The Java Memory Model[*]

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

Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, IL
sadve@cs.uiuc.edu

ABSTRACT

This paper describes the new Java memory model, which has been revised as part of Java 5.0. The model specifies the legal behaviors for a multithreaded program; it defines the semantics of multithreaded Java programs and partially determines legal implementations of Java virtual machines and compilers.

The new Java model provides a simple interface for correctly synchronized programs – it guarantees sequential consistency to data-race-free programs. Its novel contribution is requiring that the behavior of incorrectly synchronized programs be bounded by a well defined notion of causality. The causality requirement is strong enough to respect the safety and security properties of Java and weak enough to allow standard compiler and hardware optimizations. To our knowledge, other models are either too weak because they do not provide for sufficient safety/security, or are too strong because they rely on a strong notion of data and control dependences that precludes some standard compiler transformations.

Although the majority of what is currently done in compilers is legal, the new model introduces significant differences, and clearly defines the boundaries of legal transformations. For example, the commonly accepted definition for control dependence is incorrect for Java, and transformations based on it may be invalid.

In addition to providing the official memory model for Java, we believe the model described here could prove to be a useful basis for other programming languages that currently lack well-defined models, such as C++ and C#.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.0 [**Programming Languages**]: Standards; F.3.2 [**Logics and Meanings of Programs**]: Operational Semantics

General Terms: Design, Languages

Keywords: Concurrency, Java, Memory Model, Multithreading

1. INTRODUCTION

The memory model for a multithreaded system specifies how memory actions (e.g., reads and writes) in a program will appear to execute to the programmer, and specifically, which value each read of a memory location may return. Every hardware and software interface of a system that admits multithreaded access to shared memory requires a memory model. The model determines the transformations that the system (compiler, virtual machine, or hardware) can apply to a program written at that interface. For example, given a program in machine language, the memory model for the machine language / hardware interface will determine the optimizations the hardware can perform.

For a high-level programming language such as Java, the memory model determines the transformations the compiler may apply to a program when producing bytecode, the transformations that a virtual machine may apply to bytecode when producing native code, and the optimizations that hardware may perform on the native code.

The model also impacts the programmer; the transformations it allows (or disallows) determine the possible outcomes of a program, which in turn determines which design patterns for communicating between threads are legal. Without a well-specified memory model for a programming language, it is impossible to know what the legal results are for a program in that language. For example, a memory model is required to determine what, if any, variants of double checked locking [37] are valid within a particular language.

1.1 Background

Over the last two decades, there has been a tremendous amount of work in the area of memory models at the hardware interface [2, 3, 4, 12, 14, 17, 31, 40, 43], but little at the programming language level. Programming languages present two challenges that do not occur in the hardware world. First, many programming languages, such as Java, have strong safety and security properties that must be respected. Second, the ability of a compiler to perform global and subtle analysis in order to transform a program is essentially unlimited, while hardware systems tend to use a more circumscribed scope in performing optimizations.

[*]Jeremy Manson and William Pugh were supported by National Science Foundation grants CCR-9619808, ACI-9720199 and CCR-0098162, and a gift from Sun Microsystems. Sarita Adve was supported in part by an Alfred P. Sloan research fellowship.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

Previous work [33, 34] has shown that the original semantics for multithreaded Java [18, §17] had serious problems. To address these issues, the Java memory model has recently undergone a major revision. The new model now provides greater flexibility for implementors, a clear notion of what it means to write a correct program, and clear semantics for both correct and incorrect programs. Despite this fundamental change, it is largely, although not entirely, consistent with previous good programming practice and existing JVM implementations and hardware.

The memory model must strike a balance between ease-of-use for programmers and implementation flexibility for system designers. The model that is easiest to understand, sequential consistency [26], specifies that memory actions must appear to execute one at a time in a single total order; actions of a given thread must appear in this total order in the same order in which they appear in the program (called program order).

Whether sequential consistency by itself is an appropriate model for high-level language programmers is a subject of debate: it makes it easier for the programmer to write subtle and complicated concurrent algorithms that do not use explicit synchronization. Unfortunately, such algorithms are often designed or implemented incorrectly, and some would argue that almost all programmers should be discouraged from writing subtle and complicated synchronization-free concurrent algorithms.

Sequential consistency substantially restricts the use of many compiler and hardware transformations. In general, sequential consistency restricts the reordering of any pair of memory statements within a thread, even if there are no data or control dependences between the statements (see Figure 1). This can be a serious limitation since many important optimizations involve reordering program statements. 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. In a sequentially consistent system, no compiler or processor reorderings should become visible to the programmer.

Recently, hardware designers have developed techniques that alleviate some of the limitations of sequential consistency by reordering accesses speculatively [16, 35]. The reordered accesses are committed only when they are known to not be visible to the programmer. Compiler techniques to determine when reorderings are safe have also been proposed [38, 42], but are not yet comprehensively evaluated or implemented in commercial compilers.

A common method to overcome the limitations of sequential consistency is the use of relaxed memory models, which allow more optimizations [12, 17, 19, 31, 40, 43]. Many of these models were originally motivated by hardware optimizations and are described in terms of low-level system attributes such as buffers and caches. Generally, these models have been hard to reason with and, as we show later, do not allow enough implementation flexibility either.

To achieve both programming simplicity and implementation flexibility, alternative models, referred to as *data-race-free* models [2, 4, 6] or *properly-labeled* models [14, 15], have been proposed. This approach exploits the observation that good programming practice dictates that programs be correctly synchronized (data-race-free); a data race is often a symptom of a bug. The data-race-free models formalize correct programs as those that are data-race-free, and guarantee the simple semantics of sequential consistency for such programs. For full implementation flexibility, these models do not provide any guarantees for programs that contain data races. This approach allows standard hardware and compiler optimizations, while providing a simple model for programs written according to widely accepted practice.

Initially, x == y == 0

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

r2 == 2, r1 == 1 violates sequential consistency.

With sequential consistency, the result `r2 == 2, r1 == 1` is impossible. Such a result would imply that statement 4 came before 1 (because `r2 == 2`), 1 came before 2 (due to program order), 2 came before 3 (because `r1 == 1`), 3 came before 4 (due to program order), and so 4 came before 4. Such a cycle is prohibited by sequential consistency. However, if the hardware or the compiler reorder the instructions in each thread, then statement 3 may not come before 4, and/or 1 may not come before 2, making the above result possible. Thus, sequential consistency restricts reordering instructions in a thread, even if they are not data or control dependent.

Figure 1: A Violation of Sequential Consistency.

1.2 The Java Memory Model

The new Java model adopts the data-race-free approach for correct programs – correct programs are those that are data-race-free; such programs are guaranteed sequential consistency. Unfortunately, if a model leaves the semantics for incorrect programs unspecified, it allows violations of safety and security properties that are necessary for Java programs.

A key contribution of the revision effort has been to show that the safety and security properties of Java require prohibiting a class of executions that contain data races and have not been previously characterized. The challenge is to characterize and prohibit this class of executions without prohibiting standard compiler and hardware transformations or unduly complicating the model. In earlier relaxed models, such executions were either allowed, or were only prohibited by enforcing traditional control and data dependences. We show that the latter approach restricts standard compiler optimizations and is not viable for Java.

The revised Java model is based on a new technique that prohibits the necessary executions while allowing standard optimizations. Our technique builds legal executions iteratively. In each iteration, it *commits* a set of memory actions; actions can be committed if they occur in some *well-behaved execution* that also contains the actions committed in previous iterations. A careful definition of "well-behaved executions" ensures that the appropriate executions are prohibited and standard compiler transformations are allowed. To our knowledge, this is the only model with this property.

This paper focuses on the semantics for ordinary reads and writes, locks, and volatile variables in the Java memory model. It also covers the memory semantics for interaction with the outside world, and infinite executions. For space reasons, we omit discussion of two important issues in the

Java memory model: the treatment of final fields, and finalization / garbage collection. We discuss the rest of the model in detail; the full specification is available elsewhere [20].

The revision process for the Java model involved continual feedback and participation from the broader Java community; we refer the reader to the Java memory model mailing list archives for more information on the evolution of the model [21].

While this model was developed for Java, many of the issues that are addressed apply to every multithreaded language. It is our hope that the work presented here can be leveraged to provide the right balance of safety and efficiency for future programming languages.

2. REQUIREMENTS FOR THE JAVA MEMORY MODEL

Our major design goal was to provide a balance between (1) sufficient ease of use and (2) transformations and optimizations used in current (and ideally future) compilers and hardware.

Given that current hardware and compilers employ transformations that violate sequential consistency, it is currently not possible to provide a sequentially consistent programming model for Java. The memory model for Java is therefore a relaxed model.

A detailed description of the requirements for the Java model and their evolution can be found elsewhere [30]. Here we provide a brief overview.

2.1 Correctly Synchronized Programs

It is difficult for programmers to reason about specific hardware and compiler transformations. For ease of use, we therefore specify the model so that programmers do not have to reason about hardware or compiler transformations or even our formal semantics for correctly synchronized code. We follow the data-race-free approach to define a correctly synchronized program (data-race-free) and correct semantics for such programs (sequential consistency). The following definitions formalize these notions.

- **Conflicting Accesses:** A read of or write to a variable is an *access* to that variable. Two accesses to the same shared field or array element are said to be *conflicting* if at least one of the accesses is a write.

- **Synchronization Actions:** *Synchronization actions* include locks, unlocks, reads of volatile variables, and writes to volatile variables.

- **Synchronization Order:** Each execution of a program is associated with a *synchronization order*, which is a total order over all synchronization actions. To define data-race-free programs correctly, we are allowed to consider only those synchronization orders that are also consistent with program order and where a read to a volatile variable v returns the value of the write to v that is ordered last before the read by the synchronization order. These latter requirements on the synchronization order are explicitly imposed as a synchronization order consistency requirement in the rest of the paper.

- **Synchronizes-With Order:** For two actions x and y, we use $x \overset{sw}{\to} y$ to mean that x *synchronizes-with* y. An unlock action on monitor m synchronizes-with all subsequent lock actions on m that were performed by any thread, where subsequent is defined according to the synchronization order. Similarly, a write to a volatile variable v synchronizes-with all subsequent reads of v by any thread. There are additional synchronized-with edges in Java [20] that are not discussed here for brevity.

- **Happens-Before Order:** For two actions x and y, we use $x \overset{hb}{\to} y$ to mean that x *happens-before* y [25]. Happens-before is the transitive closure of program order and the synchronizes-with order.

- **Data Race:** Two accesses x and y form a *data race* in an execution of a program if they are from different threads, they conflict, and they are not ordered by happens-before.

- **Correctly Synchronized or Data-Race-Free Program [4, 6]:** A program is said to be *correctly synchronized* or *data-race-free* if and only if all sequentially consistent executions of the program are free of data races.

The first requirement for the Java model is to ensure sequential consistency for correctly synchronized or data-race-free programs as defined above. Programmers then need only worry about code transformations having an impact on their programs' results if those programs contain data races.

As an example, the code in Figure 1 is incorrectly synchronized. Any sequentially consistent execution of the code contains conflicting accesses to x and y and these conflicting accesses are not ordered by happens-before (since there is no synchronization in the program). A way to make this program correctly synchronized is to declare x and y as volatile variables; then the program is assured of sequentially consistent results with the Java model.

2.2 Out-of-Thin-Air Guarantees for Incorrect Programs

The bulk of the effort for the revision, and our focus here, was on understanding the requirements for incorrectly synchronized code. The previous strategy of leaving the semantics for incorrect programs unspecified is inconsistent with Java's security and safety guarantees. Such a strategy has been used in some prior languages. For example, Ada 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. This is similar to the 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 above approach does not lend itself to the writing of secure and safe code. In an ideal world, every programmer would write correct code all of the time. In our world, programs frequently contain errors; not only does this cause code to misbehave, but it can also allows attackers to violate safety assumptions in a program (as is true with buffer overflows). Our earlier work has described the dangers of such scenarios in more detail [28].

Program semantics must be completely defined: if programmers don't know what their code is doing, they won't

Initially, x == y == 0

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

Incorrectly synchronized, but we want to disallow `r1 == r2 == 42`.

Figure 2: An Out Of Thin Air Result

be able to know what their code is doing wrong. The second broad requirement of the Java model is to provide a clear and definitive semantics for how code should behave when it is not correctly written, but without substantially affecting current compilers and hardware.

Figure 2 contains a common example of an program for which careful semantics are required. It is incorrectly synchronized; all sequentially consistent executions of this program exhibit data races between Threads 1 and 2. However, we need to provide a strong guarantee for this code: we need to ensure that a value such as 42 will not appear *out of thin air* in `r1` and `r2`.

There are no current optimizations that would permit such an out-of-thin-air result. However, in a future aggressive system, Thread 1 could speculatively write the value 42 to `y`, which would allow Thread 2 to read 42 for `y` and write it out to `x`, which would allow Thread 1 to read 42 for `x`, and justify its original speculative write of 42 for `y`. A self-justifying write speculation like that one can create serious security violations and needs to be disallowed. For example, this would be a serious problem if the value that was produced out of thin air was a reference to an object that the thread was not supposed to have (because of, for example, security guarantees).

Characterizing precisely what constitutes an out-of-thin-air violation is complicated. Not all speculations that appear to be self-justifying are security violations, and some can even arise out of current compiler transformations. How we characterize the appropriate set of violations is at the core of the new Java model, and discussed further in Section 4.2.

3. HAPPENS-BEFORE MEMORY MODEL

Before presenting the Java model in full, we present a simpler memory model, called the *happens-before memory model*.

Each legal execution under this model has several properties/requirements (Section 5 provides a more formal version of the requirements):

- There is a synchronization order over synchronization actions, synchronization actions induce synchronizes-with edges between matched actions, and the transitive closure of the synchronizes-with edges and program order gives an order known as the happens-before order (as described in Section 2.1).
- For each thread t, the actions t performs in the execution are the same as those that t would generate in program order in isolation, given the values seen by the reads of t in the execution.
- A rule known as happens-before consistency (see below) determines the values a non-volatile read can see.
- A rule known as synchronization order consistency (see below) determines the value a volatile read can see.

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: Use of Happens-Before Memory Model with Volatiles

Happens-before consistency says that a read r of a variable v is *allowed* to observe a write w to v if, in the happens-before partial order of the execution:

- r does not happen-before w (i.e., it is not the case that $r \xrightarrow{hb} w$) – a read cannot see a write that happens-after it, and
- there is no intervening write w' to v, ordered by happens-before (i.e., no write w' to v such that $w \xrightarrow{hb} w' \xrightarrow{hb} r$) – the write w is not overwritten along a happens-before path.

Synchronization order consistency says that (i) synchronization order is consistent with program order and (ii) each read r of a volatile variable v sees the last write to v to come before it in the synchronization order.

The happens-before memory model is a simple approximation to the semantics we want for the Java memory model. For example, the happens-before memory model allows the behavior shown in Figure 1. There are no synchronizes-with or happens-before edges between threads, and each read is allowed to return the value of the write by the other thread.

Figure 3 provides an example of how to use the happens-before memory model to restrict the values a read can return. Notice that `ready` is a volatile boolean variable. There is therefore a happens-before edge from the write `ready = true` to any read that sees that write.

Assume Thread 2's read of `ready` sees the write to `ready`. Thread 1's write of `x` must happen-before Thread 2's read of `x` (because of program order), and Thread 2's read must return the new value of x. In more detail: the initial value of x is assumed to happen-before all actions. There is a path of happens-before edges that leads from the initial write, through the write to `x`, to the read of `x`. On this path, the initial write is seen to be overwritten. Thus, if the read of `ready` in Thread 2 sees `true`, then the read of `x` must see the value `1`.

If `ready` were not volatile, one could imagine a compiler transformation that reordered the two writes in Thread 1, resulting in a read of `true` for `ready`, but a read of `0` for `x`.

4. CAUSALITY

Although it is simple, the happens-before memory model (as discussed in Section 3) allows unacceptable behaviors. Notice that the behavior we want to disallow in Figure 2 is consistent with the happens-before memory model. If both write actions write out the value 42, and both reads see those writes, then both reads see values that they are allowed to see.

The happens-before memory model also does not fulfill our requirement that correctly synchronized programs have sequentially consistent semantics (first observed in [6, 7]).

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 5: Effects of Redundant Read Elimination

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 we must disallow r1 == r2 == 42.

Figure 4: Correctly Synchronized Program

For example, the code shown in Figure 4 [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.

For the above program, the only sequentially consistent result is r1 == r2 == 0; this is therefore the only result that we want to allow under the Java model. However, as the result in Figure 2, the result of r1 == r2 == 42 for Figure 4 is consistent with the happens-before memory model. If both writes occur, and both reads see them, then the if guards are both true, and both reads see writes that they are allowed to see. As mentioned, this violates the guarantee that we wish to make of sequential consistency for correctly synchronized programs.

Regardless of these weaknesses, the happens-before memory model provides a good outer bound for our model; all of our executions must be consistent with it.

4.1 Data and Control Dependencies

The missing link between our desired semantics and the happens-before memory model is one of *causality.* In Figures 2 and 4, actions which caused the illegal writes to occur were, themselves, caused by the writes. This kind of circular causality seems undesirable in general, and these examples are prohibited in the Java memory model.

In both of those examples, a write that is data or control dependent on a read appears to occur before that read, and then causes the read to return a value that justifies that write. In Figure 2, the value written by the write is used to justify the value that it does write. In Figure 4, the occurrence of the write is used to justify the fact that the write will execute.

Unfortunately, the notion of "cause" is a tricky one, and cannot employ the notion of data and control dependencies.

Initially, x = y = 0

Thread 1	Thread 2
1: r1 = x;	4: r3 = y;
2: r2 = r1 | 1;	5: x = r3;
3: y = r2;	

Is r1 == r2 == r3 == 1 possible?

Figure 6: Using Global Analysis

For example, the Java memory model allows the behavior shown in Figure 5, even though that example also seems to involve a case of circular causality. This behavior must be allowed, because a compiler can

- 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.

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. For this transformed program (illustrated in Figure 5(b)), the outcome r1 == r2 == r3 == 2 is sequentially consistent and could be produced on many commercial systems. Thus, dependence-breaking optimizations can also result in executions containing apparent "causal cycles".

Figure 6 shows another example in which the compiler should be allowed to perform a global analysis that eliminates a data dependence. For the behavior in Figure 6 to occur, it would seem that the write to y must occur before the read of x, which would seem to violate the data dependence of the write to y on the read of x. However, a compiler could use an interthread analysis to determine that the only possible values for x and y are 0 or 1. Knowing this, the compiler can determine that r2 is always 1. This breaks the data dependence, allowing a write of the constant value 1 to y before the read of x by thread 1. The Java model allows both this analysis and the behavior in Figure 6.

4.2 Causality and the Java Memory Model

The key question now becomes how to formalize the notion of causality so we restrict causal violations of the type in Figures 2 and 4, but allow outcomes of the type in Figure 5.

We address these issues by considering when actions can occur in an execution. Both Figures 4 and 5 involve moving actions (specifically, writes) earlier than they otherwise would have occurred under pure sequential consistency.

One difference between the acceptable and unacceptable results is that in Figure 5, the write that we perform early `4: b = 2` would also have occurred if we had carried on the execution in a sequentially consistent way, accessing the same location and writing the same value. In Figure 4, the early write could never write the same value (`42`) in any sequentially consistent execution. These observations provide an intuition as to when an action should be allowed to occur in a legal Java execution.

To justify performing the writes early in Figures 5 and 6, we find a *well-behaved* execution in which those writes took place, and use that execution to perform the justification. Our model therefore builds an execution iteratively; it allows an action (or a group of actions) to be *committed* if it occurs in some well-behaved execution that also contains the actions committed so far.

Identifying what constitutes a "well-behaved" execution is the key to our notion of causality. We distinguished the early writes in Figures 2 and 4 from the early writes in Figures 5 and 6 by considering whether those writes could occur in a sequentially consistent execution. While it may appear intuitive to use the notion of a sequentially consistent execution to justify the occurrence of early writes, this notion turns out to be too relaxed in some subtle cases.

After trying several possibilities for formalizing the "well-behaved" executions that could justify early writes, we converged on the following. We observed that early execution of an action does not result in an undesirable causal cycle *if its occurrence is not dependent on a read returning a value from a data race.* This insight led to our notion of a "well-behaved execution": a read that is not yet committed must return the value of a write that is ordered before it by happens-before.

We can use a "well-behaved" execution to "justify" further executions where writes occur early. Given a well-behaved execution, we may commit any of the uncommitted writes that occur in it (with the same address and value). We also may commit any uncommitted reads that occur in such an execution, but additionally require that the read return the value of a previously committed write in both the justifying execution and the execution being justified (we allow these writes to be different).

To keep consistency among the successive justifying executions, we also require that across all justifying executions, the happens-before and synchronization order relations among committed accesses remains the same, and the values returned by committed reads remain the same.

Our choice of justifying executions ensures that the occurrence of a committed action and its value does not depend on an uncommitted data race. We build up a notion of causality, where actions that are committed earlier may cause actions that are committed later to occur. This approach ensures that later commits of accesses involved in data races will not result in an undesirable cycle.

Consider again the executions with undesirable causal behaviors in Figure 2 and 4. To get the undesirable behavior, the actions in those executions must all read and write the value 42. This implies that some action must read or write the value 42 in an execution where reads only return values of writes that happen-before them. However, there is no such execution, so we are unable to commit any of the actions in those executions.

On the other hand, given the code in Figure 5, there is an execution where all reads return values from writes that happen-before them and where the write `4: b = 2` occurs. This is any execution where both `r1` and `r2` read the value 0 for `a`. Thus, we can commit `4: b = 2`. This allows us to commit a read of `b` by Thread 2 that returns the value 2. We use the same justifying execution for this – the read returns the value 1 in the justifying execution (since this does not involve a race), but can return the value 2 in the execution being justified (since the write of b to 2 is already committed). Using a similar procedure, we can commit the rest of the accesses. Thus, the model allows the execution in Figure 5, but not those in Figure 4 and 2.

The next section presents the model, as we have discussed it, using a formal notation. The causality constraints are presented in Section 5.4.

5. FORMAL SPECIFICATION

This section provides the formal specification of the Java memory model.

5.1 Actions and Executions

An action a is described by a tuple $\langle t, k, v, u\rangle$, comprising:

t - the thread performing the action

k - the kind of action: volatile read, volatile write, (non-volatile) read, (non-volatile) write, lock, unlock, special synchronization action, thread divergence actions and external actions. Volatile reads, volatile writes, locks and unlocks are synchronization actions, as are special synchronization actions such as the start of a thread, the synthetic first or last action of a thread, and actions that detect that a thread has terminated, as described in Section 2.1.

v - the (runtime) variable or monitor involved in the action

u - an arbitrary unique identifier for the action

An execution E is described by a tuple

$$\langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} \rangle$$

comprising:

P - a program

A - a set of actions

$\overset{po}{\rightarrow}$ - program order, which for each thread t, is a total order over all actions performed by t in A

$\overset{so}{\rightarrow}$ - synchronization order, which is a total order over all synchronization actions in A

W - a write-seen function, which for each read r in A, gives $W(r)$, the write action seen by r in E.

V - a value-written function, which for each write w in A, gives $V(w)$, the value written by w in E.

$\overset{sw}{\rightarrow}$ - synchronizes-with, a partial order over synchronization actions.

$\xrightarrow{hb}$ - happens-before, a partial order over actions

Note that the synchronizes-with and happens-before are uniquely determined by the other components of an execution and the rules for well-formed executions.

Two of the action types require special descriptions, and are detailed further in Section 7. These actions are introduced so that we can explain why such a thread may cause all other threads to stall and fail to make progress.

external actions - An external action is an action that may be observable outside of an execution, and may have a result based on an environment external to the execution. An external action tuple contains an additional component, which contains the results of the external action as perceived by the thread performing the action. This may be information as to the success or failure of the action, and any values read by the action.

Parameters to the external action (e.g., which bytes are written to which socket) are not part of the external action tuple. These parameters are set up by other actions within the thread and can be determined by examining the intra-thread semantics. They are not explicitly discussed in the memory model.

In non-terminating executions, not all external actions are observable. Non-terminating executions and observable actions are discussed in Section 7.

thread divergence action - A thread divergence action is only performed by a thread that is in an infinite loop in which no memory, synchronization or external actions are performed. If a thread performs a thread divergence action, that action is followed in program order by an infinite sequence of additional thread divergence actions.

5.2 Definitions

1. **Definition of synchronizes-with**. The synchronizes-with order is described in Section 2.1. The source of a synchronizes-with edge is called a *release*, and the destination is called an *acquire*.

2. **Definition of happens-before**. The happens-before order is given by the transitive closure of the program order and the synchronizes-with order. This is discussed in detail in Section 2.1.

3. **Definition of sufficient synchronization edges**. A set of synchronization edges is *sufficient* if it is the minimal set such that if the transitive closure of those edges with program order edges is taken, all of the happens-before edges in the execution can be determined. This set is unique.

4. **Restrictions of partial orders and functions**. We use $f|_d$ to denote the function given by restricting the domain of f to d: for all $x \in d$, $f(x) = f|_d(x)$ and for all $x \notin d, f|_d(x)$ is undefined. Similarly, we use $\xrightarrow{e}|_d$ to represent the restriction of the partial order $\xrightarrow{e}$ to the elements in d: for all $x, y \in d$, $x \xrightarrow{e} y$ if and only if $x \xrightarrow{e}|_d y$. If either $x \notin d$ or $y \notin d$, then it is not the case that $x \xrightarrow{e}|_d y$.

5.3 Well-Formed Executions

We only consider well-formed executions. An execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$ is well formed if the following conditions are true:

1. **Each read of a variable x sees a write to x. All reads and writes of volatile variables are volatile actions.** For all reads $r \in A$, we have $W(r) \in A$ and $W(r).v = r.v$. The variable $r.v$ is volatile if and only if r is a volatile read, and the variable $w.v$ is volatile if and only if w is a volatile write.

2. **Synchronization order is consistent with program order and mutual exclusion.** Having synchronization order consistent with program order implies that the happens-before order, given by the transitive closure of synchronizes-with edges and program order, is a valid partial order: reflexive, transitive and antisymmetric. Having synchronization order consistent with mutual exclusion means that on each monitor, the lock and unlock actions are correctly nested.

3. **The execution obeys intra-thread consistency.** For each thread t, the actions performed by t in A are the same as would be generated by that thread in program-order in isolation, with each write w writing the value $V(w)$, given that each read r sees / returns the value $V(W(r))$. Values seen by each read are determined by the memory model. The program order given must reflect the program order in which the actions would be performed according to the intrathread semantics of P, as specified by the parts of the JLS that do not deal with the memory model.

4. **The execution obeys synchronization-order consistency.** Consider all volatile reads $r \in A$. It is not the case that $r \xrightarrow{so} W(r)$. Additionally, there must be no write w such that $w.v = r.v$ and $W(r) \xrightarrow{so} w \xrightarrow{so} r$.

5. **The execution obeys happens-before consistency.** Consider all reads $r \in A$. It is not the case that $r \xrightarrow{hb} W(r)$. Additionally, there must be no write w such that $w.v = r.v$ and $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$.

5.4 Causality Requirements for Executions

A well-formed execution

$$E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$$

is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E_i containing C_i that meets certain conditions.

Formally, an execution E satisfies the causality requirements of the Java memory model if and only if there exist

- Sets of actions $C_0, C_1, \ldots$ such that
  - $C_0 = \emptyset$
  - $C_i \subset C_{i+1}$

- $A = \cup(C_0, C_1, C_2, \ldots)$

such that E and $(C_0, C_1, C_2, \ldots)$ obey the restrictions listed below.

The sequence $C_0, C_1, \ldots$ may be finite, ending in a set $C_n = A$. However, if A is infinite, then the sequence $C_0, C_1, \ldots$ may be infinite, and it must be the case that the union of all elements of this infinite sequence is equal to A.

- Well-formed executions $E_1, \ldots$, where $E_i = \langle P, A_i, \xrightarrow{po_i}, \xrightarrow{so_i}, W_i, V_i, \xrightarrow{sw_i}, \xrightarrow{hb_i} \rangle$.

Given these sets of actions $C_0, \ldots$ and executions $E_1, \ldots$, every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally,

1. $C_i \subseteq A_i$
2. $\xrightarrow{hb_i} |_{C_i} = \xrightarrow{hb} |_{C_i}$
3. $\xrightarrow{so_i} |_{C_i} = \xrightarrow{so} |_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. The reads in C_{i-1} need to see the same writes in E_i as in E (but not the reads in $C_i - C_{i-1}$) Formally,

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in $C_i - C_{i-1}$ must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally,

6. For any read $r \in A_i - C_{i-1}$, we have $W_i(r) \xrightarrow{hb_i} r$
7. For any read $r \in C_i - C_{i-1}$, we have $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before an action in $C_i - C_{i-1}$, then that pair must be present in all E_j, where $j \geq i$. Formally,

8. Let $\xrightarrow{ssw_i}$ be the $\xrightarrow{sw_i}$ edges that are in the transitive reduction of $\xrightarrow{hb_i}$ but not in $\xrightarrow{po_i}$. We call $\xrightarrow{ssw_i}$ the sufficient synchronizes-with edges for E_i. If $x \xrightarrow{ssw_i} y \xrightarrow{hb_i} z$ and $z \in C_i - C_{i-1}$, then $x \xrightarrow{sw_j} y$ for all $j \geq i$.

If an action y is committed, all external actions that happen-before y are also committed.

9. If $y \in C_i$, x is an external action and $x \xrightarrow{hb_i} y$, then $x \in C_i$.

6. EXAMPLE

As a simple example of how the memory model works, consider Figure 7. Note that there are initially writes of the default value 0 to x and y. We wish to get the result r1 == r2 == 1, which can occur if a compiler reorders the statements in Thread 1. This result is consistent with the happens-before memory model, so we only have to ensure that it complies with the causality rules in Section 5.4.

Initially, x = y = 0

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

r1 == r2 == 1 is a legal behavior

Figure 7: A Standard Reordering

Action	Final Value	First Committed In	First Sees Final Value In
x = 0	0	C_1	E_1
y = 0	0	C_1	E_1
y = 1	1	C_1	E_1
r2 = y	1	C_2	E_3
x = r2	1	C_3	E_3
r1 = x	1	C_4	E

Figure 8: Table of commit sets for Figure 7

The set of actions C_0 is the empty set. Therefore, according to Rule 6, in execution E_1, all reads must see values of writes that happen-before them. That is, in E_1, both reads must see the value 0. We first commit the initial writes of 0 to x and y, as well as the write of 1 to y by Thread 1; these writes are contained in the set C_1.

We wish the action r2 = y to see the value 1. C_1 cannot contain this action seeing this value: neither write to y had been committed. C_2 may contain this action; however, the read of y must return 0 in E_2, because of Rule 6. Execution E_2 is therefore identical to E_1.

In E_3, by Rule 7, r2 = y can see any conflicting write that occurs in C_2 (as long as that write is happens-before consistent). This action can now see the write of 1 to y in Thread 1, which was committed in C_1. We commit one additional action in C_3: a write of 1 to x by x = r2.

C_4, as part of E_4, contains the read r1 = x; it still sees 0, because of Rule 6. In our final execution $E = E_5$, however, Rule 7 allows r1 = x to see the write of 1 to x that was committed in C_3.

7. OBSERVABLE BEHAVIOR AND NONTERMINATING EXECUTIONS

For programs that always terminate in some bounded finite period of time, their behavior can be understood (informally) simply in terms of their allowable executions. For programs that can fail to terminate in a bounded amount of time, more subtle issues arise.

The observable behavior of a program is defined by the finite sets of external actions that the program may perform. A program that, for example, simply prints "Hello" forever is described by a set of behaviors that for any non-negative integer i, includes the behavior of printing "Hello" i times.

Termination is not explicitly modeled as a behavior, but a program can easily be extended to generate an additional external action "executionTermination" that occurs when all threads have terminated.

We also define a special *hang* action. If a behavior is described by a set of external actions including a *hang* action, it indicates a behavior where after the (non-hang) external

Initially, `v` is volatile and `v = false`

Thread 1	Thread 2
`while (!v);`	`v = true;`
`System.out.println("Thread 1 done");`	`System.out.println("Thread 2 done");`

If we observe print message by thread 2, Thread 1 must see write to `v`, print its message and terminate. But program can also be observed to hang and not print any messages.

Figure 9: Fairness Example

actions are observed, the program can run for an unbounded amount of time without performing any additional external actions or terminating. Programs can *hang*:

- if all non-terminated threads are blocked, and at least one such blocked thread exists, or

- if the program can perform an unbounded number of actions without performing any external actions.

A thread can be blocked in a variety of circumstances, such as when it is attempting to acquire a lock or perform an external action (such as a read) that depends on an external data. If a thread is in such a state, `Thread.getState` will return `BLOCKED` or `WAITING`. An execution may result in a thread being blocked indefinitely and the execution not terminating. In such cases, the actions generated by the blocked thread must consist of all actions generated by that thread up to and including the action that caused the thread to be blocked indefinitely, and no actions that would be generated by the thread after that action.

To reason about observable behaviors, we need to talk about sets of observable action. If O is a set of observable actions for E, then set O must be a subset of A, and must contain only a finite number of actions, even if A contains an infinite number of actions. Furthermore, if an action $y \in O$, and either $x \overset{hb}{\rightarrow} y$ or $x \overset{so}{\rightarrow} y$, then $x \in O$.

Note that a set of observable actions is not restricted to containing external actions. Rather, only external actions that are in a set of observable actions are deemed to be observable external actions.

A behavior B is an allowable behavior of a program P if and only if B is a finite set of external actions and either

- There exists an execution E of P, and a set O of observable actions for E, and B is the set of external actions in O (if any threads in E end in a blocked state and O contains all actions in E, then B may also contain a *hang* action), or

- There exists a set O of actions such that
  - B consists of a *hang* action plus all the external actions in O and
  - for all $K \geq |O|$, there exists an execution E of P and a set of actions O' such that:
    - Both O and O' are subsets of A that fulfill the requirements for sets of observable actions.
    - $O \subseteq O' \subseteq A$
    - $|O'| \geq K$
    - $O' - O$ contains no external actions

Note that a behavior B does not describe the order in which the external actions in B are observed, but other (implicit and unstated) constraints on how the external actions are generated and performed may impose such constraints.

7.1 Discussion

The Java language specification does not guarantee preemptive multithreading or any kind of fairness: there is no hard guarantee that any thread will surrender the CPU and allow other threads to be scheduled. The lack of such a guarantee is partially due to the fact that any such guarantee would be complicated by issues such as thread priorities and real-time threads (in Real-time Java virtual machine implementations). Most Java virtual machine implementations will provide some sort of fairness guarantee, but the details are implementation specific and are treated as a quality of service issue, rather than a rigid requirement.

However, there are some limitations on compiler transformations that reduce fairness. For example, in Figure 9, if we observe the print message from Thread 2, and no threads other than Threads 1 and 2 are running, then Thread 1 must see the write to `v`, print its message and terminate. This prevents the compiler from hoisting the volatile read of `v` out of the loop in Thread 1.

The fact that Thread 1 must terminate if the print by Thread 2 is observed follows from the rules on observable actions. If the print by Thread 2 is in a set of observable actions O, then the write to `v` and all reads of `v` that see the value 0 must also be in O. Additionally, the program cannot perform an unbounded amount of additional actions that are not in O. Therefore, the only observable behavior of this program in which the program *hangs* (runs forever without performing additional external actions) is one in which it performs no observable external actions other than hanging. This includes the print action.

It follows from this that in Figure 9, the instructions in Thread 2 cannot be arbitrarily reordered. If this reordering were performed, it would be possible for Thread 2 to print, after which there could be a context switch to Thread 1; Thread 1 would never terminate.

Note that the program in Figure 9 has only finite executions. Since there are no blocking operations in Thread 2, it must run to completion in any execution. However, we want to allow this program to be executed such that it is not observed to terminate (which might happen if Thread 1 were run and never performed a context switch). One of the things we needed to accomplish in the formalization of observable behavior was that no distinction should be made between a program that can run for a finite, but unbounded, number of steps and a program that can run for an infinite number of steps. For example, if accesses to volatile fields in Figure 9 were implemented by surrounding them with

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

Thread 1	Thread 2	Thread 3	Thread 4
1: z = 1	2: r1 = z	5: r2 = x	7: r3 = y
	3: if (r1 == 0)	6: y = r2	8: x = r3
	4: x = 1		

Must not allow r1 == r2 == r3 == 1

Figure 10: Prohibited "bait-and-switch" behavior

Initially, x == y == 0

Thread 1	Thread 2	Thread 3
1: r1 = x	4: r2 = x	6: r3 = y
2: if (r1 == 0)	5: y = r2	7: x = r3
3: x = 1		

Must not allow r1 == r2 == r3 == 1

Figure 11: A variant "bait-and-switch" behavior

locks, Thread 2 could block and never acquire the lock. In this situation, the program would execute infinitely.

8. SURPRISING AND CONTROVERSIAL BEHAVIORS

Many of the requirements and goals for the Java memory model were straightforward and non-controversial (e.g., correctly synchronized programs exhibit only sequentially consistent behavior). Other decisions about the requirements generated quite a bit of discussion; the final decision often came down to a matter of taste and preference rather than any concrete requirement.

One of the most subtle of these is that in programs with data races, certain kinds of behaviors were deemed to display an unacceptable "bait-and-switch" circular reasoning. One such example is shown in Figure 10, in which it was decided that the behavior `r1 == r2 == r3 == 1` was unacceptable. In an execution that exhibits this behavior, only Threads 3 and 4 read and write to `x` and `y`. Thus, this seems to be an "out-of-thin-air" example, as in Figure 2. What differentiates this example from Figure 2 is that in Figure 10, Thread 2 *might have, but didn't* write 1 to `x`.

After *much* discussion within the Java community, it was decided that allowing such behaviors would make reasoning about the safety properties of programs too difficult. Reasoning about the behaviors of Threads 3 and 4 in Figure 10 requires reasoning about Threads 1 and 2, even when trying to reason about executions in which Threads 1 and 2 did not interact with Threads 3 and 4.

Given that the behavior in Figure 10 is unacceptable, the behavior in Figure 11 is very similar, and it seems reasonable to prohibit it as well (the behaviors in Figures 10 and 11 are prohibited by the Java memory model).

Figure 12 shows a code fragment very similar to that of Figure 11. However, for the code in Figure 12, we must allow the behavior that was prohibited in Figure 11. We do this because that behavior can result from well understood and reasonable compiler transformations.

Initially, x == y == 0

Thread 1	Thread 2
1: r1 = x	6: r3 = y
2: if (r1 == 0)	7: x = r3
3: x = 1	
4: r2 = x	
5: y = r2	

Compiler transformations can result in r1 == r2 == r3 == 1

Figure 12: Behavior that must be allowed

- The compiler can deduce that the only legal values for `x` and `y` are `0` and `1`.
- The compiler can then replace the read of `x` on line 4 with a read of `1`, because either
  - 1 was read from `x` on line 1 and there was no intervening write, or
  - 0 was read from `x` on line 1, 1 was assigned to `x` on line 3, and there was no intervening write.
- Via forward substitution, the compiler is allowed to transform line 5 to `y = 1`. Because there are no dependencies, this line can be made the first action performed by Thread 1.

After these transformations are performed, a sequentially consistent execution of the program will result in the behavior in question.

The fact that the behavior in Figure 11 is prohibited and the behavior in Figure 12 is allowed is, perhaps, surprising. We could derive Figure 12 from Figure 11 by "inlining" Threads 1 and 2 into a single thread. This shows that, in general, thread inlining is not legal under the Java memory model.

This is an example where adding happens-before relationships can increase the number of allowed behaviors. This property can be seen as an extension of the way in which causality is handled in the Java memory model (as discussed in Section 4.2). The happens-before relationship is used to express causation between two actions; if an additional happens-before relationship is inserted, the causal relationships change.

Other approaches may exist to defining an acceptable memory model for a programming language. However, much energy was expended on the Java memory model, and more than a dozen different approaches were considered. The approach presented in this paper, part of the Java 5.0 standard, was found to be the only acceptable approach.

9. FORMALIZING THE IMPACT OF THE MODEL

Section 2 discussed some of the properties we wanted the model to reflect. This section discusses how those properties are realized.

9.1 Considerations for Implementors

This section discusses some of the key ways in which the new Java memory model impacts the decisions that must be taken by Java platform implementors. Remember from Section 2, that one of our key informal requirements was that the memory model allow as many optimizations as it can.

9.1.1 Control Dependence

The new Java memory model makes changes that are subtle, but deep, to the way in which implementors must reason about Java programs. For example, the standard definition of control dependence assumes that execution always proceeds to exit. This must not be casually assumed in multithreaded programs.

Consider the program in Figure 13. Under the traditional definitions of control dependence, neither of the writes in either thread are control dependent on the loop guards. This might lead a compiler to decide that the writes could be reordered to before the loops. However, this would be illegal in Java. This program is, in fact, correctly synchronized: in all sequentially consistent executions, neither thread writes to shared variables and there are no data races (this figure is very similar to Figure 4). A compiler must not reorder the writes in that example.

The notion of control dependence that correctly encapsulates this is called *weak control dependence* [32] in the context of program verification. This property has also been restated as *loop control dependence* [10] in the context of program analysis and transformation.

9.1.2 Semantics Allow Reordering

We mentioned earlier that a key notion for program optimization was that of *reordering*. We demonstrated in Figure 1 that standard compiler reorderings, unobservable in single threaded programs, can have observable effects in multithreaded programs. However, reorderings are crucial in many common code optimizations; e.g., instruction scheduling, register allocation, common sub-expression elimination and redundant read elimination.

In this section, we demonstrate that many of the reorderings necessary for these optimizations are legal. This is not a complete list of legal reorderings; others can be derived from the model. However, this demonstrates a sample of common reorderings, used for many common optimizations. Specifically, we demonstrate the legality of reordering two independent actions when doing so does not change the happens-before relationship for any other actions.

THEOREM 1. *Consider a program P and the program P' that is obtained from P by reordering two adjacent statements s_x and s_y. Let s_x be the statement that comes before s_y in P, and after s_y in P'. The statements s_x and s_y may be any two statements such that*

- *reordering s_x and s_y doesn't eliminate any transitive happens-before edges in any valid execution (it will reverse the direct happens-before edge between the actions generated by s_x and s_y)*
- *Reordering s_x and s_y does not hoist an action above an infinite loop*
- *s_x and s_y are not conflicting accesses to the same variable,*
- *s_x and s_y are not both synchronization actions or external actions, and*
- *the intra-thread semantics of s_x and s_y allow reordering (e.g., s_x doesn't store into a register that is read by s_y).*

Transforming P into P' is a legal program transformation.

Proof:

Assume that we have a valid execution E' of program P'. It has a legal set of behaviors B'. To show that the transformation of P into P' is legal, we need to show that there is a valid execution E of P that has the same observable behaviors as E'.

Let x be the action generated by s_x and y be the action generated by s_y. If x and y are executed multiple times, we repeat this analysis for each repetition.

The execution E' has a set of observable actions O', and the execution E has a set of observable actions O. If O includes x, O must also include y because $x \xrightarrow{hb} y$ in E. If O' does not include y (and therefore y is not an external action, as it must not take place after an infinite series of actions), we can use O as the set of observable actions for E' instead: they induce the same behavior.

Since E' is legal, we have $E'_0, E'_1 \ldots$, a sequence of executions that eventually justifies E. E'_0 doesn't have any committed actions and $\forall i, 0 \leq i$, E'_i is used to justify the additional actions that are committed to give E'_{i+1}.

We will show that we can use $E_i \equiv E'_i$ to show that $E \equiv E'$ is a legal execution of P.

If x and y are both uncommitted in E'_i, the happens-before ordering between x and y doesn't change the possible behaviors of actions in E_i and E'_i. Any action that happens-before x or y happens-before both of them. If either x or y happens-before an action, both of them do (excepting, of course, x and y themselves). Thus, the reordering of x and y can't affect the write seen by any uncommitted read.

Similarly, the reordering doesn't affect which (if any) incorrectly synchronized write a read can be made to see when the read is committed.

If E'_i is used to justify committing x in E'_{i+1}, then E_i may be used to justify committing x in E_{i+1}. Similarly for y.

If one or both of x or y is committed in E'_i, it can also be committed in E_i, without behaving any differently, with one caveat. If y is a lock or a volatile read, it is possible that committing x in E'_i will force some synchronization actions that happen-before y to be committed in E'_i. However, we are allowed to commit those actions in E_i, so this does not affect the existence of E_i.

Thus, the sequences of executions used to justify E' will also justify E, and the program transformation is legal. □

9.1.3 Other Code Transformations

Although the Java memory model is not defined in terms of which transformations are legal, it is possible to derive the legality of a transformation based on the model. For example,

- synchronization on thread local objects can be ignored or removed altogether (the caveat to this is the fact that invocations of methods like `wait` and `notify` have to obey the correct semantics – for example, even if the lock is thread local, it must be acquired when performing a `wait`),
- redundant synchronization (e.g., when a synchronized method is called from another synchronized method on the same object) can be ignored or removed,
- volatile fields of thread local objects can be treated as normal fields.

9.2 Considerations for Programmers

The most important property of the memory model that is provided for programmers is the notion that if a program is

Initially, `x == y == 0`

Thread 1	Thread 2
do { r1 = x;	do { r2 = y;
} while (r1 == 0);	} while (r2 == 0);
y = 42;	x = 42;

Correctly synchronized, so non-termination is the only legal behavior

Figure 13: Correctly Synchronized Program

correctly synchronized, it is unnecessary to worry about reorderings. In this section, we prove this property holds of the Java memory model. First, we prove a lemma that shows that when each read sees a write that happens-before it, the resulting execution behaves in a sequentially consistent way. We then show that reads in executions of correctly synchronized programs can only see writes that happen-before them. Thus, by the lemma, the resulting behavior of such programs is sequentially consistent.

9.2.1 *Correctly synchronized programs exhibit only sequentially consistent behaviors*

Lemma 2. *Consider an execution E of a correctly synchronized program P that is legal under the Java memory model. If, in E, each read sees a write that happens-before it, E has sequentially consistent behavior.*

Proof:
Since the execution is legal under the memory model, the execution is happens-before consistent and synchronization order consistent.

A topological sort on the happens-before edges of the actions in an execution gives a total order consistent with program order and synchronization order. Let r be the first read in E that doesn't see the most recent conflicting write w in the sorted order but instead sees w'. Let the topological sort of E be $\alpha w' \beta w \gamma r \delta$.

Let $\alpha w' \beta w \gamma r' \delta'$ be the topological sort of an execution E'. E' is obtained exactly as E, except that instead of r, it performs the action r', which is the same as r except that it sees w; δ' is any sequentially consistent completion of the program such that each read sees the previous conflicting write.

The execution E' is sequentially consistent, and it is not the case that $w' \xrightarrow{hb} w \xrightarrow{hb} r$, so P is not correctly synchronized.

Thus, no such r exists and the program has sequentially consistent behavior. □

Theorem 3. *If an execution E of a correctly synchronized program is legal under the Java memory model, it is also sequentially consistent.*

Proof: By Lemma 2, if an execution E is not sequentially consistent, there must be a read r that sees a write w such that w does not happen-before r. The read must be committed, because otherwise it would not be able to see a write that does not happen-before it. There may be multiple reads of this sort; if so, let r be the first such read that was committed. Let E_i be the execution that was used to justify committing r.

The relative happens-before order of committed actions and actions being committed must remain the same in all executions considering the resulting set of committed actions. Thus, if we don't have $w \xrightarrow{hb} r$ in E, then we didn't have $w \xrightarrow{hb} r$ in E_i when we committed r.

Since r was the first read to be committed that doesn't see a write that happens-before it, each committed read in E_i must see a write that happens-before it. Non-committed reads always sees writes that happens-before them. Thus, each read in E_i sees a write that happens-before it, and there is a write w in E_i that is not ordered with respect to r by happens-before ordering.

A topological sort of the actions in E_i according to their happens-before edges gives a total order consistent with program order and synchronization order. This gives a total order for a sequentially consistent execution in which the conflicting accesses w and r are not ordered by happens-before edges. However, Lemma 2 shows that executions of correctly synchronized programs in which each read sees a write that happens-before it must be sequentially consistent. Therefore, this program is not correctly synchronized. This is a contradiction. □

10. RELATED WORK

Hardware architectures have motivated most prior work on memory models. Lamport provides one of the earliest discussions of memory models [25], which provides the widely used definition for sequential consistency. Several relaxed models have been proposed in academia and for commercial hardware [4, 6, 12, 17, 19, 31, 40, 43]. Adve and Gharachorloo provide a primer for this work [3]. Adve and Hill [4, 6] and Gharachorloo et al. [17] first formalized the property of sequential consistency for data-race-free (or properly-labeled [17]) programs for memory models. The data-race-free work also observed that models such as happens-before are insufficient to provide this property [6, 7].

There has also been substantial work on relaxed models in the area of runtime systems for software distributed shared-memory [9, 22]. Much of this work has built on the hardware models.

Our work is primarily motivated by requirements for programming languages, which differ significantly from those for hardware-driven memory models. Specifically, the out-of-thin-air requirement stems from security and safety properties of programming languages. Formalizing this requirement in a way that would not prohibit compiler transformations was particularly challenging. In particular, the amount of analysis and transformation that a compiler might perform is essentially unlimited. Thus, devising a memory model that doesn't interfere with desirable compiler transformations is significantly harder; it is also undesirable, as it would require a major restructuring of current compilers and incur a significant performance penalty.

Not surprisingly, all of the above hardware-driven memory models either allow unacceptable out-of-thin-air behavior for incorrectly synchronized programs (e.g., allow the behavior shown in Figure 2), or they do not allow dependence-breaking transformations as shown in Figures 5 and 6.

In programming languages such as C [23] and C++ [41], threads are not part of the language specification. Instead, libraries such as POSIX threads (pthreads) [27] support multi-threading. Since the C/C++ spec doesn't talk about threads and the pthreads specification doesn't talk about the C/C++ language features, various elements of the semantics can be left unspecified. For example, it is unclear if double-checked locking [37] works in C as described in various publications or if there is a way of modifying it to work. In contrast, the Java specification makes it clear that standard double-checked locking idiom only works if the checked field is volatile.

After the flaws in the original Java memory model were pointed out [33], a number of proposals for replacement memory models for Java emerged. Most of these are based around techniques originally used for hardware memory models.

The various proposals for revising the Java memory model differed in a number of details, such as whether they allowed removal of synchronization on thread-local objects and whether they defined special semantics for final fields. Due to space limitations, we are unable to enumerate all of the differences between this work and previous proposals. However, none of the previous proposals for revising the Java memory model adequately handled cases involving the apparent reordering of data and control dependences, such as shown in Figures 5 and 6. This deficiency was even in previous work [28] by two of the authors of this paper. The SC- model [5] handled Figures 5 and 6, but not Figure 10.

Maessen, Arvind and Shen [8] present an operational semantics for Java threads based on the Commit / Reconcile / Fence protocol [39]. The CRF semantics prohibits reordering of control and data dependences, and thus prohibits the behaviors in Figures 5 and 6.

Yang, Gopalakrishnan and Lindstrom [44, 45] attempt to characterize the approaches taken in [8, 28, 29] using a single, unified memory model (called UMM). They also [44] use the same notation to propose their own memory model for Java. In their semantics, an action can only be performed early if the action is guaranteed to occur in all executions. This disallows a variety of possible optimizations, including the transformation seen in 5 and 6.

Kotrajaras [24] proposes a memory model for Java that is based on the original, flawed model. It suffers not only from the complexity of that model, but from its reliance on a single, global shared memory.

Saraswat [36] presents a memory model for Java based on solving a system of constraints between actions for a unique fixed point, rather than depending on control and data dependence. Thus, Saraswat's model allows the behavior in Figure 5; given which writes are seen by each read, there is a unique fixed point solution. However, Saraswat's model does not allow the behavior in Figure 6; in that example, given the binding of reads to writes, there are multiple fixed-point solutions.

The ECMA specification for the Common Language Infrastructure (CLI) provides a memory model [13]. However, it is vague and informal; as a result it seems impossible to determine whether that model allows or disallows behaviors such as shown in Figures 2, 4 – 6 and 10 – 13.

It is also worth noting that some of the Microsoft engineers have published articles [11] in which they claim that the CLI specification is too relaxed, and that they have written code as part of Microsoft's core libraries that won't work according to the ECMA spec.

11. CONCLUSION

In this paper, we have outlined the necessary properties for a programming language memory model, and outlined how those properties can be achieved. The resulting model balances two crucial needs: it allows implementors flexibility in their ability to perform code transformations and optimizations, and it also provides a clear and simple programming model for those writing concurrent code.

The model meets the needs of programmers in several key ways:

- For data-race-free programs, it allows programmers to reason about their programs using the simple semantics of sequential consistency, oblivious to any compiler and hardware transformations.
- It provides a clear definition for the behavior of programs in the presence of data races, including the most comprehensive treatment to date of the dangers of causality and how they can be avoided.

This paper clarifies and formalizes these needs, balancing them carefully with a wide variety of optimizations and program transformations commonly performed by compilers and processor architectures. It also provides proof techniques to ensure that the model reflects this balancing act accurately and carefully. It is this balance that has given us the necessary confidence to use this model as the foundation for concurrency in the Java programming language.

Acknowledgments

Many other people have made significant contributions to this work, all those who participated in discussions on the JMM mailing list. The total set of people that have contributed is far too large to enumerate, but certain people made particularly significant contributions, including Doug Lea, Victor Luchangco, Jan-Willem Maessen, Hans Boehm, Joseph Bowbeer, and David Holmes. We also thank Mark Hill for comments on this paper.

12. 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.* PhD thesis, University of Wisconsin, Madison, December 1993. Ph.D. Thesis.

[3] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] 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.

[5] Sarita V. Adve. The SC- memory model for Java, 2004. http://www.cs.uiuc.edu/∼sadve/jmm.

[6] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[7] Sarita V. Adve and Mark D. Hill. Sufficient conditions for implementing the data-race-free-1 memory model. Technical Report #1107, Computer Sciences Department, University of Wisconsin-Madison, September 1992.

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

[9] John K. Bennett, John B. Carter, and Willy Zwanaepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990.

[10] Gianfranco Bilardi and Keshav Pingali. A Framework for Generalized Control Dependence. In *ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, United States, June 1996.

[11] Christopher Brumme. C# memory model. http://blogs.msdn.com/cbrumme/archive/2003/05/17/51445.apx.

[12] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Ann. Intl. Symp. on Computer Architecture*, pages 434–442, June 1986.

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

[14] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors*. PhD thesis, Stanford University, 1996.

[15] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[16] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. Intl. Conf. on Parallel Processing*, pages I355–I364, 1991.

[17] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.

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

[19] *IBM System/370 Principles of Operation*, May 1983. Publication Number GA22-7000-9, File Number S370-01.

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

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

[22] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. 19th Ann. Intl. Symp. on Computer Architecture*, pages 13–21, 1992.

[23] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[24] Vishnu Kotrajaras. *Towards an Improved Memory Model for Java*. PhD thesis, Department of Computing, Imperial College, August 2001.

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

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

[27] Bil Lewis and Daniel J. Berg. *Multithreaded programming with pthreads*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1998.

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

[29] Jeremy Manson and William Pugh. Semantics of Multithreaded Java. Technical Report CS-TR-4215, Dept. of Computer Science, University of Maryland, College Park, March 2001.

[30] Jeremy Manson and William Pugh. Requirements for Programming Language Memory Models. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, Newfoundland, Canada, July 2004.

[31] C. May et al., editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufmann, San Francisco, 1994.

[32] Andy Podgurski and Lori Clarke. A formal model of program dependences and its implications for software testing debugging and maintenance. *IEEE Transactions on Software Engineering*, 1990.

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

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

[35] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, 1997.

[36] Vijay Saraswat. Concurrent Constraint-based Memory Machines: A framework for Java Memory Models. Technical report, IBM TJ Watson Research Center, March 2004.

[37] Douglas Schmidt and Tim Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *3rd Annual Pattern Languages of Program Design Conference*, 1996.

[38] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, April 1988.

[39] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. *Proceedings of the 26th International Symposium on Computer Archite cture*, 1999.

[40] R. L. Sites and R. T. Witek, editors. *Alpha AXP Architecture Reference Manual*. Digital Press, Boston, 1995. 2nd edition.

[41] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, Reading Mass. USA, 3rd edition, 1997.

[42] Z. Sura, C.-L. Wong, X. Fang, J. Lee, S. Midkiff, and D. Padua. Automatic implementation of programming language consistency models. In *Proc. of the 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, 2002. To appear Lecture Notes in Computer Science, Springer-Verlag.

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

[44] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Specifying Java thread semantics using a uniform memory model. In *ACM Java Grande Conference*, November 2001.

[45] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Formalizing the Java memory model for multithreaded program correctness and optimization. Technical Report UUCS-02-011, University of Utah, April 2002.