Requirements for Programming Language Memory Models

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
{jmanson, pugh}@cs.umd.edu

ABSTRACT

One of the goals of the designers of the Java programming language was that multithreaded programs written in Java would have consistent and well-defined behavior. This would allow Java programmers to understand how their programs might behave; it would also allow Java platform architects to develop their platforms in a flexible and efficient way, while still ensuring that Java programs ran on them correctly.

Unfortunately, Java's original *memory model*, which described the way in which Java threads interact through memory, was not defined in a way that allowed programmers and architects to understand the requirements for a Java system. As part of Java Specification Request (JSR) 133 [7], a new memory model has been defined for Java. This paper outlines how the requirements for a new memory model were established, and what those requirements are. It does not outline the model itself; it merely provides a rationale.

1. INTRODUCTION

The work in [13, 14] showed that the original semantics for Java's threading specification [6, §17] had serious problems. To address these issues, the Java programming language [6] has recently undergone a revision; it now provides greater flexibility for implementors and a clearer notion of what it means to write a correct program. The new specification is widely known as the *Java memory model*.

To provide a clearer semantics, the informal properties of the memory model had to be described. This was accomplished through a great deal of thinking, staring at white boards, and spirited debate. A careful balance had to be maintained. On one hand, it was necessary for the model to allow programmers to be able to reason carefully and correctly about their multithreaded code. On the other, it was necessary for the model to allow compiler writers, virtual machine designers and hardware architects to optimize code ruthlessly, possibly interfering with the intuitive results of a program.

At the end of this process, a consensus emerged as to what the informal requirements for a programming language memory model are. In this paper, we discuss these requirements in detail. We do not discuss how these requirements were met. For more details on the actual model, see [7].

For a more detailed record of the process of designing this memory model, it might be instructive for the reader to look at the Java memory model mailing list archives [8].

2. WHY A SEMANTICS?

In the past, multithreaded languages have not defined a full semantics for multithreaded code. Ada, for example, simply defines unsynchronized code as "erroneous" [1]. The reasoning behind this is that since such code is incorrect (on some level), no guarantees should be made when it occurs. What it means for code to be correctly synchronized should be fully defined; after that, nothing.

This is the same strategy that some languages take with array bounds overflow – unpredictable results may occur, and it is the programmer's responsibility to avoid these scenarios.

The problem with this strategy is one of security and safety. In an ideal world, all programmers would write correct code all of the time. However, this does not always happen. Programs frequently contain errors; not only does this cause code to misbehave, but it also allows attackers an easy way into a program. Buffer overflows, in particular, are frequently used to compromise a program's security. Program semantics must be carefully defined: otherwise, it becomes harder to track down errors, and easier for attackers to take advantage of those errors. If programmers don't know what their code is doing, programmers won't be able to know what their code is doing wrong.

The new Java memory model provides strong guarantees for correctly written code, but also provides a clear and definitive semantics for how code should behave when it is not correctly written.

3. SIMPLE REORDERING

Many of the most important optimizations that can be performed on a program involve reordering program statements. For example, superscalar architectures frequently reorder instructions to ensure that the execution units are all in use as much as possible. Even optimizations as ubiquitous as common subexpression elimination and redundant read elimination can be seen as reorderings: each evaluation of the common expression is conceptually "moved" to the point at which it is evaluated for the first time.

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

May return r2 == 2, r1 == 1

Figure 1: Behaves Surprisingly

In a single threaded program, a compiler can (and, indeed, must) be careful that these program transformations not interfere with the possible results of the program. We refer to this as a compiler's maintaining of the *intra-thread semantics* of the program – a thread in isolation has to behave as if no code transformations occurred at all.

However, it is much more difficult to maintain a simple, straightforward semantics while optimizing multithreaded code. Consider Figure 1. It may appear that the result `r2 == 2, r1 == 1` is impossible. Intuitively, if `r2` is 2, then instruction 4 came before instruction 1. Further, if `r1` is 1, then instruction 2 came before instruction 3. So, if `r2 == 2` and `r1 == 1`, then instruction 4 came before instruction 1, which comes before instruction 2, which came before instruction 3, which comes before instruction 4. This is a cyclic execution, which is, on the face of it, absurd.

On the other hand, we must consider the fact that a compiler can reorder the instructions in each thread. If instruction 3 does not come before instruction 4, and instruction 1 does not come before instruction 2, then the result `r2 == 2` and `r1 == 1` is perfectly reasonable.

In fact, in Java, for performance reasons, we always allow actions that are not control or data dependent on each other in a program to be reordered. This leads us to our first requirement:

Reorder1 Independent actions can be reordered.

In a multithreaded context, doing this may lead to counter-intuitive results, like the one in Figure 1. However, it should be noted that this code is improperly synchronized: there is no ordering of the accesses by synchronization. When synchronization is missing, weird and bizarre results are allowed.

It should be noted that Reorder1 guarantees that independent actions can be reordered regardless of the order in which they appear in the program. It does **not** guarantee that two independent actions can always be reordered. For example, a write action clearly cannot be reordered out of a locking region. We shall see another example of how these reorderings are limited in Section 5.4.

4. GUARANTEES FOR CORRECTLY SYNCHRONIZED PROGRAMS

It is very difficult for programmers to reason about the kinds of transformations that compilers perform. One of the goals of the Java memory model is to provide programmers a mechanism that allows them not to have to reason about reorderings in a program.

For example, in the code in Figure 1, the programmer can only see the result of the reordering because the code is improperly synchronized. Our first goal is to ensure that this is the only reason that a programmer can see the result of a reordering.

We say that a program obeys *sequentially consistent semantics* (as defined in [10]) if the result of any execution

Initially, `x == y == 0`

`Thread 1`	`Thread 2`
`r1 = x;`	`r2 = y;`
`if (r1 != 0)`	`if (r2 != 0)`
`  y = 42;`	`  x = 42;`

Correctly synchronized, so `r1 == r2 == 0` is the only legal behavior

Figure 2: Surprising Correctly Synchronized Program

is the same as if all of the actions in that execution took place in some total order that reflects the order of the program, and each read sees the last write to that variable that occurred in that order. If a program obeys sequentially consistent semantics, then no compiler or processor reorderings will be visible.

Two accesses (reads of or writes to) the same shared field or array element are said to be *conflicting* if at least one of the accesses is a write. A *data race* occurs in an execution of a program if there are conflicting actions in multiple threads in that execution that are not ordered by synchronization. The program in Figure 1 has data races on both `x` and `y`. A program is *correctly synchronized* if and only if all sequentially consistent executions are free of data races.

Having defined these terms, we can talk a little more about the guarantees we provide. One possibility would be that we could eliminate all reorderings. On contemporary systems, this would have too much of an impact on performance. However, it is perfectly reasonable to ensure that code reordering should only be visible between threads when those threads are involved in data races. Our first guarantee for programmers, therefore, applies to data-race-free programs:

DRF Correctly synchronized programs have sequentially consistent semantics.

Given this requirement, programmers need only worry about code transformations having an impact on their programs' results if those program contain data races.

This requirement leads to some interesting corner cases. For example, the code shown in Figure 2 (first described in [2]) is correctly synchronized. This may seem surprising, since it doesn't perform any synchronization actions. Remember, however, that a program is correctly synchronized if, when it is executed in a sequentially consistent manner, there are no data races. If this code is executed in a sequentially consistent way, each action will occur in program order, and neither of the writes will occur. Since no writes occur, there can be no data races: the program is correctly synchronized. A program transformation (such as an aggressive write speculation) that permitted this result would be disallowed.

5. SYNCHRONIZATION

We haven't really discussed how code can use explicit synchronization (in whatever form we give it) to make sure their code is correctly synchronized. The typical way of doing this is by using locking. Another way is to use *volatile* variables.

The properties of volatile variables arose from the need to provide a way to communicate between threads without the overhead of ensuring mutual exclusion. A very simple

Initially, `x == 0, ready == false`. `ready` is a volatile variable.

Thread 1	Thread 2
`x = 1;`	`if (ready)`
`ready = true`	`r1 = x;`

If `r1 = x;` executes, it will read 1.

Figure 3: Simple Use of Volatile Variables

example of their use can be seen in Figure 3. If `ready` were not volatile, the write to it in Thread 1 could be reordered with the write to `x`. This might result in `r1` containing the value 0. We define volatiles so that this reordering cannot take place; if Thread 2 reads `true` for `ready`, it must also read 1 for `x`.

Locks and unlocks work in a way similar to volatiles: actions that take place before an unlock must also take place before any subsequent locks on that monitor. The resulting property reflects the way synchronization is used to communicate between threads:

HB Volatile writes are ordered before subsequent volatile reads of the same variable. Unlocks are ordered before subsequent locks of the same monitor.

The word *subsequent* needs to be defined for HB. *Synchronization actions* include locks, unlocks, and reads of and writes to volatile variables. We have a total order over all synchronization actions in an execution of a program; this is called the *synchronization order*. An action `y` is subsequent to another action `x` if `x` comes before `y` in the synchronization order.

5.1 Happens-Before Consistency

We can describe a simple, interesting memory model using HB by abstracting a little from locks and unlocks.

A *happens-before relationship* between two actions is what enforces an ordering between those actions. For example, if one action occurs before another in the program order for a single thread, then the first action happens-before the second. The program has to be executed in a way that does not make it appear to the second that it occurred out of order with respect to the first.

This may seem at odds with the result in Figure 1. However, a "reordering" is only visible here if we assume that the program executed all of its actions in a single total order; the surprising behavior makes it appear as if the writes are occurring before the reads. If the individual threads are examined in isolation, no reordering is visible; it is simply not known where the values seen by the reads are written.

The basic principle at work here is that threads in isolation will appear to behave as if they are executing in program order; however, the memory model will tell you what values can be seen by a particular read;

Synchronization actions can create happens-before relationships between threads. In addition to the happens-before relationship between actions in a single thread, we also have (in accordance with HB)

- An unlock on a particular monitor happens-before a lock on that monitor that comes after it in the synchronization order.
- A write to a volatile variable happens-before a read of that volatile variable that comes after it in the synchronization order.
- A call to start a thread happens-before the actual start of that thread.
- The termination of a thread happens-before a join performed on that thread.
- Happens-before is transitive. That is, if `a` happens-before `b`, and `b` happens-before `c`, then `a` happens-before `c`.

We say that it is *happens-before consistent* for a read to see a write in an execution of a program in two cases. First, a read is happens-before consistent if the write happens-before the read and there is no intervening write to the same variable. So, if a write of 1 to `x` happens-before a write of 2, and the write of 2 happens-before a read, then that read cannot see the value 1. Second, it is happens-before consistent for the read to see the write if the write does not happen-before the read. If the read does not happen-before the write, then the read is allowed to see the write. This can happen, for example, if the write occurs in another thread (as in Figure 1).

If all of the reads in an execution see writes they are happens-before consistent to see, then we say that execution is happens-before consistent. Note that happens-before consistency implies that every read must see a write that occurs somewhere in the program.

Although it is simple, happens-before consistency is not a good memory model. Notice that the behavior we want to disallow in Figure 2 is happens-before consistent. If both writes occur, and both reads see them, then both reads see writes that they are allowed to see.

Nevertheless, happens-before consistency provides a good outer bound for our model; based on HB, all executions must be happens-before consistent. Later sections of this paper (mostly Section 7) discuss ways of locating a more exact bound; for now, we focus on how happens-before affects implementation.

5.2 Implementing Synchronization

At the abstract level, happens-before consistency provides a relatively simple memory model. In this section, we talk a little about how we implement happens-before guarantees.

A happens-before relationship can be thought of as an ordering edge with two points; we call the start point a *release*, and the end point an *acquire*. Unlocks and volatile writes are release actions, and locks and volatile reads are acquire actions.

An acquire ensures an ordering with a previous release. Consider an action that takes place before an acquire. It may or may not have been visible to actions that took place before the previous release, depending on how the threads are scheduled. If we move the access to after the acquire, we are simply saying that the access is definitely scheduled after the previous release. This is therefore a legal transformation. For example, in Figure 3, if there were a read of a normal variable that occurred before the read of `ready`, then it could be moved after the read of `ready`.

Similarly, the only thing that the release does is ensure an ordering with a subsequent acquire. Consider an action that takes place after a release. It may or may not be visible to

Initially, `v1 == v2 == 0`

Thread 1	Thread 2	Thread 3	Thread 4
`v1 = 1;`	`v2 = 2;`	`r1 = v1;`	`r3 = v2;`
		`r2 = v2;`	`r4 = v1;`

Is `r1 == r3 == 1, r2 == r4 == 0` legal behavior?

Figure 4: Volatiles Must Occur In A Total Order

Initially, `x == y == v == 0`, `v` is volatile.

Thread 1	Thread 2
r1 = x;	r3 = y;
v = 0;	v = 0;
r2 = v;	r4 = v;
y = 1;	x = 1;

Is the behavior `r1 == r3 == 1` possible?

Figure 5: Strong or Weak Volatiles?

Initially, `a == b == v == 0`, `v` is volatile.

Thread 1	Thread 2
`r1 = a;`	`do {`
`if (r1 == 0)`	`  r2 = b;`
`  v = 1;`	`  r3 = v;`
`else`	`} while (r2 + r3 < 1);`
`  b = 1;`	`a = 1;`

Correctly synchronized, so `r1 == 1` is illegal

Figure 6: Another Surprising Correctly Synchronized Program

particular actions after the subsequent acquire, depending on how the threads are scheduled. If we move the access to before the release, we are simply saying that the access is definitely scheduled before the next acquire. This is therefore also a legal transformation. For example, in Figure 3, if there were a write to a normal variable that occurred after the write to `ready`, then it could be moved before the write to `ready`.

All of this is simply a roundabout way of saying that accesses to normal variables can be reordered with a following volatile read or monitor enter, or a preceding volatile write or monitor exit, This implies that normal accesses can be moved inside locking regions, but not out of them; for this reason, we sometimes call this property *roach motel semantics*.

It is relatively easy for compilers to ensure this property; indeed, most do already. Processors, which also reorder instructions, often need to be given *memory barrier* instructions to execute at these points in the code to ensure that they do not perform the reordering. Processors often provide a wide variety of these barrier instructions – for information about which are needed on which processor and for which action, consult [11].

5.3 Additional Guarantees for Volatiles

Figure 4 gives us another interesting glimpse into the guarantees we provide to programmers. The reads of `v1` and `v2` should be seen in the same order by both Thread 3 and Thread 4. The memory model does not allow writes to volatiles to be seen in different orders by different threads. In fact, it makes a much stronger guarantee:

VolatileAtomicity All accesses to volatile variables are performed in a total order.

This is clear cut, implementable, and has the unique property that the original Java memory model not only came down on the same side, but was also clear on the subject.

Another issue that arises with volatiles has come to be known as *strong versus weak* volatility. There are two possible interpretations of volatile, according to the happens-before order:

- **Strong interpretation** There is a happens-before relationship from each write to each subsequent read of that volatile.
- **Weak interpretation** There is a happens-before relationship from each write to each subsequent read of that volatile that sees that write.

In Figure 5, under the weak interpretation, the read of `v` in each thread might see its own volatile write. If this were the case, then the happens-before edges would be redundant, and could be removed. The resulting code could behave much like the simple reordering example in Figure 1.

To avoid confusion stemming from when multiple writer threads are communicating to reader threads via a single volatile variable, Java supports the strong interpretation.

StrongVolatile There must be a happens-before relationship from each write to each subsequent read of that volatile.

5.4 Optimizers Must Be Careful

Optimizers have to consider volatile accesses as carefully as they consider locking. In Figure 6, we have a correctly synchronized program. When executed in a sequentially consistent way, Thread 2 will loop until Thread 1 writes to `v` or `b`. Since the only value available for the read of `a` to see is 0, `r1` will have that value. As a result, the value 1 will be written to `v`, not `b`. There will therefore be a happens-before relationship between the read of `a` in Thread 1 and the write to `a` in Thread 2.

Knowing that the write to `a` will always happen, we might want to apply the principle that we can reorder the write to `a` with the loop. In this case, Thread 1 would be able to see the value 1 for `a`, and write to `b`. Thread 2 would see the write to `b` and terminate the loop. Since `b` is not a volatile variable, there would be no ordering between the read in Thread 1 and the write in Thread 2. There would therefore be data races on both `a` and `b`.

The result of this would be a correctly synchronized program that does not behave in a sequentially consistent way. This violates DRF, so we do not allow it. The need to prevent this sort of reordering caused many difficulties in formulating a workable memory model.

Compiler writers need to be very careful when reordering code past all synchronization points, not just those involving locking and unlocking.

Before compiler transformation

Initially, `a = 0, b = 1`

Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

Is `r1 == r2 == r3 == 2` possible?

After compiler transformation

Initially, `a = 0, b = 1`

Thread 1	Thread 2
4: b = 2;	5: r3 = b;
1: r1 = a;	6: a = r3;
2: r2 = r1;	
3: if (true) ;	

`r1 == r2 == r3 == 2` is sequentially consistent

Figure 7: Effects of Redundant Read Elimination

5.5 Optimizations Based on Happens-Before

Notice that lock and unlock actions only have happens-before relationships with other lock and unlock actions **on the same monitor**. Similarly, accesses to a volatile variable only create happens-before relationships with accesses to the same volatile variable.

There have been many optimizations proposed (for example, in [15]) that have tried to remove excess, or "redundant" synchronization. One of the requirements of the Java memory model was that redundant synchronization (such as locks that are only accessed in a single thread) could be removed.

One possible memory model would require that all synchronization actions have happens-before relationships with all other synchronization actions. If we forced all synchronization actions to have happens-before relationships with each other, none of them could ever be described as redundant – they would all have to interact with the synchronization actions in other threads, regardless of what variable or monitor they accessed. Java does not support this; it does not simplify the programming model sufficiently to warrant the additional synchronization costs.

This is therefore another of our guarantees:

RS Synchronization actions that only introduce redundant happens-before edges can be treated as if they don't introduce any happens-before edges.

This is reflected in the definition of happens-before. For example, a lock that is only accessed in one thread will only introduce happens-before relationships that are already captured by the program order edges. This lock is redundant, and can therefore be removed.

6. TRANSFORMATIONS THAT INVOLVE DEPENDENCIES

In Section 3, we gave Reorder1, which is a guarantee that independent actions can be reordered. Reorder1 is a strong guarantee, but not quite strong enough. Sometimes, compilers can perform transformations that have the effect of removing dependencies.

For example, the behavior shown in Figure 7 is allowed. The compiler should be allowed to

- eliminate the redundant read of `a`, replacing `r2 = a` with `r2 = r1`, then
- determine that the expression `r1 == r2` is always true, eliminating the conditional branch `3`, and finally
- move the write `4: b = 2` early.

Initially, `x == y == 0`

Thread 1	Thread 2
r1 = x;	r3 = y;
r2 = 1 + r1*r1 - r1;	x = r3;
y = r2;	

`r1 == r2 == r3 == 1` is legal behavior

Figure 8: Compilers Can Think Hard About When Actions Are Guaranteed to Occur

Initially, `x == y == 0`

Thread 1	Thread 2
r1 = x;	r3 = y;
if (r1 == 1)	if (r2 == 1)
y = 1;	x = 1;
	if (r2 == 0)
	x = 1;

`r1 == r2 == 1` is legal behavior

Figure 9: Sometimes Dependencies are not Obvious

After the compiler does the redundant read elimination, the assignment `4: b = 2` is guaranteed to happen; the second read of `a` will always return the same value as the first. Without this information, the assignment seems to cause itself to happen. With this information, there is no dependency between the reads and the write. Thus, dependence-breaking optimizations can also lead to apparent cyclic executions.

Note that intra-thread semantics guarantee that if `r1` $\neq$ `r2`, then Thread 1 will not write to `b` and `r3 == 1`. Additionally, either `r1 == 0, r2 == 1`, or `r1 == 1, r2 == 0`.

Figure 8 shows another surprising behavior. In order to see the result `r1 == r2 == 1`, it would seem as if Thread 1 would need to write 1 to `y` before reading `x`. However, it also seems as if Thread 1 can't know what value `r2` will be until after `x` is read.

In fact, it is easy for the compiler to perform an inter-thread analysis that shows that only the values 0 and 1 will be written to `x`. Knowing that, the compiler can determine that the quadratic equation always returns 1, resulting in Thread 1's always writing 1 to `y`. Thread 1 may, therefore, write 1 to `y` before reading `x`. The write to `y` is not dependent on the values seen for `x`. Our analysis of the program reveals that there is no real dependency in Thread 1.

A similar example of an apparent dependency can be seen in the code in Figure 9. In the same way as it does for Figure 8, a compiler can determine that only the values 0 and 1 are ever written to `x`. As a result, the compiler can remove

Initially, `x = 0`

Thread 1	Thread 2
`r1 = x;`	`r2 = x;`
`x = 1;`	`x = 2;`

`r1 == 2` and `r2 == 1` is a legal behavior

Figure 10: An Unexpected Reordering

Initially, x == y == 0

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = r2;`

Incorrectly Synchronized: But `r1 == r2 == 42` Still Cannot Happen

Figure 11: An Out Of Thin Air Result

the dependency and move the write to `x` to the beginning of Thread 2. If the resulting code were executed in a sequentially consistent way, it would result in the circular behavior described.

It is clear, then, that compilers can perform many optimizations that remove dependencies. So we make another guarantee:

Reorder2 If a compiler can detect that an action will always happen (with the same value written to the same variable), it can be reordered regardless of apparent dependencies.

Like Reorder1, this guarantee does not allow an implementation to reorder actions around synchronization actions arbitrarily. In Figure 6, for example, we saw an example of this: we could not reorder the accesses to `a` because of the happens-before relationships.

Even though Reorder1 and Reorder2 are strong guarantees for compilers, they are not a complete set of reorderings allowed. They are simply a set that is always guaranteed to be allowed.

6.1 Reordering Not Visible to Current Thread

Figure 10 contains a small but interesting example. The behavior `r1 == 2` and `r2 == 1` is a legal behavior, although it may be difficult to see how it could occur. A compiler would not reorder the statements in each thread; this code must never result in `r1 == 1` or `r2 == 2`. However, the behavior `r1 == 2` and `r2 == 1` might be allowed by an optimizer that performs the writes early, but does so without allowing them to be visible to local reads that came before them in program order. This behavior, while surprising, is allowed by several processor memory architectures, and therefore is one that should be allowed by a programming language memory model.

7. OUT-OF-THIN-AIR GUARANTEES

In Figure 2, the writes are control dependent on the reads. Figure 11 is a very similar example; in this case, the writes will always happen, but the values written are data dependent on the reads.

This is no longer a correctly synchronized program, because there is a data race between Thread 1 and Thread 2. However, as it is in many ways a very similar example, we would like to provide a similar guarantee. In this case, we say that the value 42 cannot appear *out of thin air*.

In fact, the behavior of this case may be even more of a cause for concern than the other. If, for example, the value that was being produced out of thin air was a reference to an object which the thread was not supposed to have, then such a transformation could be a serious security violation. There are no reasonable compiler transformations that produce this result.

An example of this can be seen in Figure 12. Let's assume that there is some object `o` which we do not wish Thread 1 or Thread 2 to see. `o` has a self-reference stored in the field `f`. If our compiler were to decide to perform an analysis that assumed that the reads in each thread saw the writes in the other thread, and saw a reference to `o`, then `r1 = r2 = r3 = o` would be a possible result. The value did not spring from anywhere – it is simply an arbitrary value pulled out of thin air.

Determining what constitutes an out-of-thin-air read is complicated. A first (but inaccurate) approximation would be that we don't want reads to see values that couldn't be written to the variable being read in some sequentially consistent execution. Because the value 42 is never written in Figure 11, no read can ever see it.

The problem with this solution is that a program can contain writes whose program statements don't occur in any sequentially consistent executions. Imagine, as an example, a write that is only performed if the value of `r1 + r2` is equal to 3 in Figure 1. This write would not occur in any sequentially consistent execution, but we would still want a read to be able to see it.

One way to think about these issues is to consider when actions can occur in an execution. These transformations all involve moving actions earlier than they would otherwise have occurred. You can perform an action earlier in an execution than it would have otherwise occurred if, had we carried on the execution in a sequentially consistent way, it would have been possible for the action to have occurred afterward.

If we had, for example, a write that was control dependent on the value of `r1 + r2` being equal to 3 in Figure 1, we would know that write could have occurred in an execution of the program that behaves in a sequentially consistent way after the result of `r1 + r2` is determined.

We can apply this form of reasoning to our other example, as well. In Figure 1, the writes to `x` and `y` can occur first because they will always occur in sequentially consistent executions. In Figure 7, the write to `b` can occur early because it occurs in a sequentially consistent execution when `r1` and `r2` see the same value. In Figure 11, the writes of 42 to `y` and `x` cannot happen, because they do not occur in any sequentially consistent execution. This, then, is our first "out of thin air" guarantee:

ThinAir1 A write can occur earlier in an execution than it would have otherwise occurred. However, that write must have been able to occur without the assumption that any reads that take place after the point where the write occurs see non-sequentially consistent values.

7.1 When Actions Can Occur

7.1.1 Disallowing Some Results

It is difficult to define the boundary between the kinds of results that are reasonable and the kind that are not. The example in Figure 11 provides an example of a result

Initially, x = null, y = null.
o is an object with a field f that refers to o.

Thread 1	Thread 2
r1 = x;	r3 = y;
r2 = x.f;	x = r4;
y = r2;	

r1 == r2 == o is not an acceptable behavior

Figure 12: An Unexpected Reordering

Initially, x == y == z == 0

Thread 1	Thread 2
r3 = x;	r2 = y;
if (r3 == 0)	x = r2;
x = 42;	
r1 = x;	
y = r1;	

r1 == r2 == r3 == 42 is a legal behavior

Figure 15: A Complicated Inference

that is clearly unacceptable, but other examples may be less straightforward.

The examples in Figures 13 and 14 are similar to the examples in Figures 2 and 11, with one major distinction. In those examples, the value 42 could never be written to x in any sequentially consistent execution. In the examples in Figures 13 and 14, 42 can be written to x in some sequentially consistent executions. Could it be legal for the reads in Threads 1 and 2 to see the value 42 even if Thread 4 does not write that value?

This is a potential security issue. Consider what happens if, instead of 42, we write a reference to an object that Thread 4 controls, but does not want Threads 1 and 2 to see without Thread 4's first seeing 1 for z. If Threads 1 and 2 see this reference, they can be said to manufacture it out of thin air.

This sort of behavior is not known to result from any combination of known reasonable and desirable optimizations. However, there is also some question as to whether this reflects a real and serious security requirement. In Java, the semantics usually side with the principle of having safe, simple and unsurprising semantics when possible. Thus, the Java Memory Model prohibits the behaviors shown in Figures 13 and 14.

7.1.2 Allowing Other Results

Now consider the code in Figure 15. A compiler could determine that the only values ever assigned to x are 0 and 42. From that, the compiler could deduce that, at the point where we execute r1 = x, either we had just performed a write of 42 to x, or we had just read x and seen the value 42. In either case, it would be legal for a read of x to see the value 42. By the principle we articulated as Reorder2, it could then change r1 = x to r1 = 42; this would allow y = r1 to be transformed to y = 42 and performed earlier, resulting in the behavior in question.

This is a reasonable transformation that needs to be balanced with the out-of-thin-air requirement. Notice that the code in Figure 15 is quite similar to the code in Figures 13 and 14. The difference is that Threads 1 and 4 are now joined together; in addition, the write to x that was in Thread 4 is now performed in every sequentially consistent execution – it is only when we try to get non-sequentially

Initially, x = y = 0; a[0] = 1, a[1] = 2

Thread 1	Thread 2
r1 = x;	r3 = y;
a[r1] = 0;	x = r3;
r2 = a[0];	
y = r2;	

r1 == r2 == r3 == 1 is unacceptable

Figure 16: Another Out Of Thin Air Example

consistent results that the write does not occur.

There is a significant difference between these two cases. One way of articulating it is that in Figure 15, we know that r1 = x can see 42 without reasoning about what might have occurred in another thread because of a data race. In Figures 13 and 14, we need to reason about the outcome of a data race to determine that r1 = x can see 42.

This is, then, what differentiates out of thin air reads from those that are allowable. A solution must be available that does not involve reasoning about what happens in the execution solely because of data races. This is also our second out of thin air principle:

ThinAir2 Actions may only be performed earlier than their original place in the program if it can be determined that they could occur in the execution without assuming that any additional reads see values via a data race.

We can use ThinAir2 as a basic principle to reason about multithreaded programs. Consider, for example, the code in Figure 16. The only way in which the unacceptable result could occur is if a write of 1 to one of the variables were performed early. However, we cannot reason that a write of 1 to x or y will occur without reasoning about data races. Therefore, this result is impossible.

7.2 Isolation

Sometimes, when debugging a program, we are given an execution trace of that program in which the error occurred. Given a particular execution of a program, the debugger can create a *partition* of the threads and variables in the program so that if a thread accessed a variable in that execution, then the thread and variable are in the same partition. Monitors can be included in with variables for the purposes of this discussion.

Given this partitioning, you can explain the behavior in the execution of the threads in each partition without having to examine the behavior or code for the other threads. If a thread or a set of threads is isolated from the other threads in an execution, the programmer can reason about that isolated set separately from the other threads. This is called the *isolation* principle:

Isolation Consider a partition P of the threads and variables in the program so that if a thread accessed a variable in that execution, then the thread and variable are in the same partition. Given P, you can explain the behavior in the execution of the threads in each partition without having to examine the behavior or code for the other threads.

How is this helpful? Consider the code in Figure 14. If we allowed the unacceptable execution, then we could say

Initially, x == y == z == 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x;	r2 = y;	z = 42;	r0 = z;
y = r1;	x = r2;		x = r0;

Is r0 == 0, r1 == r2 == 42 legal behavior?

Figure 13: Can Threads 1 and 2 see 42, if Thread 4 didn't write 42?

Initially, x == y == z == 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x;	r2 = y;	z = 1;	r0 = z;
if (r1 != 0)	if (r2 != 0)		if (r0 == 1)
y = r1;	x = r2;		x = 42;

Is r0 == 0, r1 == r2 == 42 legal behavior?

Figure 14: Can Threads 1 and 2 see 42, if Thread 4 didn't write to x?

Initially, a = b = c = d = 0

Thread 1/2/3	Thread 4
r1 = a;	
if (r1 == 0)	
b = 1;	r4 = d;
r2 = b;	if (r4 == 1) {
if (r2 == 1)	c = 1;
c = 1;	a = 1;
r3 = c;	}
if (r3 == 1)	
d = 1;	

Behavior in question: r1 == r3 == r4 == 1; r2 == 0

Figure 18: Result of thread inlining of Figure 17; behavior allowed by semantics

that the actions in Threads 3 and 4 affected the actions in Threads 1 and 2, even though they touched none of the same variables. Reasoning about this would be difficult, at best.

The Isolation principle closely interacts with our out of thin air properties. If a thread A does not access the variables accessed by a thread B, then the only way A could have really affected B is if A might have accessed those variables along another program path not taken. The compiler might speculate that the other program path would be taken, and that speculation might affect B. The speculation could only really affect B if B could happen at the same time as A. This would imply a data race between A and B, and we would be speculating about that race; this is something ThinAir2 is designed to avoid.

Isolation is not necessarily a property that should be required in all memory models. It seems to capture a property that is useful and important in a memory model, but all of the implications of it are not understood well enough for us to decide if it must be true of any acceptable memory model.

7.3 Thread Inlining

One behavior that is disallowed by a straightforward interpretation of the out of thin air property that we have developed is shown in Figure 17. An implementation that always scheduled Thread 1 before Thread 2 and Thread 2 before Thread 3 could reasonably decide that the write to d by Thread 3 could be performed before anything in Thread 1 (as long as the guard r3 == 1 evaluates to true). This could lead to a result where the write to d occurs, then Thread 4 writes 1 to c and a. The write to b does not occur, so the read of b by Thread 2 sees 0, and does not write to c. The read of c in Thread 3 then sees the write by Thread 4.

However, this requires reasoning that Thread 3 will see a value for c that is given by a data race. A straightforward interpretation of ThinAir2 therefore disallows this.

In Figure 18, we have another example, similar to the one in Figure 17, where Threads 1, 2 and 3 are combined. We can use the same reasoning that we were going to use for Figure 17 to decide that the write to d can occur early. Here, however, it does not clash with ThinAir2: we are only reasoning about the actions in the combined Thread 1/2/3. The behavior is therefore allowed in this execution.

As a result of this distinction, the compiler writer must be careful when considering inlining threads. When a compiler does decide to inline threads, as in this example, it may not be possible to utilize the full flexibility of the Java memory model when deciding how the resulting code can execute.

8. RELATED WORK

The happens-before relationship has a long history in concurrency literature. It is first described in [9].

The notion that correctly synchronized programs should behave in a sequentially consistent way was first articulated in [3].

An earlier, substantially simpler version of this work appeared in [12]. It did not address the full range of causality issues addressed here.

Most multithreaded languages do not provide strong semantics for multithreaded programs in the presence of data races. The Ada programming language [1], for example, refers to such programs as "erroneous", and discusses them no further. The C# language's [4] underlying framework, the Common Language Infrastructure [5], is also multithreaded; it explicitly allows optimizations to take place when there are data races, but it does not offer specific semantics.

9. CONCLUSION

The adoption of a new Java memory model was a long process. In order to carefully define the requirements, the needs of programmers, compiler writers and processor architects had to be carefully balanced. The end result is a strong statement, not only of what the requirements are for Java (as listed in Figure 19), but one that identifies and classifies these issues for future memory models.

10. ACKNOWLEDGMENTS

Initially, a = b = c = d = 0

Thread 1	Thread 2	Thread 3	Thread 4
`r1 = a;`	`r2 = b;`	`r3 = c;`	`r4 = d;`
`if (r1 == 0)`	`if (r2 == 1)`	`if (r3 == 1)`	`if (r4 == 1) {`
`  b = 1;`	`  c = 1;`	`  d = 1;`	`  c = 1;`
			`  a = 1;`
			`}`

Behavior in question: r1 == r3 == r4 == 1; r2 == 0

Figure 17: Behavior disallowed by semantics

The authors wish to thank the members of the Java memory model mailing list for their time and contribution to this effort. Gratitude is particularly extended to Doug Lea and Sarita Adve for their contributions. Gramercy, also, to David Hovemeyer, for his feedback on this paper.

11. REFERENCES

[1] Ada Joint Program Office. *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, Massachusetts, 1995.

[2] Sarita Adve. Designing memory consistency models for shared-memory multiprocessors. Technical Report 1198, University of Wisconsin, Madison, December 1993. Ph.D. Thesis.

[3] Sarita Adve and Mark Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[4] ECMA. C# Language Specification, December 2002. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[5] ECMA. Common Language Infrastructure (CLI), December 2002. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

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

[7] Java Specification Request (JSR) 133. Java Memory Model and Thread Specification Revision, 2004. http: //jcp.org/jsr/detail/133.jsp.

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

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–564, 1978.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.

[11] Doug Lea. JSR-133 Cookbook, 2004. Available from http: //gee.cs.oswego.edu/dl/jmm/cookbook.html.

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

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

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

[15] Eric Ruf. Effective Synchronization Removal for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC Canada, June 2000.

Name	Description	Exemplar
Guarantees for Optimizers		
Reorder1	Independent actions can be reordered.	Figure 1
Reorder2	If a compiler can detect that an action will always happen (with the same value written to the same variable), it can be reordered regardless of apparent dependencies.	Figures 7, 8, 9, 10, 15
RS	Synchronization actions that only introduce redundant happens-before edges can be treated as if they don't introduce any happens-before edges.	Section 5.5
Guarantees for Programmers		
DRF	Correctly synchronized programs have sequentially consistent semantics.	Figures 2, 6
HB	Volatile writes are ordered before subsequent volatile reads of the same variable. Unlocks are ordered before subsequent locks of the same monitor.	Figure 3
VolatileAtomicity	All accesses to volatile variables are performed in a total order.	Figure 4
StrongVolatile	There is a happens-before relationship from each write to each subsequent read of that volatile.	Figure 5
ThinAir1	A write can only occur earlier in an execution than it would have otherwise occurred if that write would have occurred without assuming that any additional, later reads see non-sequentially consistent values.	Figures 11, 12, 16
ThinAir2	Actions may only be performed earlier than their original place in the program if it can be determined that they could occur in the execution without assuming that any additional reads see values via a data race.	Figures 13, 14, 17, 18
Isolation	Consider a partition P of the threads and variables in the program so that if a thread accessed a variable in that execution, then the thread and variable are in the same partition. Given P, you can explain the behavior in the execution of the threads in each partition without having to examine the behavior or code for the other threads.	Figures 13, 14

Figure 19: Properties of the Java Memory Model