C++11 memory model to continue functioning as designed, it requires continued support from hardware designers. Architectures designed for embedded and low power use are becoming increasing used in the mainstream due to the rise in mobile computing, and these architectures are therefore gaining capabilities such as out-of-order execution.

ARM processors, for example have increasingly common, and have gained superscalar and out-of-order architectures during the time between C++03 and C++11. Work needs to be done both on the architecture and compiler side in order to ensure that the memory model is upheld.

Otherwise, as the model has been successfully implemented and standardised, work has slowed. While a new C++ version is in development [20], no changes to the memory model are proposed.

IV. Comparison Between Models

As mentioned prior, although the new C++ memory model is heavily based on the large quantity of research that went into the Java Memory Model as updated by JSR133, the two models are quite separate.

Insofar as the general specification of the two memory models, the Java memory model must abide by a slightly weaker model from the developer's point of view. It uses a happens-before ordering in order to specify its data races. C++ on the other hand, can use sequenced before, which is only slightly weaker than program order, and is simpler to both specify and understand.

However, in concrete terms, C++ has the ability to weaken the memory model to happens-before. The point behind this mechanism is to allow the use of atomic variables which are not sequentially correct. In contrast, as operations cannot be reordered around Java volatiles, and access is always designed to be atomic, Java volatiles are always sequentially correct, even in lieu of performance.

Java, as part of its other language concern, must implement certain concerns into its memory model. These being; Data races under Java must still conform to type safety, data races must still not include speculative stores or derivation of out-of-thin-air variables.

In contrast to this, C++ has no such concerns, and simply gives no formal semantics to code which contains data races. This does mean that C++ in general is more likely to cause a system failure, but this risk isn't unique to threading and synchronisation. As such, the lack of specification is justified here.

However C++, in order to keep to sequential consistency rules, general also cannot insert speculative stores. Also, if the target language of the compiler (be it source code or specific machine assembly) prohibits speculative loads and stores both.

If ignoring the C++11 method for weakening sequential consistency, the C++ memory model is far simpler, only requiring a program ordering, and not needing to perform a slight global weakening and relying on the happens-before ordering.

V. Conclusion

This paper has reported on the current state of memory models to two very popular programming languages, Java and C++. Comparisons were also made between the two, showing that the two are quite different despite C++11 being built on knowledge gained from Java.

From this, it is clear that memory model research is much further ahead now, with 2 of the most commonly used programming languages having a memory model far more robust than ever previously considered.

Research has not stalled though, with more hardware architectures gaining features which further necessitate memory models, and the continuing desire for speed in correctly synchronised code driving thought into which disallowed compiler optimisations can be made permissible without sacrificing correctness.

Acknowledgement

Thanks to Dr Julian Rathke in his role as both supervisor for guidance and support, and draft/proof reader for this work.

References

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

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

[3] J. Manson and B. Goetz. 2004, February. The JSR-133 (Java Memory Model) FAQ [Online]. Available: http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html

[4] D. Lea. Concurrency jsr-166 interest site. Available: http://gee.cs.oswego.edu/dl/concurrency-interest.

[5] D. Lea. JSR-166 Documentation [Online]. Available: http://gee.cs.oswego.edu/dl/jsr166/dist/docs/.

[6] The “Double Checked Locking is Broken” Declaration [Online]. Available: http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

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

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

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

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

[11] H.-J. Boehm, S. V. Adve. Foundations of the C++ Concurrency Memory Model. 2008, May 21.

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

[13] Intel Corp. August 2007. Intel 64 Architecture Memory Ordering White Paper [Online]. Available: http://www.intel.com/products/processor/manuals/318147.pdf.

[14] AMD Corp. AMD64 Architecture Programmer’s Manual - Volume 2: System Programming, July 2007.

[15] B. Goetz. 2002, July. Java Theory and Practice: Safe Construction Techniques [Online]. Available: http://www.ibm.com/developerworks/java/library/j-jtp0618/index.html

[16] D. Schmidt and T. Harrison. Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects, 3rd annual Pattern Languages of Program Design conference, 1996

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

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

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

[20] H. Sutter, 2013. Trip Report: ISO C++ Spring 2013 Meeting. Available: http://isocpp.org/blog/2013/04/trip-report-iso-c-spring-2013-meeting

[21] Verbrugge, C.et. al. There is nothing wrong with out-of-thin-air: compiler optimization and memory models. In Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, 2011, pp. 1-6

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

[23] J. Alglave, et al. "Fences in weak memory models." Computer Aided Verification. Springer Berlin Heidelberg, 2010.