Fixing the Java Memory Model

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. The Java memory model is hard to interpret and poorly understood; it imposes constraints that prohibit common compiler optimizations and are expensive to implement on existing hardware. At least one shipping optimizing Java compiler violates the constraints of the existing Java memory model. These issues are particularly important for high-performance Java applications, since they are more likely to use and need aggressive optimizing compilers and parallel processors.

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.

This paper reviews these issues and suggests replacement memory models for Java.

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.

In addition to the problem that the memory model is very hard to understand, it has two basic problems: it is too weak and it is too strong. It is too strong in that it prohibits many compiler optimizations and requires many memory barriers on architectures such Sun's Relaxed Memory Order (RMO). It is too weak in that much of the code that has been written for Java, including code in Sun's JDK, is not guaranteed to be valid.

2 The Java Memory Model

In this section, I try to interpret *JMM*, the existing Java Memory Model, as defined in Chapter 17 of the Java Language Specification [GJS96]. The same definition also appears in Chapter 8 of the Java Virtual Machine Specification [LY96].

A number of terms are used in the Java memory model but not 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

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

To appear at the Java Grande Conference, June 12–14, 1999

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

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.

2.1 Bug fixes

The JMM suggests that at thread termination, a thread doesn't need to write back the results of assigns to main memory. This is obviously (to me) a bug and I assume it is fixed by saying that there must be a store associated with the last assign to a variable in a thread.

The JMM also doesn't force a thread to pushed cached writes out to main memory before starting a new thread. This has been acknowledged as a bug.

2.2 Interpretation

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*

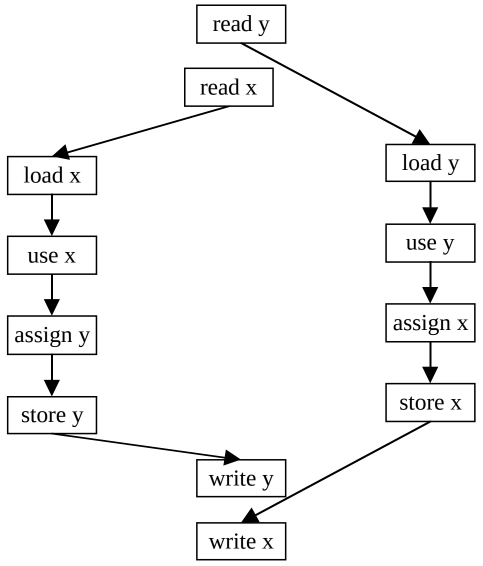

Figure 2: Actions and orderings for Figure 1 without prescient stores (with prescient stores, delete orderings from assign actions to store actions)

```
// 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

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).

2.3 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.

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, and at least one shipping commercial Java compiler violates Coherence.

2.4 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 counter-example. 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.

```
// 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

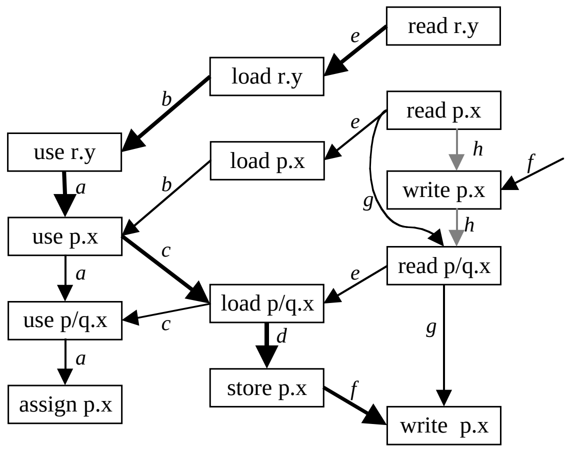

Figure 5: JMM actions for Figure 4

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 V intervenes between the relocated [prescient] store and the assign.

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

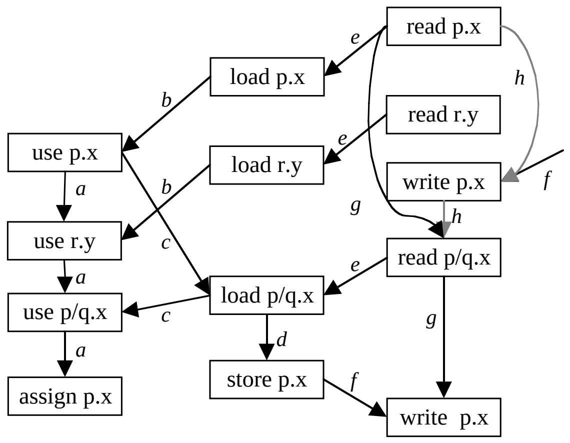

Figure 6: JMM actions for Figure 4 after re-ordering use of r.y and use of p.x

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

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`.

I showed this example to Guy Steele and he tentatively agreed that the JMM imposed the constraints shown in Figure 5, although he did not double check it at length.

This ordering constraints was definitely not intended, and has a substantial impact on optimizing Java compilers and on Java programs running on aggressive processor architectures.

2.4.1 Reorderings are not closed under composition

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.

There may be other strange constraints imposed by the existing JMM, but at this point we switch from trying to decipher the existing JMM to deciding what features we want in a new Java memory model.

3 Reality

We would like the Java memory model to interfere as little as possible with compiler optimizations and to not require memory barrier instructions on hardware with loose memory models, such as the Sparc V9 Relaxed Memory Order (RMO) [WG94].

Here are some of the issues that drive us to weaken the memory model. All of these are in the absence of explicit synchronization:

1. We want to give the compiler/optimizer freedom to reorder instructions that could be reordered in a single threaded environment.
2. We want to allow the compiler/optimizer to do forward substitution / scalar replacement (e.g., replace a getfield instruction with a reuse of the value last stored into that variable).
3. We want to allow the processor to reorder instructions during execution.
4. We want to allow the processor to use a write-buffer.

As it turns out, issue 1 is largely equivalent to issue 3, and issue 2 is largely equivalent to issue 4.

3.1 Instruction Reordering

In memory models such as the Sparc-V9 Relaxed Memory Order (RMO) [WG94, Chap. 8], the processor execute instructions out of order, so long as it does so in a way that would not be detectable in absence of any shared memory interaction with other processors. In doing so, the processor is allowed to rename registers (allowing it to ignore output and anti dependences on registers) and perform control-speculation on loads so as to reduce the ordering constraints. However, it does have to respect output and anti dependences for memory locations.

Initially: a[0] = a[1] = 2

Processor 1	Processor 2
a[0] = 1	a[1] = 0
w = a[0]	y = a[1]
x = a[w]	z = a[y]

Anomalous result: x = z = 2

Figure 7: Execution only possible due to write buffer

3.2 Write Buffers

The memory models for most processors ignore the cache: instructions can be reordered, but when the instructions execute, they update main memory immediately (this is, of course, only a model). Directly following this model would be expensive, so most memory models are relaxed further by allowing a write buffer. When a write is initiated, it goes into the write buffer. The write is not considered to actually occur until it reaches main memory. If a read occurs for a memory location in the write buffer, the read gets the value in the memory buffer. In essence, this allows the processor to ignore flow dependences on memory locations when reordering instructions, and yet still get the right answer. Figure 7 shows a program execution legal only due to the existence of a write buffer in the memory model (without a write buffer, flow dependences would order the statements in each thread).

3.3 Coherence is difficult

As noted above, the existing Java memory model enforces Coherence. Unfortunately, Coherence cannot be enforced on architectures such as Sparc RMO without memory barriers. The Sparc RMO doesn't not guarantee that reads of the same memory location will be executed in their original order. To enforce this, a load/load memory barrier is required between any two successive loads of the same memory location. It is unclear if any existing implementations of the Sparc RMO would actually violate Coherence.

As mentioned earlier (Section 2.3), Coherence also interferes with a number of compiler optimizations.

3.4 Flushing memory is expensive

The semantics of the `lock` and `unlock` actions in the JMM are that they cause a thread to flush all dirty variables from the thread's working memory (registers, cache, ...) to main memory, and a `lock` action also causes a thread empty all variables from the thread's working memory, so that they have to be reloaded from main memory before they can be used.

Some have suggested that, particularly in a multiprocessor server, this will be expensive. An alternative would be to say that only memory accessed inside the synchronized block is flushed/emptied. This would probably be a good idea if you were designing a memory model from scratch, although more analysis is needed. However, people writing to the current memory model might expect that

```
synchronized(unsharedObject) {}
```

would have the effect of a memory barrier. Careful thought is required about the amount of existing code that would break if this change were made.

4 A New Proposal

In this section, I propose a new Java memory model. This model is closely coupled to the Java virtual machine. The rules for Java source programs can be derived by a simple and naive translation from Java source to Java bytecode, and then using the rules of this model. A Java thread executes `read`, `write`, `lock`, `unlock` and `think` actions:

- A `read` action corresponds to a getfield, getstatic or arrayload Java bytecode instruction.
- An `write` action corresponds to a putfield, putstatic or arraystore Java bytecode instruction.
- A `lock` action corresponds to a monitorenter Java bytecode instruction.
- A `unlock` action corresponds to a monitorexit Java bytecode instruction.
- A `think` action corresponds to all other Java bytecode instructions.

A `memory` action is either a `read` or `write` action.

Within a thread, there is a direct dependence ordering between two actions A and B if A occurs before B in the original program, and:

1. there is a flow dependence from A to B (i.e., the value computed/read/written by A effects the action performed by B). Issues such as stack depth and stack manipulation instructions (e.g., `swap`) are ignored in determining flow dependences.
2. A and B are `lock` and `memory` actions (either order).
3. A is a `write` action and B is an `unlock` action (either order).
4. A and B are both `memory` actions on the same variable and at least one of them is a `write` action.
5. A and B are both memory action on volatile variables.

The required dependence ordering of actions is the transitive closure of direct dependence ordering. Actions within a thread can be ordered in any way that respects these orders. These constraints make it impossible to determine that a threads actions have been reordered, except through interaction with another thread or other external agent (e.g., a debugger).

Control Speculation Note that there is no requirement that the ordering of actions respect control dependences (there is a control dependence when one instruction influences whether another instruction will be performed). The Sparc RMO memory model allows reordering of loads that doesn't respect control dependences (e.g., speculative loads), but doesn't allow speculative stores (since you can't undo them). Defining control dependence in Java is a little tricky, since many instructions (and all memory actions) can throw an exception that prevent following instructions from occurring). If we included such exceptions in computing control dependence, then we wouldn't be able to perform any reordering of writes at all.

Instead, we allow actions to be reordered as though the system had exact knowledge of the path of program execution. Loads may be done speculatively, and stores may be done in a manner that appears to be speculative. However, a store may not be performed unless it is guaranteed that the thread will execute the store (excluding situations such as a `VirtualMachineError` or `ThreadDeath` error). This is intended to allow the compiler to use any form of static or runtime analysis to predict which paths will be taken and which exceptions cannot be thrown.

Scalar Replacement If a `memory` action A and a `read` action B reference the same non-volatile variable and A and B are reordered so that B immediately follows A, then B can be replaced with a `think` action that computes the same value as was read/written by A. This rule subsumes both scalar replacement by the compiler and write buffers within a processor. For example, this rule, combined with the reordering rules above, allow for the behavior seen in Figure 7. Without this scalar replacement rule, such behavior would be illegal.

Dead Store Elimination If two write actions A and B reference the same non-volatile variable and A and B are reordered so that A immediately precedes B, then A can be deleted.

4.1 Comparison with the Old JMM

My proposed Java memory model is neither stronger nor weaker than the existing Java memory model. My

Initially: a[0] = 3, a[1] = a[2] = 0

Processor 1		Processor 2	
Thread 1	Thread 2	Thread 3	Thread 4

a[1] = 2	w = a[1]
	x = a[w]
		a[2] = 1	y = a[2]
			z = a[y]

Anomalous result: w = 2, x = 0, y = 1, z = 0

Figure 8: Interference by other threads on same processor

model requires that memory operations not be reordered in a way that violations the data dependences of the program, while the old model does not. However, it is hard to imagine how one could take advantage of the additional freedom offered by the old model.

On the other hand, my model does not require Coherence nor does it impose anomalous constraints such as shown in Figures 4 – 5.

4.2 Enforcing Coherence

The above proposal is designed so that in the absence of synchronization, it has no impact on compiler optimizations and can be executed on architectures such as the Sparc V9 RMO without memory barriers. However, it does not enforce Coherence, while the original JMM did. The only effect this has is on successive reads of the same variable.

The benifits of enforcing Coherence is unclear. But if is it desired, Coherence can be enforced by changing rule 4 so that there is a direct dependence even if both A and B are `read` actions.

4.3 Threads, not Processors

One issue that needs to be addressed is that processor memory models are in terms of *processors*, while the Java memory model is in terms of *threads* and has no concept of processors. Consider the example in Figure 8. Threads 2 and 4 should only be able to see the writes to `a[1]` and `a[2]` through main memory. Which write happened first in main memory? If `y = 0`, then we must have `w = 2` and the write of `a[1]` occurring before the write of `a[2]`. If `z = 0`, then we must have `x = 2` and the write of `a[2]` occurring before the write of `a[1]`. This suggests that the result in Figure 8 can't happen.

However, unless we are careful, it can. Consider the case where, on processor 1, the write to `a[1]` is initiated first, followed by the instructions for thread 2. On processor 2, the write to `a[2]` is executed before the instructions for thread 4. All of the instructions in thread 2 and 4 finish execution before the writes from threads 1 and 3 exit the write buffers on processors 1 and 2. In this case, `w` and `x` will get their values from the write buffer, and `y` and `z` could get their values from the cache

Initially: p = new Point(1,2)

Thread 1	Thread 2
p = new Point(3,4);	a = p.x

a = 0 (!?), 1 or 3

Figure 9: Reordering of field initialization and ref update

Initially: this.p = new Point(1,2)

Thread 1	Thread 2

synchronized (this) {
this.p = new Point(3,4);}
	a = this.p.x;

a = 0 (!?), 1 or 3

Figure 10: Synchronized reordering of field initialization and ref update

(which is coherent because the writes haven't exited the write buffer).

One way to fix this is to require a memory barrier when switching threads on a processor. On a multiprocessor that implemented the Sparc RMO memory model, you would need a `Membar #Lookaside` instruction as part of a context switch. The context switch is probably expensive enough that you won't notice the cost of the `Membar` instruction.

On an architecture such as the Tera, which has very fast context-switching (between instructions), this could prove to be more of a problem. It might be possible to weaken the memory model to allow for the execution shown in Figure 8, but I'll leave that for another time.

5 The JMM is too weak

Joshua Bloch of Javasoft was one of the first to recognize that many of the idioms used in writing Java programs were not guaranteed to be safe according to the JMM. Consider the example in Figure 9. The JMM given in [GJS96, Chap 17] doesn't require that the writes initializing the point allocated by thread 1 be sent to main memory before the write of the reference to the newly created point into `p`, nor does it require that the read of `p.x` be done after the read of `p`.

This is rather unpleasant. For one things, final fields aren't final. Even if a field is declared as a final, this loophole could allow another thread accessing the object might see the default value for the field. In all kinds of code, you would need to worry about whether the object a method is invoked on is properly initialized.

Note that synchronization isn't a magic fix to this problem. If we add synchronization to the update, but not to the read (as in Figure 10), we still have the exact same problem; both all writes need to be sent to

Initially: this.p = new Point(1,2)

Thread 1	Thread 2

synchronized (this) {
synchronized (this) {
tmp = new Point(3,4);}
this.p = tmp; }
	a = this.p.x;

a = 0 (!?), 1 or 3

Figure 11: More synchronization doesn't help

```
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 12: Double-check and lazy instantiation idioms

main memory before the unlock action, but they can be sent in any order. You might think that putting a `monitorexit` between the creation of the `Point` and the storing of the `Point` into `this.p` might fix the problem; this is equivalent to making the constructor synchronized (see Figure 11). Unfortunately, this doesn't fix the problem either, because in the existing JMM, the write to `this.p` can be moved above the `monitorexit` instruction. The only way to fix this in the existing JMM is to require that the reader be synchronized.

Now of course, you can always say "Don't write code with race conditions!" But if you were writing a library that was sensitive from a security viewpoint, you would have to worry about other programmers using race conditions to attack your code. To fix this, we probably need to make all of the get*Foo*() methods synchronized (a get*Foo*() method is one that provides controlled access to a field/attribute *Foo* of an object). In the `java.*`, `java.*.*` and `java.*.*.*` packages of Sun's 1.2 distribution, there are a total of 829 get*Foo*() methods that return object references, of which only 26 are synchronized. Also, encouraging programmers to be very aggressive about using synchronization could also introduce more problems with deadlock.

Another example of a programming idiom that is unsafe according to the current JMM is the double-check and lazy instantiation idioms, described in a recent article [BW99b] and book [BW99a, Chap. 9]. Figure

12 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`.

I am convinced that we must fix this problem by making it possible to enforce an order on the writes. Trying to solve this problem solely by requiring synchronization whenever accessing shared data just isn't going to work.

I don't believe that there are any current Java implementations that could exhibit the behavior shown in Figures 9 – 11. As a result, few developers would bother avoiding idioms like that, feeling confident that they won't get bit. However, with advanced optimizing compilers and aggressive architectures, we might see this behavior down the road, at which point a huge codebase of unsafe code will exist.

Before trying to fix the problem, we should explore it in more detail. The basic problem in Figure 9 is that there are two writes to global memory that can be reordered, either by compiler optimizations or by the processor. There is no dependence forcing one write to come after the other, so the ordering is feasible and plausible unless we forbid it. If these writes are reordered, it could be detected by other threads, possibly with severe consequences. The reads might also be reordered, but this is more difficult because the memory location read by the second read is dependent on the value read by the first read.

In addition to arising in constructors, as shown in Figures 9 – 11, it also arises in the situations shown in Figures 13 – 15. If we are going to prohibit the anomalous behavior in Figures 9 – 11, we should also examine the behavior in Figures 13 – 15 and decide if they need to be prohibited.

I am not going to give a definitive answer. Instead, I will suggest several solutions, and discuss which behaviors they prohibit and their potential impact on compiler optimizations. My suggestions are roughly ordered from least protection/least cost to highest protection/highest cost, except making unlock a bidirectional write-barrier, which I consider a necessary prerequisite.

Unlock must be a bidirectional write-barrier The first fix that must be made to allow an ordering constraint to be imposed on writes. The existing JMM [GJS96, §17.6] prohibits moving a store/write to after an unlock action, but it doesn't prohibit a store/write from being moved to before an unlock action. The existing JMM can be patched by making an unlock action act as a bi-directional store/write barrier. In my proposed new Java memory model, I have already make this change (item 3 of Section 4).

Once this change is made, Figure 11 can no longer exhibit anomalous behavior. You might try to fix the problems in Figures 9 – 10 by declaring the constructors as synchronized. Unfortunately, that isn't legal in Java. Without additional changes, the only solution would be to put synchronized blocks inside in each constructor. This would work, but it would be a substantial pain.

p = new Point(1,1), q = new Point(2,2)

Thread 1	Thread 2
q.x = 3	
p = q	a = p.x

a = 1, 2 (!?) or 3

Figure 13: Reordering of field update and ref update

int a[] = {1,2}, b[] = {3,4}

Thread 1	Thread 2
a[0] = 17	
b = a	i = b[0]

i = 1, 17 or 3 (?)

Figure 14: Reordering of element update and ref update

5.1 Don't do that

The easiest solution is to say "Don't write programs with race conditions", and to not prohibit any of the anomalous behavior. Although I think that people need to be much more leery of race conditions than many are, I don't recommend this approach. Among other problems, a package developer would have to worry too much about whether users were avoiding data races. A developer could put synchronized blocks inside constructors to prohibit the behavior of Figures 9 – 10 on a case-by-case basis, but I suspect few developers would.

5.2 Allowing constructors to be synchronized

By allowing programmers to specify that a constructor is synchronized, a developer could, on a case-by-case basis, prohibit the behavior in Figures 9 –10 . This would be easier than putting synchronized blocks in constructor methods, but still I suspect few developers would bother doing so.

5.3 Ordering writes across a constructor completion

We could add the following rule to the set given in Section 4

a if A and B are both write actions, A writes to a field of an object X, B writes X into some variable, A occurs during some constructor C invoked to create X, and B occurs after C finishes, then there is a direct dependence ordering between A and B and they cannot be reordered.

Note that since there isn't (and shouldn't be) any action corresponding to a completion of a constructor, keeping track of this requirement requires more than just looking at the actions. In particular, if a constructor has been inlined, then forcing the appropriate ordering constraints might require forcing some sort of memory barrier (as in Section 5.4).

This will complete prohibit the behavior Figure 9. It won't completely prohibit seeing pre-initialized values for final fields. If a constructor passes `this` to another method before all of the final fields are initialized, the other method can see them (but this is an evil programming style). This doesn't prohibit any of the behavior in Figures 13 – 15.

5.4 Forcing a write barrier after a constructor call

We could say that the completion of a constructor call acts as a special `barrier` action, and add the rule:

b if A and B are a `write` action and a `barrier` action (either order), then there is a direct dependence ordering between A and B and they cannot be reordered.

The virtual machine could enforce a rule that the completion of a constructor acted as a `write` barrier, in the same way as a `unlock` action. This could apply to all constructors, or the spec might only require a `write` barrier at the completion of the outermost constructor (although putting one at the completion of every constructor would be allowed).

This is similar to allowing constructors to be synchronized. But since it doesn't actually lock the object, it couldn't possibly cause deadlock and would likely have minimal effects on performance. Thus, we don't have to worry about which constructors to synchronize; we just force a write barrier after `every` object is constructed. It also doesn't force a thread to empty the thread's working memory, so it may be less expensive than synchronization.

This approach is a little simpler to explain than rule a above, since it is simply explained in terms of actions. However, in code that created lots of light weight objects (particularly if the Java language is changed to provide better support for light weight objects than can be unboxed), the large number of memory barriers generated could significantly reduce the transformations that could be applied to the program.

5.5 Ordering writes

If we want to prohibit the anomalous behavior in Figures 13 – 14, we can do it by imposing constraints on the reordering of writes (the reordering of the reads in these examples is prohibited by the data dependence between the two reads).

We have a couple of options as to how strong we want this constraint to be. The basic, strong form of it is:

c if A and B are both write actions, A writes to a field or element of an object X, B writes X into some variable, then there is a direct dependence ordering between A and B and they cannot be reordered.

This would prohibit the anomalous behavior in Figures 9 – 14. We can relax this with any combination of the following two options:

1. Enforce c only if the A writes is to a field, not an element.

2. Enforce c only if the B writes X to a volatile variable.

One of the problems with enforcing this ordering is that we have to enforce it whenever a write *might* be to a field/element of an object being stored by the later write. Others I discussed this with expressed the opinion that this constraint should only be enforced if the writes involve the same variable, so that you know they reference the same object. In other words, if I write to `p.x`, then I write `p` into some variable, then I can't reorder the writes. However, if I write to `p.x`, and then I write `q` into some variable, than I can reorder them even though `p` and `q` might reference the same object. The problem with basing this constraint on variable names is that while variable names are fairly obvious in Java source code, they are not present in the Java virtual machine. When writing `p`, the value being stored comes off of the stack and might have gotten there through any number of stack manipulation instructions. Defining this constraint so that it only enforced the constraint when the same "variable" is involved would be very difficult to define and implement at the JVM level.

A decision to enforce one of these constraints should not be made without an understanding of the performance impact. Particularly due to the problem with aliases, the impact could be substantial (e.g., you would pretty much have to insert a write barrier before any store of an `java.lang.Object`, because it might reference any object that you have previously updated.

If we enforce this constraint only when the second write is to a volatile field, I suspect the performance impact will not be substantial. It makes a certain amount of sense, because if you are playing with data races, making your variables volatile is appropriate.

5.6 Prohibit all write reorderings

If you want to prohibit the anomalous behavior in Figure 15, I think you would really have to prohibit all write reorderings. But I don't think this should be seriously considered. This example is just a straw man a = {1,2}, i = 1

Thread 1	Thread 2
a[2] = 17	
i = 2	j = a[i]

j = 1, 2(?!) or 17

Figure 15: Reordering of element update and index update

to suggest that we are going to have to accept some anomalous behavior due to write reordering.

6 Conclusion

In this paper, I have described some of the problems with the existing Java memory model: it has unforeseen impacts on compiler optimizations, requires memory barriers on architectures such as the Sparc RMO even in the absence of synchronization, renders unsafe programming idioms commonly used, and is very hard to understand.

Intentionally writing code with data races is something best reserved for low-level native implementations of synchronization primitives. Most programmers should just not count on any specific behavior in code containing data races. However, the expectation that all objects are properly initialized (assuming the constructors are written properly), seems a worthwhile property to guarantee.

The existing Java memory model impacts both compiler optimization and insertion of memory barriers. Unfortunately, I have no empirical data on the performance impact of these issues. Part of the problem is that the impact may be minimal now, but grow as compilers and processor architectures become more aggressive.

More debate is needed on the Java memory model, and I have no illusions that this paper will settle the issue. But I hope it will be an important step in discussions leading to a solution.

Acknowledgments

Thanks to whole chorus of people who I discussed this topic with, including Guy Steele, Doug Lea, David Holmes, Josh Bloch and Sarita Adve.

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.

[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.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[WG94] David Weaver and Tom Germond. *The SPARC Architecture Manual, version 9.* Prentive-Hall, 1994.