Semantics of Multithreaded Java

Jeremy Manson and William Pugh

Institute for Advanced Computer Science and Dept. of Computer Science
Univ. of Maryland, College Park
{jmanson,pugh}@cs.umd.edu

March 29, 2001

Abstract

Java has integrated multithreading to a far greater extent than most programming languages. It is also one of the only languages that specifies and requires safety guarantees for improperly synchronized programs. It turns out that understanding these issues is far more subtle and difficult than was previously thought. The existing specification makes guarantees that prohibit standard and proposed compiler optimizations; it also omits guarantees that are necessary for safe execution of much existing code. Some guarantees that are made (e.g., type safety) raise tricky implementation issues when running unsynchronized code on SMPs with weak memory models.

This paper reviews those issues. It proposes a new semantics for Java that allows for aggressive compiler optimization and addresses the safety and multithreading issues.

1 Introduction

Java has integrated multithreading to a far greater extent than most programming languages. One desired goal of Java is to be able to execute untrusted programs safely. To do this, we need to make safety guarantees for unsynchronized as well as synchronized programs. Even potentially malicious programs must have safety guarantees.

Pugh [Pug99, Pug00b] showed that the existing specification of the semantics of Java's memory model [GJS96, §17] has serious problems. However, the solutions proposed in the first paper [Pug99] were naïve and incomplete. The issue is far more subtle than anyone had anticipated.

Many of the issues raised in this paper have been discussed on a mailing list dedicated to the Java Memory Model [JMM]. There is a rough consensus on the solutions to these issues, and the answers proposed here are similar to those proposed in another paper [MS00] (by other authors) that arose out of those discussions. However, the details and the way in which those solutions are formalized are different.

The authors published a somewhat condensed version of this paper [MP01]. Some of the issues dealt with in this paper, such as improperly synchronized access to longs and doubles, were elided in that paper.

2 Memory Models

Almost all of the work on memory models area has been done on *processor* memory models. Programming language memory models differ in some important ways.

First, most programming languages offer some safety guarantees, such as type safety. These guarantees must be absolute: there must not be a way for a programmer to circumvent them.

Second, the run-time environment for a high level language contains many hidden data structures and fields that are not directly visible to a programmer (for example, the pointer to a virtual method table). A data race resulting in the reading of an unexpected value for one of these hidden fields could be impossible to debug and lead to substantial violations of the semantics of the high level language.

Third, some processors have special instructions for performing synchronization and memory barriers. In a programming language, some variables have special properties (e.g., volatile or final), but there is usually no way to indicate that a particular write should have special memory semantics.

Finally, it is impossible to ignore the impact of compilers and the transformations they perform. Many standard compiler transformations violate the

This work was supported by National Science Foundation grants ACI9720199 and CCR9619808, and a gift from Sun Microsystems.

rules of existing memory models [Pug00b].

2.1 Terms and Definitions

In this paper, we concern ourselves with the semantics of the Java virtual machine [LY99]. While defining a semantics for Java source programs is important, there are many issues that arise only in the JVM that also need to be resolved. Informally, the semantics of Java source programs is understood to be defined by their straightforward translation into classfiles, and then by interpreting the classfiles using the JVM semantics.

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 semantics.
- The fact that two variables may be stored in adjacent bytes (e.g., in a byte array) is immaterial. Two variables can be simultaneously updated by different threads without needing to use synchronization to account for the fact that they are "adjacent".

3 Proposed Informal Semantics

The proposed informal semantics are very similar to *lazy release consistency* [CZ92, GLL+90]. A formal operational semantics is provided in Section 7.

All Java objects act as monitors that support reentrant locks. For simplicity, we treat the monitor associated with each Java object as a separate variable. The only actions that can be performed on the monitor are Lock and Unlock actions. A Lock action by a thread blocks until the thread can obtain an exclusive lock on the monitor.

The actions on monitors and volatile fields are executed in a sequentially consistent manner (i.e., there must exist a single, global, total execution order over these actions that is consistent with the order in which the actions occur in their original threads). Actions on volatile fields are always immediately visible to other threads, and do not need to be guarded by synchronization.

If two threads access a normal variable, and one of those accesses is a write, then the program should be synchronized so that the first access is *visible* to the second access. When a thread T_1 acquires a lock on/enters a monitor m that was previously held by another thread T_2, all actions that were visible to T_2 at the time it released the lock on m become visible to T_1.

If thread T_1 starts thread T_2, then all actions visible to T_1 at the time it starts T_2 become visible to T_2 before T_2 starts. Similarly, if T_1 joins with T_2 (waits for T_2 to terminate), then all accesses visible to T_2 when T_2 terminates are visible to T_1 after the join completes.

When a thread T_1 reads a volatile field v that was previously written by a thread T_2, all actions that were visible to T_2 at the time T_2 wrote to v become visible to T_1. This is a strengthening of volatile over the existing semantics. The existing semantics make it very difficult to use volatile fields to communicate between threads, because you cannot use a signal received via a read of a volatile field to guarantee that writes to non-volatile fields are visible. With this change, many broken synchronization idioms (e.g., double-checked locking [Pug00a]) can be fixed by declaring a single field volatile.

There are two reasons that a value written to a variable might not be available to be read after it becomes visible to a thread. First, another write to that variable in the same thread can overwrite the first value. Second, additional synchronization can provide a new value for the variable in the ways described above. Between the time the write becomes visible and the time the thread no longer can read that value from that variable, the write is said to be *eligible* to be read.

When programs are not properly synchronized, very surprising behaviors are allowed.

There are additional rules associated with final fields (Section 5) and finalizers (Section 6)

4 Safety guarantees

Java allows untrusted code to be executed in a *sandbox* with limited access rights. The set of actions allowed in a sandbox can be customized and depends upon interaction with a security manager, but the ability to execute code in this manner is essential. In a language that allows casts between pointers and integers, or in a language without garbage collection, any such guarantee is impossible. Even for code that is written by someone you trust not to act maliciously, safety guarantees are important: they limit the possible effects of an error.

Safety guarantees need to be enforced regardless of whether a program contains a synchronization error or data race.

In this section, we go over the implementation issues involved in enforcing certain virtual machine safety guarantees, and in the issues in writing libraries that promise higher level safety guarantees.

4.1 VM Safety guarantees

Consider execution of the code on the left of Figure 1*a* on a multiprocessor with a weak memory model (all of the `r`*i* variables are intended to be registers that do not require memory references). Can this result in `r2` = -1? For this to happen, the write to `p` must precede the read of `p`, and the read of `*r1` must precede the write to `y`.

It is easy to see how this could happen if the `MemBar` (Memory Barrier) instruction were not present. A `MemBar` instruction usually requires that actions that have been initiated are completed before any further actions can be taken. If a compiler or the processor tries to reorder the statements in Thread 1 (leading to `r2 = -1`), then a `MemBar` would prevent that reordering. Given that the instructions in thread 1 cannot be reordered, you might think that the data dependence in thread 2 would prohibit seeing `r2 = -1`. You'd be wrong. The Alpha memory model allows the result `r2 = -1`. Existing implementations of the Alpha wouldn't actually reorder the instructions. However, existing Alpha implements could fulfill the `r2 = *r1` instruction out of a stale cache line, which has the same effect. Future implementations may use value prediction to actually allow the instructions to be executed out of order.

Stronger memory orders, such as TSO (Total Store Order), PSO (Partial Store Order) and RMO (Relaxed Memory Order) would not allow this reordering. Sun's SPARC chip typically runs in TSO mode, and Sun's new MAJC chip implements RMO. Intel's IA-64 memory model does not allow `r2 = -1`; the IA-32 has no memory barrier instructions or formal memory model (the implementation changes from chip to chip), but many knowledgeable experts have claimed that no IA-32 implementation would allow the result `r2=-1` (assuming an appropriate ordering instruction was used instead of the memory barrier).

Now consider Figure 1*b*. This is very similar to Figure 1*a*, except that `y` is replaced by heap allocated memory for a new instance of Point. What happens if, when Thread 2 reads `Foo.p`, it sees the address written by Thread 1, but it doesn't see the writes performed by Thread 1 to initialize the instance?

When thread 2 reads `r2.x`, it could see whatever was in that memory location before it was allocated from the heap. If that memory was uninitialized before allocation, an arbitrary value could be read. This would obviously be a violation of Java semantics. If `r2.x` were a reference/pointer, then seeing a garbage value would violate type safety and make any kind of security/safety guarantee impossible.

One solution to this problem is allocate objects out of memory that all threads know to have been zeroed (perhaps at GC time). This would mean that if we see an early/stale value for `r2.x`, we see a zero or null value. This is typesafe, and happens to be the default value the field is initialized with before the constructor is executed.

Now consider Figure 1*c*. When thread 2 dispatches `hashCode()`, it needs to read the virtual method table of the object referenced by `r2`. If we use the idea suggested previously of allocating objects out of prezeroed memory, then the repercussions of seeing a stale value for the `vptr` is limited to a segmentation fault when attempting to load a method address out of the virtual method table. Other operations such as arraylength, instanceOf and checkCast could also load header fields and behave anomalously.

But consider what happens if the creation of the Bar object by Thread 1 is the very first time Bar has been referenced, and this forces the loading and initialization of class Bar. Then not only might thread 2 see a stale value in the instance of Bar, thread 2 could see a stale value in any of the data structures or code loaded for class Bar. What makes this particularly tricky is that thread 2 has no indication that it might be about to execute code of a class that has just been loaded.

4.1.1 Proposed VM Safety Guarantees

Synchronization errors can only cause surprising or unexpected values to be returned from a read action (i.e., a read of a field or array element). Other actions, such as getting the length of an array, performing a checked cast or invoking a virtual method behave normally. They cannot throw any exceptions or errors because of a data race, cause the VM to crash or be corrupted, or behave in any other way not allowed by the semantics.

Values returned by read actions must be both typesafe and "*not out of thin air*". To say that a value must be "*not out of thin air*" means that it must be a value written previously to that variable by some thread. For example, Figure 7 must not be able to

Initially
p = &x; x = 1; y = -1

Thread 1	Thread 2
y = 2	r1 = p
MemBar	r2 = *r1
p = &y	

Could result in
r2 = -1
(*a*)

Initially
Foo.p = new Point(1,2)

Thread 1	Thread 2
r1 = new Point(3,4)	r2 = Foo.p
MemBar	r3 = r2.x
Foo.p = r1	

Could result in
r3 = 0 or garbage
(*b*)

Initially
Foo.o = "Hello"

Thread 1	Thread 2
r1 = new Bar(3,4)	r2 = Foo.o
MemBar	r3 = r2.hashCode()
Foo.o = r1	

Could result in
almost anything
(*c*)

Figure 1: Surprising results from weak memory models

produce any result other than `i == j == 0`. The exception to this is that incorrectly synchronized reads of non-volatile longs and doubles are not required to respect the "*not out of thin air*" rule (see Section 7.7 for details).

4.2 Library Safety guarantees

Many programmers assume that immutable objects (objects that do not change once they are constructed) do not need to be synchronized. This is only true for programs that are otherwise correctly synchronized. However, if a reference to an immutable object is passed between threads without correct synchronization, then synchronization within the methods of the object is needed to ensure that the object actually appears to be immutable.

The motivating example is the `java.lang.String` class. This class is typically implemented using a length, offset, and reference to an array of characters. All of these are immutable (including the contents of the array), although in existing implementations are not declared final.

The problem occurs if thread 1 creates a String object `S`, and then passes a reference to `S` to thread 2 without using synchronization. When thread 2 reads the fields of `S`, those reads are improperly synchronized and could see the default values for the fields of `S`, then later reads by thread 2 could see the values set by thread 1.

As an example of how this can affect a program, it is possible to show that a String that is supposed to be immutable can appear to change from "/tmp" to "/usr". Consider an implementation of StringBuffer whose `substring` method creates a string using the StringBuffer's character array. It only creates a new array for the new String if the StringBuffer is changed. We create a String using `new StringBuffer ("/usr/tmp") .substring(4);`. This will produce a string with an offset field of 4 and a length of 4. If thread 2 incorrectly sees an offset with the default value of 0, it will think the string represents "/usr" rather than "/tmp". This behavior can only occur on systems with weak memory models, such as an Alpha SMP.

Under the existing semantics, the only way to prohibit this behavior is to make all of the methods and constructors of the String class synchronized. This solution would incur a substantial performance penalty. The impact of this is compounded by the fact that the synchronization is not necessary on all platforms, and even then is only required when the code contains a data race.

5 Guarantees for Final fields

If an object contains mutable fields, then synchronization is required to protect the class against attack via data race. Therefore, we propose allowing immutable objects to be defended by use of final fields.

Final fields must be assigned exactly once in the constructor for the class that defines them. The existing Java memory model contains no discussion of final fields. In fact, at each synchronization point, final fields need to be reloaded from memory just like normal fields.

We propose additional semantics for final fields. These semantics will allow more aggressive optimizations of final fields, and allow them to be used to guard against attack via data race.

5.1 When these semantics matter

The semantics defined here are only significant for programs that either:

- Allow objects to be made visible to other threads before the object is fully constructed
- Have data races

We strongly recommend against allowing objects to escape during construction. Since this is simply a

```
class ReloadFinal extends Thread {
  final int x;
  ReloadFinal() {
    synchronized(this) {
      start();
      sleep(10);
      x = 42;
      }
    };
  public void run() {
    int i,j;
    i = x;
    synchronized(this) {
      j = x;
      }
    System.out.println(i + ", " + j);
    // j must be 42, even if i is 0
    }
  }
```

Figure 2: Final fields must be reloaded under existing semantics

matter of writing constructors correctly, it is not too difficult a task. While we also recommend against data races, defensive programming may require considering that a user of your code may deliberately introduce a data race, and that there is little or nothing you can do to prevent it.

5.2 Final fields of object that escape their constructors

Figure 2 shows an example of where the existing specification requires final fields to be reloaded. In this example, the object being constructed is made visible to another thread before the final field is assigned. That thread reads the final field, waits to be signaled that the constructor has assigned the final field, and then reads the final field again. The current specification guarantees that even if the first read of `tmp1.x` in `foo` sees 0, the second read will see 42.

It is difficult to give a useful semantics of final fields that can be seen to change. So we don't. The (informal) rule for final fields is that you must ensure that the constructor for a object has completed before another thread is allowed to load a reference to that object.

5.3 Informal semantics of final fields

The formal detailed semantics for final fields are given in Section 7.6. For now, we just describe the informal semantics of final fields that are constructed properly.

The first part of the semantics of final fields is:

F1 When a final field is read, the value read is the value assigned in the constructor.

Consider the scenario postulated at the bottom of Figure 3. The question is, which of the variables `i1` - `i7` are guaranteed to see the value 42?

F1 alone guarantees that `i1` is 42. However, that rule isn't sufficient to make Strings absolutely immutable. Strings contain a reference to an array of characters; the contents of that array must be seen to be immutable in order for the String to be immutable. Unfortunately, there is no way to declare the contents of an array as final in Java. Even if you could, it would mean that you couldn't reuse the mutable character buffer from a StringBuffer in constructing a String.

To use final fields to make Strings immutable requires that when we read a final reference to an array, we see both the correct reference to the array and the correct contents of the array. Enforcing this should guarantee that `i2` is 42. For `i3`, the relevant question is: do the contents of the array need to be set before the final field is set (i.e, `i3` might not be 42), or merely before the constructor completes (`i3` must be 42)?

Although this point is debatable, we believe that a requirement for objects to be completely initialized before they are assigned to final fields would often be ignored or incorrectly performed. Thus, we recommend that the semantics only require that such objects be initialized before the constructor completes.

Since `i4` is very similar to `i2`, it should clearly be 42. What about `i5`? It is reading the same location as `i4`. However, simple compiler optimizations would simply reuse the value loaded for `j` as the value of `i5`. Similarly, a processor using the Sparc RMO memory model would only require a memory barrier at the end of the constructor to guarantee that `i4` is 42. However, ensuring that `i5` is 42 under RMO would require a memory barrier by the reading thread. For these reasons, we recommend that the semantics not require that `i5` be 42.

All of the examples to this point have dealt with references to arrays. However, it would be very confusing if these semantics applied only to array elements and not to object fields. Thus, the semantics should require that `i6` is 42.

We need to decide if these special semantics apply only to the fields/elements of the object/array directly referenced, or if it applies to those referenced indirectly. If the semantics apply to indirectly referenced fields/elements, then `i7` must be 42. We be-

```
class FinalTest {
  public static FinalTest ft;
  public static int [] x = new int[1];
  public final int a;
  public final int [] b,c,d;
  public final Point p;
  public final int [][] e;

  public FinalTest(int i) {

    a = i;

    int [] tmp = new int[1];
    tmp[0] = i;
    b = tmp;

    c = new int[1];
    c[0] = i;

    FinalTest.x[0] = i;
    d = FinalTest.x;

    p = new Point();
    p.x = i;

    e = new int[1][1];
    e[0][0] = i;
    }
```

```
  static void foo() {
    int [] myX = FinalTest.x;
    int j = myX[0];
    FinalTest f1 = ft;
    if (f1 != null || j == -1) return;
                          // Guaranteed to see value
                          // set in constructor?
    int i1 = f1.a;        // yes
    int i2 = f1.b[0];     // yes
    int i3 = f1.c[0];     // yes
    int i4 = f1.d[0];     // yes
    int i5 = myX[0];      // no
    int i6 = f1.p.x;      // yes
    int i7 = f1.e[0][0];  // yes
    // use i1 ... i7
    }
  }


// Thread 1:
//   FinalTest.ft = new FinalTest(42);

// Thread 2:
//   FinalTest.foo();
```

Figure 3: Subtle points of the revised semantics of final

lieve making the semantics apply only to directly referenced fields would be difficult to program correctly, so we recommend that `i7` be required to be 42.

To formalize this idea, we say that a read r2 is derived from a read r1 if

- r2 is a read of a field or element of an address that was returned by r1, or
- there exists a read r3 such that r3 is derived from r1 and r2 is derived from r3.

Thus, the additional semantics for final fields are:

F2 Assume thread T1 assigns a value to a final field f of object X defined in class C. Assume that T1 does not allow any other thread to load a reference to X until after the C constructor for X has terminated. Thread T2 then reads field f of X. Any writes done by T1 before the class C constructor for object X terminates are guaranteed to be ordered before and visible to any reads done by T2 that are derived from the read of f.

5.4 Final Static Fields

Final static fields must be initialized by the class initializer for the class in which they are defined. The semantics for class initialization guarantee that any thread that reads a static field sees all the results of the execution of the class initialization.

Note that final static fields do not have to be reloaded at synchronization points.

Under certain complicated circumstances involving circularities in class initialization, it is possible for a thread to access the static variables of a class before the static initializer for that class has started. Under such situations, a thread which accesses a final static field before it has been set sees the default value for the field. This does not otherwise affect the nature or property of the field (any other threads that read the static field will see the final value set in the class initializer). No special semantics or memory barriers are required to observe this behavior; the standard memory barriers required for class initialization ensure it.

5.5 Native code changing final fields

JNI allows native code to change final fields. To allow optimization (and sane understanding) of final fields, that ability will be prohibited. Attempting to use JNI to change a final field should throw an immediate exception.

5.5.1 Write Protected Fields

`System.in`, `System.out`, and `System.err` are final static fields that are changed by the methods `System.setIn`, `System.setOut` and `System.setErr`. This is done by having the methods call native code that modifies the final fields. We need to create a special rule to handle this situation.

These fields should have been accessed via getter methods (e.g., `System.getIn()`). However, it would be impossible to make that change now. If we simply made the fields non-final, then untrusted code could change the fields, which would also be a serious problem (functions such as `System.setIn` have to get permission from the security manager).

The (ugly) solution for this is to create a new kind of field, *write protected*, and declare these three fields (and only these fields) as write protected. They would be treated as normal variables, except that the JVM would reject any bytecode that attempts to modify them. In particular, they need to be reloaded at synchronization points.

6 Guarantees for Finalizers

When an object is no longer reachable, the `finalize()` method (i.e., the finalizer) for the object may be invoked. The finalizer is typically run in a separate finalizer thread, although there may be more than one such thread.

The loss of the last reference to an object acts as an asynchronous signal to another thread to invoke the finalizer. In many cases, finalizers should be synchronized, because the finalizers of an unreachable but connected set of objects can be invoked simultaneously by different threads. However, in practice finalizers are often not synchronized. To naïve users, it seems counter-intuitive to synchronize finalizers.

Why is it hard to make guarantees? Consider the code in Figure 4. If `foo()` is invoked, an object is created and then made unreachable. What is guaranteed about the reads in the finalizer?

An aggressive compiler and garbage collector may realize that after the assignment to `ft.y`, all references to the object are dead and thus the object is unreachable. If garbage collection and finalization were performed immediately, the write to `FinalizerTest.z` would not have been performed and would not be visible.

But if the compiler reorders the assignments to `FinalizerTest.x` and `ft.y`, the same would hold for `FinalizerTest.x`. However, the object referenced

```
class FinalizerTest {
 static int x = 0;
 int y = 0;
 static int z = 0;

 protected void finalize() {
  int i = FinalizerTest.x;
  int j = y;
  int k = FinalizerTest.z;
  // use i, j and k
  }

 public static void foo() {
  FinalizerTest ft = new FinalizerTest();
  FinalizerTest.x = 1;
  ft.y = 1;
  FinalizerTest.z = 1;
  ft = null;
  }
 }
```

Figure 4: Subtle issues involving finalization

by `ft` is clearly reachable at least until the assignment to `ft.y` is performed.

So the guarantee that can be reasonably made is that all memory accesses to the fields of an object X during normal execution are ordered before all memory accesses to the fields of X performed during the invocation of the finalizer for X. Furthermore, all memory accesses visible to the constructing thread at the time it completes the construction of X are visible to the finalizer for X. For a uniprocessor garbage collector, or a multiprocessor garbage collector that performs a global memory barrier (a memory barrier on all processors) as part of garbage collection, this guarantee should be free.

For a garbage collector that doesn't "stop the world", things are a little trickier. When an object with a finalizer becomes unreachable, it must be put into special queue of unreachable objects. The next time a global memory barrier is performed, all of the objects in the unreachable queue get moved to a finalizable queue, and it now becomes safe to run their finalizer. There are a number of situations that will cause global memory barriers (such as class initialization), and they can also be performed periodically or when the queue of unreachable objects grows too large.

7 Formal Specification

The following is a formal, operational semantics for multithreaded Java. It isn't intended as a method anybody would use to implement Java. A JVM implementation is legal iff for any execution observed on the JVM, there is a execution under these semantics that is observationally equivalent.

The model is a global system that atomically executes one operation from one thread in each step. This creates a total order over the execution of all operations. Within each thread, operations are usually done in their original order. The exception is that writes and stores may be done *presciently*, i.e., executed early (§7.5). Even without prescient writes, the process that decides what value is seen by a read is complicated and nondeterministic; the end result is not sequential consistency.

7.1 Operations

An operation corresponds to one JVM opcode. A getfield, getstatic or array load opcode corresponds to a Read. A putfield, putstatic or array store opcode corresponds to a Write. A monitorenter opcode corresponds to a Lock, and a monitorexit opcode corresponds to an Unlock.

7.2 Types and Domains

value A primitive value (e.g., int) or a reference to a object.

variable Static variable of a loaded class, a field of an allocated object, or element of an allocated array.

GUID A globally unique identifier assigned to each dynamic occurrence of write. This allows, for example, two writes of 42 to a variable v to be distinguished.

write A tuple of a variable, a value (the value written to the variable), and a GUID (to distinguish this write from other writes of the same value to the same variable).

7.3 Simple Semantics, excluding Final Fields and Prescient Writes

Establishing adequate rules for final fields and prescient writes is difficult, and substantially complicates the semantics. We will first present a version of the semantics that does not allow for either of these.

There is a set allWrites that denotes the set of all writes performed by any thread to any variable. For any set S of writes, $S(v) \subseteq S$ is the set of writes to v in S.

For each thread t, at any given step, overwritten_t is the set of writes that thread t knows are overwritten and previous_t is the set of all writes that thread t knows occurred previously. It is an invariant that for all t,

$$\text{overwritten}_t \subset \text{previous}_t \subseteq \text{allWrites}$$

Furthermore, all of these sets are monotonic: they can only grow.

When each variable v is created, there is a write w of the default value to v s.t. $\text{allWrites}(v) = \{w\}$ and for all t, $\text{overwritten}_t(v) = \{\}$ and $\text{previous}_t(v) = \{w\}$.

When thread t reads a variable v, the value returned is that of an arbitrary write from the set

$$\text{allWrites}(v) - \text{overwritten}_t$$

This is the set of writes that are eligible to be read by thread t for variable v. Every monitor and volatile variable x has an associated overwritten_x and previous_x set. Synchronization actions cause information to be exchanged between a thread's previous and overwritten sets and those of a monitor or volatile. For example, when thread t locks monitor m, it performs $\text{previous}_t \cup = \text{previous}_m$ and $\text{overwritten}_t \cup = \text{overwritten}_m$. The semantics of Read, Write, Lock and Unlock actions are given in Figure 5.

If your program is properly synchronized, then whenever thread t reads or writes a variable v, you must have done synchronization in a way that ensures that all previous writes of that variable are known to be in previous_t. In other words,

$$\text{previous}_t(v) = \text{allWrites}(v)$$

From that, you can do an induction proof that initially and before and after thread t reads or writes a variable v,

$$\mid \text{allWrites}(v) - \text{overwritten}_t \mid = 1$$

Thus, the value of v read by thread t is always the most recent write of v: $\text{allWrites}(v) - \text{overwritten}_t$. In a correctly synchronized program, there will therefore only be one eligible value for any variable in any thread at a given time. This results in sequential consistency.

7.4 Explicit Thread Communication

Starting, interrupting or detecting that a thread has terminated all have special synchronization semantics, as does initializing a class. Although we could add special rules to Figure 5 for these operations, it is easier to describe them in terms of the semantics of hidden volatile fields.

1. Associated with each thread T1 is a hidden volatile *start* field. When thread T2 starts T1, it is as though T2 writes to the *start* field, and the very first action taken by T1 is to read that field.

2. When a thread T1 terminates, as its very last action it writes to a hidden volatile *terminated* field. Any action that allows a thread T2 to detect that T1 has terminated is treated as a read of this field. These actions include:

    - Calling join() on T1 and having it return due to thread termination.
    - Calling isAlive() on T1 and having it return false because T1 has terminated.
    - Being in a shutdownHook thread after termination of T1, where T1 is a non-daemon thread that terminated before virtual machine shutdown was initiated.

3. When thread T2 interrupts or stops T1, it is as though T2 writes to a hidden volatile *interrupted* field of T1, that is read by T1 when it detects or receives the interrupt/threadDeath.

4. After a thread T1 initializes a class C, but before releasing the lock on C, it writes "true" to a hidden volatile static field *initialized* of C.

    If another thread T2 needs to check that C has been initialized, it can just check that the *initialized* field has been set to true (which would be a read of the volatile field). T2 does *not* need to obtain a lock on the class object for C if it detects that C is already initialized.

7.5 Need for Prescient Writes

Consider the example in Figure 6. If the actions must be executed in their original order, then one of the reads must happen first, making it impossible to get the result i == j == 1. However, a compiler might decide to reorder the statements in each thread, which would allow this result.

writeNormal(Write $\langle v, w, g\rangle$)
  overwritten$_t \cup =$ previous$_t(v)$
  previous$_t + = \langle v, w, g\rangle$
  allWrites$+ = \langle v, w, g\rangle$

readNormal(Variable v)
  Choose $\langle v, w, g\rangle$ from
    allWrites(v) $-$ overwritten$_t$
  return w

lock(Monitor m)
  Acquire/increment lock on m
  previous$_t \cup =$ previous$_m$;
  overwritten$_t \cup =$ overwritten$_m$;

unlock(Monitor m)
  previous$_m \cup =$ previous$_t$;
  overwritten$_m \cup =$ overwritten$_t$;
  Release/decrement lock on m

readVolatile(Variable v)
  previous$_t \cup =$ previous$_v$;
  overwritten$_t \cup =$ overwritten$_v$;
  return volatileValue$_v$

writeVolatile(Write $\langle v, w, g\rangle$)
  volatileValue$_v = w$
  previous$_v \cup =$ previous$_t$;
  overwritten$_v \cup =$ overwritten$_t$;

Figure 5: Formal semantics without final fields or prescient writes

In order to allow standard compiler optimizations to be performed, we need to allow Prescient Writes. A compiler may move a write earlier than it would be executed by the original program if the following conditions are absolutely guaranteed:

1. The write will happen (with the variable and value written guaranteed as well).
2. The prescient write can not be seen in the same thread before the write would normally occur.
3. Any premature reads of the prescient write must not be observable as a previousRead via synchronization.

When we say that something is guaranteed, this includes the fact that it must be guaranteed over all possible results from improperly synchronized reads (which are non-deterministic, because $|\text{allWrites}(v) - overWrites_t| > 1$). Figure 7 shows an example of a

```
        Initially:
         a = b = 0
Thread 1:        Thread 2:
j = b;           i = a;
a = 1;           b = 1;
```

Can this result in i == j == 1?

Figure 6: Motivation for Prescient Writes

```
        Initially:
          a = 0
Thread 1:        Thread 2:
j = a;           i = a;
a = j;           a = i;
```

Must not result in i == j == 42

Figure 7: Prescient Writes must be Guaranteed

behavior that could be considered "consistent" (in a very perverted sense) if prescient writes were not required to be guaranteed across non-deterministic reads (the value of 42 appears out of thin air in this example).

7.6 Full semantics

In this section, we give the full semantics, including final fields and prescient writes.

7.6.1 New Types and Domains

local A value stored in a stack location or local (e.g., not in a field or array element). A local is represented by a tuple $\langle a, \text{oF}\rangle$, where a is a value (a reference to an object or a primitive value) and oF is a set of writes known to be overwritten due to the special semantics of final fields.

7.6.2 Overview

The semantics of each of the actions is given in Figure 8. The write actions take one parameter: the write to be performed. The freeze actions take one parameter: the final variable to be frozen. The read actions take two parameters: a local that references an object to be read, and an element of that object (field or array element). The lock and unlock actions take one parameter: the monitor to be locked or unlocked.

We use

$$\text{info}_x \cup = \text{info}_y$$

initWrite(Write $\langle v, w, g\rangle$)
    allWrites$+=\langle v, w, g\rangle$
    uncommitted$_t+=\langle v, w, g\rangle$

performWrite(Write $\langle v, w, g\rangle$)
    Assert $\langle v, w, g\rangle \notin$ previousReads$_t$
    overwritten$_t \cup =$ previous$_t(v)$
    previous$_t+=\langle v, w, g\rangle$
    uncommitted$_t-=\langle v, w, g\rangle$

readNormal(Local $\langle a, \text{oF}\rangle$, Element e)
    Let v be the variable referenced by $a.e$
    Choose $\langle v, w, g\rangle$ from allWrites$(v)-$oF
        $-$uncommitted$_t-$ overwritten$_t$
    previousReads$_t+=\langle v, w, g\rangle$
    return $\langle w, \text{oF}\rangle$

readStatic(Variable v)
    Choose $\langle v, w, g\rangle$ from allWrites(v)
        $-$uncommitted$_t-$ overwritten$_t$
    previousReads$_t+=\langle v, w, g\rangle$
    return $\langle w, \emptyset\rangle$

lock(Monitor m)
    Acquire/increment lock on m
    info$_t \cup =$ info$_m$;

unlock(Monitor m)
    info$_m \cup =$ info$_t$;
    Release/decrement lock on m

readVolatile(Local $\langle a, \text{oF}\rangle$, Element e)
    Let v be the volatile referenced by $a.e$
    info$_t \cup =$ info$_v$
    return $\langle \text{volatileValue}_v, \text{oF}\rangle$

writeVolatile(Write $\langle v, w, g\rangle$)
    volatileValue$_v = w$
    info$_v \cup =$ info$_t$;

writeFinal(Write $\langle v, w, g\rangle$)
    finalValue$_v = w$

freezeFinal(Variable v)
    overwritten$_v =$ overwritten$_t$

readFinal(Local $\langle a, \text{oF}\rangle$, Element e)
    Let v be the final variable referenced by $a.e$
    return $\langle \text{finalValue}_v, \text{oF} \cup \text{overwritten}_v\rangle$

Figure 8: Semantics of Program Actions

as shorthand for

$$\begin{aligned} \text{previousReads}_x &\cup= \text{previousReads}_y \\ \text{previous}_x &\cup= \text{previous}_y \\ \text{overwritten}_x &\cup= \text{overwritten}_y \end{aligned}$$

7.6.3 Static variables

Before any reference to a static variable, the thread must insure that the class is initialized. Issues related to class initialization are discussed in the full paper. Because of the semantics of class initialization, no special final semantics are needed for static variables.

7.6.4 Freezing final fields

When a constructor terminates normally, the thread performs freeze actions on all final fields defined in that class. If a constructor A1 for A chains to another constructor A2 for A, the fields are only frozen at the completion of A1. If a constructor B1 for B chains to a constructor A1 for A (a superclass of B), then upon completion of A1, final fields declared in A are frozen, and upon completion of B1, final fields declared in B are frozen.

Associated with each final variable v are

- finalValue$_v$ (the value of v), and
- overwritten$_v$

Every read of a field has to be done using a local $\langle a, oF\rangle$. A read performed through this local cannot return any of the writes in the set oF due to the special semantics of final fields. For each v, overwritten$_v$ is the overwritten$_t$ set of the thread that performed the freeze on v, at the time that the freeze was performed. overwritten$_v$ is assigned when the freeze on v is performed. Whenever a read of v is performed, the tuple returned contains the value of v and the union of overwritten$_v$ with the local's oF set. This implies that the writes in overwritten$_v$ cannot be returned by any read derived from a read of v (condition F2).

The *this* parameter to the run method of a thread has an empty oF set, as done the local generated by a NEW operation.

7.6.5 Semantics of Prescient writes

Each write action is broken into two parts: initWrite and performWrite. The performWrite is always performed at the point where the write existed in the original program. Each performWrite has a corresponding initWrite that occurs before it and is performed on a write tuple with the same GUID. The initWrite can always be performed immediately before the performWrite. The initWrite may be performed prior to that (i.e., presciently) if the write is guaranteed to occur. This guarantee extends over non-deterministic choices for the values of reads.

We must guarantee that no properly synchronized read of the variable being written can be observed between the prescient write and the execution of the write by the original program. To accomplish this, we create a set previousReads(t) for every thread t which contains the set of values of variables that t knows have been read. A read can be added to this set in two ways: if t performed the read, or t has synchronized with a thread that contained the read in its previousReads(t) set.

If a properly synchronized read of the variable were to occur between the initWrite and the performWrite, the read would be placed in the previousReads set of the thread performing the write. We assert that this cannot happen; this maintains the necessary conditions for prescient writes.

The set uncommitted$_t$ contains the set of presciently performed writes by a thread whose performWrite action has not occurred. Writes contained in a thread's uncommitted$_t$ set are invisible to that thread. This set exists to reinforce the fact that the prescient write is invisible to the thread that executed it until the performWrite action. This would be handled by the assertion in performWrite, but making it clear that this is not a choice clarifies what it means for a prescient write to be guaranteed.

Final fields and Prescient writes An initWrite of a reference a must not be reordered with an earlier freeze of a field of the object o referenced by a. This prevents a prescient write from allowing a reference to o to escape the thread before o's final fields have been frozen.

7.6.6 Prescient Reads?

The semantics we have described does not need any explicit form of prescient reads to reflect ordering that might be done by a compiler or processor. The effects of prescient reads are produced by other parts of the semantics.

If a Read action were done early, the set of values that could be returned by the read would just be a subset of the values that could be done at the original location of the Read. So the fact that a compiler or processor might perform a read early, or fulfill a read out of a local cache, cannot be detected and is allowed by the semantics, without any explicit provisions for prescient reads.

7.6.7 Other reorderings

The legality of many other compiler reorderings can be inferred from the semantics. These compiler reorderings could include speculative reads or the delay of a memory reference. For example, in the absence of synchronization operations, constructors and final fields, all memory references can be freely reordered subject to the usual constraints arising in transforming single-threaded code (e.g., you can't reorder two writes to the same variable).

7.6.8 Pseudo-final fields

If a reference to an object with a final field is loaded by a thread that did not construct that object, one of two things should be true:

- That reference was written after the appropriate constructor terminated, or
- synchronization is used to guarantee that the reference could not be loaded until after the appropriate constructor terminated.

If neither of these conditions hold, then the final field of the object immediately becomes a *pseudo-final* field. A read of a pseudo-final field nondeterministically returns either the default value for the type of that field, or the value written to that field in the constructor (if that write has occurred).

Objects can have multiple constructors (e.g., if class B extends A, then a B object has a B constructor and an A constructor). In such a case, if a B object becomes visible to other threads after the A constructor has terminated, but before the B constructor has terminated, then the final fields defined in B become pseudo-final, but the final fields of A remain final.

7.7 Non-atomic longs and doubles

A read of a long or double variable v can return a combination of the first and second half of any two of the eligible values for v. If access to v is properly synchronized, then there will only be one write in the set of eligible values for v. In this case, the new value of v will not be a combination of two or more values (more precisely, it will be a combination of the first half and the second half of the same value). The specification for reads of longs and doubles is shown in Figure 9. The way in which these values might be combined is implementation dependent. This allows machines that do not have efficient 64-bit load/store instructions to implement loads/stores of longs and doubles as two 32-bit load/stores.

```
readNormalLongOrDouble(Value ⟨a, oF⟩, element e)
    Let v be the variable referenced by a.e
    Let v′ and v″ be arbitrary values from allWrites(v) − overwritten_t − oF
    return ⟨ combine(firstPart(v′), secondPart(v″)), oF⟩
```

Figure 9: Formal semantics for longs and doubles

Note that reads and writes of volatile and final long and double variables are required to be atomic.

7.8 Finalizers

Finalizers are executed in an arbitrary thread t that holds no locks at the time the finalizer begins execution. For a finalizer on an object o, overwritten_t is the union of all writes to any field/element of o known to be overwritten by any thread at the time o is determined to be unreachable, along with the overwritten set of the thread that constructed o as of the moment the constructor terminated. The set previous_t is the union of all writes to any field/element of o known to be previous by any thread at the time o is determined to be unreachable, along with the previous set of the thread that constructed o as of the moment the constructor terminated.

It is strongly recommended that objects with nontrivial finalizers be synchronized. The semantics given here for unsynchronized finalization are very weak, but it isn't clear that a stronger semantics could be enforced.

7.9 Related Work

The simple semantics is closely related to Location Consistency [GS98]; the major difference is that in location consistency, an acquire or release affects only a single memory location. However, location consistency is more of an architectural level memory model, and does not directly support abstractions such as monitors, final fields or finalizers. Also, location consistency allows actions to be reordered "in ways that respect dependencies". We feel that our rules for prescient writes are more precise, particularly with regard to compiler transformations.

To underscore the similarity to Location Consistency, the $previous_t(v)$ can be seen to be the same as the set $\{e \mid t \in \text{processorset}(e)\}$ and everything reachable from that set by following edges backwards in the poset for v. Furthermore, the MRPW set is equal to $previous_t(v) - overwritten_t$.

Thread 1:	Thread 2:
`synchronized ( new Object()) { x = 1; }`	`synchronized ( new Object()) { y = 1; }`
`synchronized ( new Object()) { j = y; }`	`synchronized ( new Object()) { i = x; }`

Figure 10: "Useless" synchronization

8 Optimizations

A number of papers [WR99, ACS99, BH99, Bla99, CGS+99] have looked at determining when synchronization in Java programs is "useless", and removing the synchronization. A "useless" synchronization is one whose effects cannot be observed. For example, synchronization on thread-local objects is "useless."

The existing Java thread semantics [GJS96, §17] does not allow for complete removal of "useless" synchronization. For example, in Figure 10, the existing semantics make it illegal to see 0 in both `i` and `j`, while under these proposed semantics, this outcome would be legal. It is hard to imagine any reasonable programming style that depends on the ordering constraints arising from this kind of "useless" synchronization.

The semantics we have proposed make a number of synchronization optimizations legal, including:

1. Complete elimination of lock and unlock operations on a monitor unless more than one thread performs lock/unlock operations on that monitor. Since no other thread will see the information associated with the monitor, the operations have no effect.

2. Complete elimination of reentrant lock/unlock operations (e.g., when a synchronized method calls another synchronized method on the same object). Since no other thread can touch the information associated with the monitor while the outer lock is in effect, any inner lock/unlock actions have no effect.

```
Initially:
p.next = null

Thread 1:
p.next = p

Thread 2:
List tmp = p.next;
if (tmp == p
      && tmp.next == null) {
    // Can't happen under CRF
    }
```

Figure 11: CRF is constrained by data dependences

3. Lock coarsening. For example, given two successive calls to synchronized methods on the same monitor, it is legal simply to perform one Lock, before the first method call, and perform one Unlock, after the second call. This is legal because if no other thread acquired the lock between the two calls, then the Unlock/Lock actions between the two calls have no effect. Note: there are liveness issues associated with lock coarsening, which need to be addressed separately. The Java specification should probably require that if a lower priority thread gives up a lock and a higher priority thread is waiting for a lock on the same object, the higher priority thread is given the lock. For equal priority threads, some fairness guarantee should be made.

4. Replacement of a thread local volatile field (i.e., one accessed by only a single thread) with a normal field. Since no other thread will see the information associated with the volatile, the overwritten and previous information associated with the volatile will not be seen by other threads; since the variable is thread local, all accesses are guaranteed to be correctly synchronized.

5. Forward substitution across lock acquires. For example, if a variable x is written, a lock is acquired, and x is then read, then it is possible to use the value written to x as the value read from x. This is because the lock action does not guarantee that any values written to x by another thread will be returned by a read in this thread if this thread performed an unsynchronized write of x. In general, it is possible to move most operations to normal variables inside synchronized blocks.

```
           Initially:
             a = 0
Thread 1:          Thread 2:
a = 1;             a = 2;
i = a;             j = a;
```

CRF does not allow i == 2 and j == 1

Figure 12: Global memory constraints in CRF

9 Related Work

Maessen et al. [MS00] present an operational semantics for Java threads based on the CRF model. At the user level, the proposed semantics are very similar to those proposed in this paper (due to the fact that we met together to work out the semantics). However, we believe are some troublesome (although perhaps not fatal) issues with that paper.

Perhaps most seriously, the CRF model doesn't distinguish between final fields and non-final fields as far as seeing the writes performed in a constructor. As discussed in [MS00, §6.1], they rely on memory barriers at the end of constructors to order the writes and data dependences to order the reads. This means that in Figure 1*b*, their semantics prohibit `r3 == 0`, even though the `x` field is not final. Since this guarantee requires additional memory barriers on systems using the Alpha memory model, it is undesirable to make it for non-final fields.

Another problem is that [MS00] does not allow as much elimination of "useless synchronization". The CRF-based specification provides a special rule to allow skipping coherence actions associated with a monitorenter if the thread that previously released the lock is the same thread as the current thread. However, no such rule applies to monitorexit. As a result, in Figure 10 it is illegal to see 0 in both `i` and `j`. Also, their model doesn't provide any "coherence-skipping" rule for volatiles, so memory barriers must be associated with thread-local volatile fields. Also, while the CRF semantics allow skipping the memory barrier instructions associated with monitorenter on thread local monitors, it isn't clear that it allows compiler reordering past thread-local synchronization.

In contrast, under our model most synchronization optimizations, such as removal of "useless synchronization", fall out naturally as a consequence of using a *lazy release consistency* [CZ92] style semantics.

Furthermore, the handling of control and data dependences is worrisome. Speculative reads are represented by moving Load instructions earlier in execution. However, for an operational semantics, it is hard to imagine executing a Load instruction before you know the address that needs to be Loaded. In fact, they specifically prohibit it [MS00, §6.1] in order to get the required semantics for final fields.

For example, the code in Figure 11 shows a behavior prohibited by CRF. Since the read of `tmp.next` is data dependent on the read of `p.next`, it must follow the read of `p.next`.

While it is hard to imagine a compiler transformation or processor architecture in which this reordering could occur, it none the less imposes a proof burden: showing that any implementation does not allow this reordering which is not allowed by CRF.

Similarly, because CRF models a single global memory through which all communication is performed, certain behaviors are prohibited. For example, in Figure 12 it is prohibited that i = 2 and j = 1. This prohibition has nothing to do with safety guarantees or execution of correctly synchronized programs. Rather, it is just an artifact of the CRF model. An implementation of Java on an aggressive SMP architecture that allowed this behavior would not correctly adhere to these semantics.

10 Conclusion

We have proposed both an informal and formal memory model for multithreaded Java programs. This model will both allow people to write reliable multithreaded programs and give JVM implementors the ability to create efficient implementations.

It is essential that a compiler writer understand what optimizations and transformations are allowed by a memory model. Ideally, in code that doesn't contain synchronization operations, all the standard compiler optimizations would be legal. In fact, no proof of this could be forthcoming because there are a very few standard optimizations that are not legal. In particular, in a single-threaded environment, if you prove there are no writes to a variable between two reads, you can assume that both reads return the same value, and possibly omit some bounds checking or null-pointer checks that would otherwise be required. In a multithreaded setting, no such causal assumptions can be made.

However, the process of understanding and documenting the interactions between the memory model and optimizations is of vital importance and will be the focus of continuing work.

Now that a broad community has reached rough consensus on an informal semantics for multithreaded Java, the important step now is to formalize that model. Doing so requires figuring out all of the corner cases, and providing a framework that would allow formal reasoning about the model. We believe that this proposal both provides the guarantees needed by Java programmers and the freedoms needed by JVM implementors.

Acknowledgments

Thanks to the many people who have participated in the discussions of this topic, particularly Sarita Adve, Arvind, Joshua Bloch, Joseph Bowbeer, David Detlefs, Sanjay Ghemawat, Paul Haahr, Doug Lea, Tim Lindholm, Jan-Willem Maessen, Xiaowei Shen, Raymie Stata, Guy Steele and Dennis Sosnoski.

References

[ACS99] Jonathan Aldrich, Craig Chambers, and Emir Gun Sirer. Eliminating unnecessary synchronization from java programs. In *OOPSLA poster session*, October 1999.

[BH99] Jeff Bogda and Urs Hoelzle. Removing unnecessary synchronization in java. In *OOPSLA*, October 1999.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages; application to Java. In *OOPSLA*, October 1999.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA*, October 1999.

[CZ92] Pete Keleher Alan L. Cox and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *The Proceedings of the 19 th International Symposium of Computer Architecture*, pages 13–21, May 1992.

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

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, , and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GS98] Guang Gao and Vivek Sarkar. Location consistency – a new memory model and cache consistency protocol. Technical Report 16, CAPSL, Univ. of Deleware, February 1998.

[JMM] The Java memory model. Mailing list and web page. http://www.cs.umd.edu/ ~pugh/ java/ memoryModel.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 2nd edition, 1999.

[MP01] Jeremy Manson and William Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.

[MS00] Arvind Jan-Willem Maessen and Xiaowei Shen. Improving the Java memory model using CRF. In *OOPSLA*, pages 1–12, October 2000.

[Pug99] William Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.

[Pug00a] William Pugh. The double checked locking is broken declaration. http://www.cs.umd.edu/ users/ pugh/ java/ memoryModel/ DoubleCheckedLocking.html, July 2000.

[Pug00b] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1–11, 2000.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, October 1999.

A Class Initialization

The JVM specification requires [LY99, §5.5] that before executing a GETSTATIC, PUTSTATIC, INVOKESTATIC or a NEW instruction on a class C, or initializing a subclass of C, class C must be initialized. Furthermore, class C may not be initialized before it is required by the above rule.

Although the JVM specification does not spell it out, it is clear that any situation that requires that a thread T1 check to see that a class C has been initialized must also require that T1 see all of the memory actions resulting from the initialization of class C.

This has a number of subtle and surprising implications for compilation, and interactions with the threading model.

Initializing a class invokes the static initializer for the class, which can be arbitrary code. Thus, any GETSTATIC, PUTSTATIC, INVOKESTATIC or a NEW instruction on a class C, which might be the very first invocation of an instruction on class C, must be treated as a potential call of the initialization code. Thus, if A and B are classes, the expression A.x+B.y+A.x cannot always be optimized to A.x*2+B.y; the read of B.y may have side effects that change the value of A.x (because it might invoke the initialization code for B that could modify A.x).

It would be possible to perform static analysis to verify that a particular instruction could not possibly be the first time a thread was required to check that a class was initialized. Also, you could check that the results of initializing a class were not visible outside the class. Either analysis would allow the instruction to be reordered with other instructions.

A quick reading of the spec might suggest that a thread can simply check a boolean flag to see if the class is initialized, and skip initialization code if the class is already initialized. This is almost true. However, the thread checking to see that the class is initialized must see all updates caused by initializing the class. This may require flushing registers and performing a memory barrier.

Similarly, once a xxxSTATIC or NEW instruction has been invoked, it is tempting to rewrite the code to eliminate the initialization check. However, this rewrite cannot be done until all threads have done the barrier required to see the effects of initializing the class.

Another surprising result is that the existing spec allows a thread to invoke methods and read/write instance fields of an instance of a class C before seeing all of the effects of the initialization of that class. How could this happen? Consider if thread T1 initializes class C, creates an instance x of class C, and then stores a reference to the instance into a global variable. Thread T2 could then, without synchronization, read the global variable, see the reference to x, and invoke a virtual method on x. At this point, although C has been initialized, T2 hasn't done the memory barrier or register flushes that would be required to see the updates performed by initializing class C. This means that *even within* virtual methods of class C, we can't automatically eliminate/skip initialization checks associated with GETSTATIC, PUTSTATIC, INVOKESTATIC or NEW instructions on a class C.

Classes can also be initialized due to use of reflection or by being designated as the initial class of the JVM.