(To appear in Concurrency: Practice and Experience)

The Java Memory Model is Fatally Flawed

William Pugh
Dept. of Computer Science
Univ. of Maryland, College Park
`pugh@cs.umd.edu`

Abstract

The Java memory model described in Chapter 17 of the Java Language Specification gives constraints on how threads interact through memory. This chapter is hard to interpret and poorly understood; it imposes constraints that prohibit common compiler optimizations and are expensive to implement on existing hardware. Most JVMs violate the constraints of the existing Java memory model; conforming to the existing specification would impose significant performance penalties.

In addition, programming idioms used by some programmers and used within Sun's Java Development Kit is not guaranteed to be valid according the existing Java memory model. Furthermore, implementing Java on a shared memory multiprocessor that implements a weak memory model poses some implementation challenges not previously considered.

1 Introduction

The Java memory model, as described in chapter 17 of the Java Language Specification [GJS96], is very hard to understand. Research papers that analyze the Java memory model interpret it differently [GS97, CKRW97, CKRW98]. Guy Steele (one of the authors of [GJS96]) was unaware that the memory model prohibited common compiler optimizations, but after several days of discussion at OOPSLA98 agrees that it does.

Given the difficulty of understanding the memory model, there may be disagreements as to whether the memory model actually has all of the features I believe it does. However, I don't believe it would be profitable to spend much time debating whether it does have these features. I am convinced that the existing style of the specification will never be clear, and that attempts to patch the existing specification by adding new rules will make even harder to understand. If we decide to change the Java memory model, a completely new description of the memory model should be devised.

A number of terms are used in the Java memory model but not explicitly related to Java source programs nor the Java virtual machine. Some of these terms have been interpreted differently by various people. I have based my understanding of these terms on conversations with Guy Steele, Doug Lea and others.

A *variable* refers to a static variable of a loaded class, a field of an allocated object, or element of an allocated array. The system must maintain the following properties with regards to variables and the memory manager:

- It must be impossible for any thread to see a variable before it has been initialized to the default value for the type of the variable.
- The fact that a garbage collection may relocate a variable to a new memory location is immaterial and invisible to the memory model.

The existing Java memory model discusses *use*, *assign*, *lock* and *unlock* actions:

- A *use* action corresponds to a `getfield`, `getstatic` or array load (e.g., `aaload`) Java bytecode instruction.
- An *assign* action corresponds to a `putfield`, `putstatic` or array store (e.g, `aastore`) Java bytecode instruction.
- A *lock* action corresponds to a `monitorenter` Java bytecode instruction.
- A *unlock* action corresponds to a `monitorexit` Java bytecode instruction.

This work was supported National Science Foundation grants ACI9720199 and CCR9619808.

Initially: x = y = 0

Thread 1	Thread 2
a = x	b = y
y = 1	x = 1

Anomalous result: a = 1, b = 1

Figure 1: Execution valid for Java only due to prescient stores

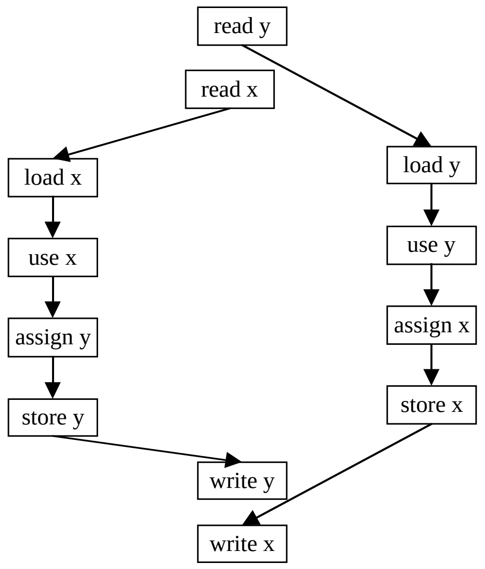

Figure 2: Actions and orderings for Figure 1 without prescient stores (with prescient stores, delete orderings from assign actions to store actions)

2 Features of the Memory Model

Due to the double indirection in the Java memory model, it is very hard to understand. What features does it provide?

Consider the example in Figure 1. Gontmakher and Schuster [GS97] state that this is an execution trace that is illegal for Java, but they are incorrect because they do not consider prescient stores [GJS96, §17.8]. Without prescient stores, the actions and ordering constraints required by the JMM are shown in Figure 2. Since the write of `y` is required to come after the read of `x`, and the write of `x` is required to come after the read of `y`, it is impossible for both the write of `x` to come before the read of `x` and for the write of `y` to come before the read of `y`.

With prescient stores, the *store* actions are not required to come after the *assign* actions; in fact, the *store* actions can be the very first actions in each thread. This makes it legal for the *write* actions for both `x` and `y` to come before either of the *read* actions, and for execution to result in `a = b = 1`.

What the JMM does require is Coherence [ABJ^{+}93]. Informally, for each variable in isolation, the uses and assigns to that variable must appear as if they acted directly on global memory in some order that respects the order within each thread (i.e., each variable in isolation is sequentially consistent). A proof that the Java memory model requires Coherence is given in [GS97]. That paper didn't consider prescient stores, but it doesn't impact the proof that the JMM requires Coherence; even with prescient stores, the load and store actions for a particular variable cannot be reordered.

In discussions, Guy Steele stated that he had intended the JMM model to have this property, because he felt it was too non-intuitive for it not to. However, Guy was unaware of the implications of Coherence on compiler optimizations (below).

```
// p and q might be aliased
int i = p.x
// concurrent write to p.x
// by another thread
int j = q.x
int k = p.x
```

Figure 3: Example showing that reads kill

2.1 Coherence means that reads kill

Consider the code fragment in Figure 3 Since `p` and `q` only might be aliased, but are not definitely aliased, then the use of `q.x` cannot be optimized away (if it were known that `p` and `q` pointed to the same object, then it would be legal to replace the assignments to `j` and `k` with assignments of the value of `i`). Consider the case where `p` and `q` are in fact aliased, and another thread writes to the memory location for `p/q.x` between the first use of `p.x` and the use of `q.x`; the use of `q.x` will see the new value. It will be illegal for the second use of `p.x` (stored into `k`) to get the same value as was stored into `i`. However, a fairly standard compiler optimization would involve eliminating the getfield for `k` and replacing it with a reuse of the value stored into `i`. Unfortunately, that optimization is illegal in any language that requires Coherence.

If all three reads were from `p.x`, then it would also be illegal to replace the third load with a reuse of the value from the first load. However, in that case there wouldn't be any motivation to perform that op-

```
// p and q might be aliased
int i = r.y
int j = p.x
// concurrent write
// to p.x by another thread
int k = q.x
p.x = 42
```

Figure 4: Counter example to JMM ≡ Coherence

timization; instead, you would just replace both the second and third loads with a reuse of the value from the first load (which would be legal). The problem is that the memory model is specified in terms of variables; the compiler may have two different names or aliases for the same variable, but the semantics need to be identical to the case where they had the same name. If `p` and `q` were not aliased, it would be legal to perform this optimization.

One way to think of it is that since a read of a memory location may cause the thread to become aware of a write by another thread, it must be treated in the compiler as a possible write.

In talking with a number of people at OOPSLA98, I found that most people were not aware of the implications for compilers of Coherence in the JMM. Most existing JVM's perform optimizations that violate Coherence; this variance between the specification and implementation is recorded as Javasoft Bug # 4242244. That bugreport (available online at Javasoft) gives code that tests to see if a JVM performs the prohibited transformation. The testing code is somewhat complicated because some JVM's (such as HotSpot) don't compile optimized code for a method until the second time the method is invoked, and because it has to check for whether threads are context switching in the middle of loop bodies and whether other optimizations are being performed.

Dan Scales, of Digital Western Research Laboratories, did a preliminary study of the impact of Coherence on performance. His results [Sca99] suggested that for computational intensive programs, such as the mpegaudio benchmark, enforcing "reads kill" in the compiler results in programs running 20% - 45% slower. Research on new techniques could probably lower this cost, and better alias analysis would help as well. Still, Coherence would create substantial complications in compiler internal representations. For example, in an SSA (Static Single Assignment) representation [AWZ88, RWZ88] of a program, each read would introduce a new phi node.

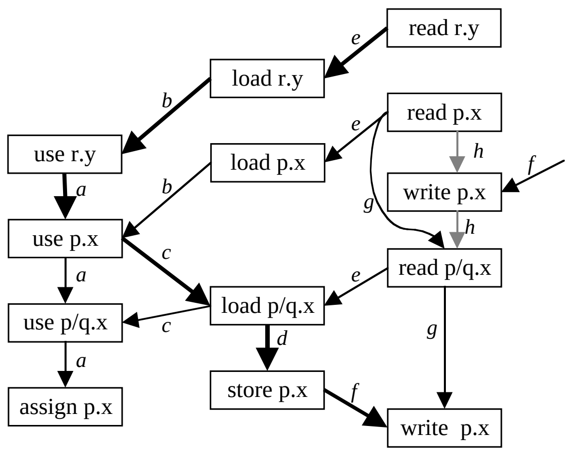

Figure 5: JMM actions for Figure 4

2.2 JMM is stronger than Coherence

Initially, I tried to derive a proof that, excluding locks and volatile variables, the Java memory model is exactly Coherence. Instead, I came up with a counterexample. Consider the code fragment in Figure 4, and the scenario in which `p` and `q` are aliased (although we are not able to prove it), and another write happens to update the value of `p/q.x` between the read of `p.x` and the read of `q.x`, so that the use of `p/q.x` sees a different value than the use of `p.x`. The actions corresponding this execution, and their ordering constraints, are shown in Figure 5.

The boxes and arrows in this diagram arise for the following reasons:

a [GJS96, §17.3, bullet 1]: All use and assign actions by a given thread must occur in the order specified by the program being executed.

b [GJS96, §17.3, bullet 4]: ... must perform a load before performing a use.

c Since the use of `p/q.x` sees a different value than the use of `p.x`, there must be a separate load instruction for the use of `p/q.x`, which must precede the use of `p/q.x` and follow the use of `p.x`.

d [GJS96, §17.8, bullet 3]: No load of a value V intervenes between the relocated [prescient] store and the assign of V.

e [GJS96, §17.3, second list of bullets, 1st bullet]: For each load, there must be a corresponding preceding read.

f [GJS96, §17.3, second list of bullets, 2nd bullet]: For each store, there must be a corresponding following write.

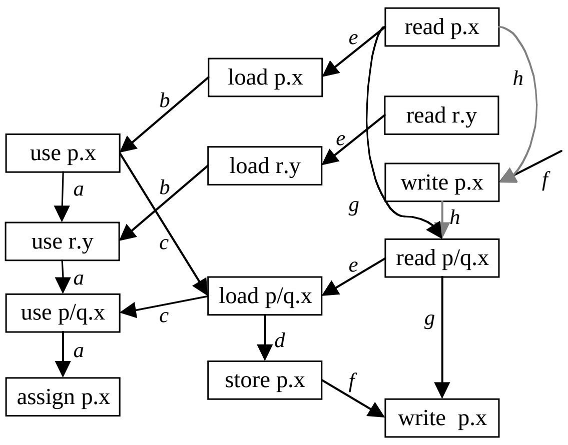

Figure 6: JMM actions for Figure 4 after re-ordering use of r.y and use of p.x

g [GJS96, §17.2, 2nd bullet]: actions performed by main memory for any one variable are totally ordered.

[GJS96, §17.3, second list of bullets, 3rd bullet]: edges between load/store actions on a variable V and the corresponding read/write actions cannot cross.

h Since we consider the situation where `p` and `q` are aliased and the use of `p/q.x` sees a different value than the use of `p.x`, there must have been an intervening write to `p.x` by another thread between the load of `p.x` and the load of `p/q.x`.

Since the existing JMM is often interpreted differently by different people, I went over this example with Guy Steele and he agrees that the existing JMM imposed the constraints shown in Figure 5.

The important point of Figure 5 is that there are a series of constraints that force the read of `r.y` to occur before the write of `p.x`. This ordering constraint is unexpected, unintuitive, unwanted, and can only be enforced on some processors by explicit memory barrier instructions. It also would cause substantial and unknown complications for optimizing Java compilers. However, a formal reading of specification requires that the ordering constraint be enforced.

2.2.1 Bytecode reorderings are illegal

In Figure 5 it would be legal for the `read r.y` action to occur after the `read p.x` action. But if we tried to perform this transformation at the bytecode level (moving the `getfield r.y` instruction to after the `getfield p.x` action), we get the actions shown in Figure 6. In these set of actions, it *would* be legal to perform the `read r.y` action after the `write p.x` action. So the set of legal transformations on Java programs are not closed under composition. You can't perform a transformation at the bytecode level without reasoning about whether or not there might exist any downstream component that might perform a reordering that, when composed with your reordering, produces an illegal reordering of the memory references.

This pretty much prohibits any bytecode transformations of memory references. Although it would depend on your intermediate representation, it is likely that similar problems would arise for any internal representation of a Java program used in a JIT (just-in-time) compiler.

3 Safety Guarantees and Weak Memory Models

Many shared memory multiprocessors implement a weak memory model. These weak memory models might allow higher performance, but also produce surprising results when multithreaded programs are not properly synchronized.

Although surprising results can arise in many programming languages, they are particular severe in object oriented languages and in languages that make safety guarantees. The difficulty for object oriented programs is due to the number of "hidden" data structures manipulated by the runtime system (e.g., the virtual function table) and the richer mental model programmers have of objects (and their surprise when these models are violated by improperly synchronized programs).

In devising a new memory model for Java, we need to be aware of the issues raised by weak memory models, and think about the safety guarantees that should be made in the language specification and the implementation cost of making those guarantees.

3.1 Weak memory models

A memory model describes how different threads/processors can see their memory actions interleave with those of other processors. A strong memory model, such as sequential consistency, imposes very strict constraints.

There are many weak memory models. They all tend to support and/or need explicit memory barriers or acquire/release operations. A possible implementation/intuition is shown in Figure 7. Each processor

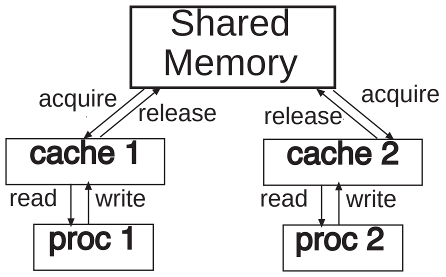

Figure 7: Implementation of a Weak Memory Model

```
Initially:
class Foo {
  final int x;
  int f(int y) { return x+y; }
  Foo(int i) { x = i; }
  static Foo bar;
  }
Foo.bar = new Foo(1);
```

```
   On Processor 1:           On Processor 2:
Foo.bar = new Foo(2);   int z = Foo.bar.f(3);
```

Figure 8: Unsynchronized access

reads and writes to a local cache. Updated memory locations may be flushed from the cache to main memory at any time, and the cache can fill a memory location from main memory at any time. When a processor does an acquire/lock operation, it must reload the cache from main memory, and when a processor does a release/unlock operation, it must flush all modified memory locations from the cache to main memory. Multiprocessors based on the DEC Alpha and the Intel Merced chip provide this kind of weak memory order.

3.1.1 What can go wrong?

Consider execution of Figure 8 on a multiprocessor with a weak memory model; the processors have unsynchronized access to `Foo.bar`. The initial code is executed first and all processors see it. Next, it happens that processor 1 executes its code first. Furthermore, the cache happens to flush the modification of Foo.bar back to main memory, but none of the other writes. Then, when processor 2 executes its code, it loads its cache only with the new value of `Foo.bar` from main memory (finding old values in the cache for all other memory locations).

What unexpected things could happen? When processor 2 executes `Foo::f` and reads the `x` field of the object allocated by processor 1, it won't see the value that the `x` field was initialized to by processor 1. It could read garbage: an arbitrary value. Since `x` is only an integer value, this is only moderately bad. If `x` were a reference/pointer, then seeing a garbage value would violate type safety and make any kind of security/safety guarantee impossible.

We can allocate objects out of memory that all processors agree has been zeroed. Essentially, you would zero memory during garbage collection, and then have all processors perform a memory barrier/acquire before restarting after a garbage collection. This would ensure that if processor 2 sees a stale value for the `x` field of the object allocated by processor 1, it will see zero/null. For references/pointers, this will ensure type safety.

We could require that a processor perform a release/flush operation between creating an object and publishing a reference to that object (by publishing, I mean store a reference to the object in a place where it might be read by other threads). A compiler could easily figure out where such flush operations are required (e.g., after object initialization) and the cost of doing such flushes would likely be small.

However, it isn't enough. Under a weak memory model, processor 2 must also do a barrier/acquire operation to see all of the writes sent to main memory by processor 1. The problem is that there isn't anything in the code executed by processor 2 to suggest that we might be reading a reference to an object created by another processor.

Perhaps we should just decide that seeing a zero/null value for a field is OK. In Java, that is the default value for a field, and if you allow an object to escape before it is properly initialized, that it what you will see anyway. (Some people are horrified by this idea, but let's run with it for the moment).

However, in an OO environment, we also have to consider the object header fields. For example, when processor 2 reads the `vtbl` (virtual method table) entry from the object referenced by `Foo.bar`, it might see null. Dispatching the `Foo::f()` method could result in a `SIGSEGV` fault crashing your virtual machine; this clearly should *not* be considered acceptable.

Other information must be considered suspect in a multithreaded environment. In Java, the length of an array might be seen as zero. In C++, the pointer to a virtual base class might be null.

There are even worse problems if you consider dynamic class loading. Consider what happens if processor 1, rather than just creating a instance of a class that processor 2 already knows about, loads an entirely new class Faz (a subclass of Foo that overrides `f()`), compiles native code for `Faz::f()` from bytecode, creates an instance of Faz, and then stores a reference to that instance in `Foo.bar`). There is

```
public MyFrame extends Frame {
  private MessageBox mb;
  private showMessage(String msg) {
    if (mb == null) {
      synchronized(this) {
        if (mb == null)
          mb = new MessageBox();
        }
    mb.setMessage(msg);
    mb.pack(); mb.show();
    }
  // .. more methods and variables ...
}
```

Figure 9: Double-check and lazy instantiation idioms

still nothing in the code being executed by processor 2 to indicate that it will need to synchronize. However, any of the memory locations read by processor 2 might be null. Even if the reference to the virtual function table isn't null, an entry of the `vtbl` could be null. Processor 2 might not see the native code generated by processor 1.

What makes this particular difficult is that just *one* of the memory locations read by processor 2 could be stale, even though all the others see properly updated values. Just checking to see if you got a valid pointer to a `vtbl` won't suffice.

4 Unsafe idioms

Many Java programmers assume that immutable objects, such as `Strings`, are automatically thread safe. But this isn't true under the existing Java memory model, and it may be difficult to efficiently implement this guarantee on share memory multiprocessors with weak memory models.

Consider what happens when thread 1 creates a `String`, stores a reference to it in shared variable, and then thread 2 reads that variable. Neither thread does any synchronization. It is possible that thread 2 could see the reference written by thread 1, but none of the writes that set the value of the fields of the `String`. At some later point, thread 2 might see those writes. Thus, thread 2 could see the value of the string change, perhaps from "/tmp" to "/usr".

For a number of security reasons, it is absolutely essential that `String`s be atomic and immutable. The appropriate way to guarantee that is open to discussion.

Another example of a programming idiom that is unsafe according to the current Java Memory Model is the double-check and lazy instantiation idioms, described in a recent article [BW99b] and book [BW99a, Chap. 9]. Figure 9 shows this idiom. This idiom is unsafe because the writes that initialize the `MessageBox` don't need to be sent to main memory before the storing of the reference to the `MessageBox` into `mb`.

5 The semantics of data races

Few programming languages have defined the semantics of programs that contain unsynchronized access to shared data. Ada and Modula3 define a multithreaded semantics, but simply say that it is erroneous to have unsynchronized access to shared data.

A group of people are discussing a number of safety guarantees that can't be taken for granted in unsynchronized code. An example is *initialization safety*: if an object isn't made visible outside the constructor until after the constructor terminates, then no code, even unsynchronized code in another thread, can see that object without seeing all of the effects of the constructor for that object. However, we don't know the cost to implement this safety guarantee on multiprocessors with weak memory models.

6 More information

There is a mailing list for discussion of a specification of the multithreaded semantics of Java and the issues involved with implementing OO runtime systems on multiprocessors with weak memory models. Information at:

http://www.cs.umd.edu/~pugh/java/memoryModel

Acknowledgments

Thanks to the many people who are participating in the discussions of this topic, particularly Sarita Adve, Joshua Block, Joseph Bowbeer, Sanjay Ghemawat, Paul Haahr, Doug Lea, Raymie Stata, Guy Steele and Dennis Sosnoski.

References

[ABJ+93] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proceedings of the Fifth ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 1993.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[BW99a] Philip Bishop and Nigel Warren. *Java in Practice: Design Styles and Idioms for Effective Java.* Addison-Wesley, 1999.

[BW99b] Philip Bishop and Nigel Warren. Lazy instantiation: Balancing performance and resource usage. *JavaWorld*, 1999. http://www.javaworld.com/javaworld/javatips/jw-javatip67.html.

[CKRW97] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From sequential to multi-threaded java: An event-based operational semantics. In *In Proc. 6th Int. Conf. Algebraic Methodology and Software Technology*, Berlin, October 1997. Springer-Verlag.

[CKRW98] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. *Formal Syntax and Semantics of Java.* Springer-Verlag, 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[GS97] Alex Gontmakher and Assaf Schuster. Java consistency: Non-operational characterizations for the java memory behavior. Technical Report CS0922, Dept. of Computer Science, Technion, November 1997.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 12–27, January 1988.

[Sca99] Dan Scales. Impact of ”reads kill” in java. http://www.cs.umd.edu/ ~pugh/java/readsKillImpact.html, May 1999.