ABSTRACT

Title of Dissertation: The Java Memory Model

Jeremy Manson, Doctor of Philosophy, 2004

Dissertation directed by: William Pugh
Department of Computer Science

After many years, support for multithreading has been integrated into mainstream programming languages. Inclusion of this feature brings with it a need for a clear and direct explanation of how threads interact through memory. Programmers need to be told, simply and clearly, what might happen when their programs execute. Compiler writers need to be able to work their magic without interfering with the promises that are made to programmers.

Java's original threading specification, its *memory model*, was fundamentally flawed. Some language features, like volatile fields, were under-specified: their treatment was so weak as to render them useless. Other features, including fields without access modifiers, were over-specified: the memory model prevents almost all optimizations of code containing these "normal" fields. Finally, some features, like final fields, had no specification at all beyond that of normal fields; no additional guarantees were provided about what will happen when they are used.

This work has attempted to remedy these limitations. We provide a clear and concise definition of thread interaction. It is sufficiently simple for programmers to work with, and flexible enough to take advantage of compiler and processor-level optimizations. We also provide formal and informal techniques for verifying that the model provides this balance. These issues had never been addressed for any programming language: in addressing them for Java, this dissertation provides a framework for all multithreaded languages. The work described in this dissertation has been incorporated into the version 5.0 of the Java programming language.

The Java Memory Model

by

Jeremy Manson

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:

Michael Hicks
Pete Keleher
Edgar G. K. Lopez-Escobar
Adam Porter
William Pugh, Chair/Advisor

ACKNOWLEDGEMENTS

The work in this thesis could not have even begun to speculate about the merest possibility of crossing my mind without the input of many, many people. First and foremost is my advisor, Bill Pugh. However, the contributions of the many members of the Java memory model mailing list were also crucial – especially Doug Lea and Sarita Adve. Although the others are too numerous to mention, I shall mention some of them anyway - Victor Luchangco, Hans Boehm, Joseph Bowbeer and David Holmes all had a significant impact on the content of this dissertation.

I would like to thank David Hovemeyer and Jaime Spacco for sitting through innumerable conversations and lectures on the Java memory model. They truly have the patience of saints.

Without Penelope Asay, I would have been even more of an emotional wreck than the typical graduate student in the throes of dissertation

writing. I might have gotten it done, but I wouldn't have been very happy about it.

Special thanks to my Mother, Father, Josh and Matthew for not making me explain what I am doing. I would still be on the phone. And, of course, I wouldn't be anywhere without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A facility for quotation covers the absence of original thought.

– Dorothy L. Sayers, "Gaudy Night"

Much of the work done in modern computer science focuses on one of two goals. A tremendous amount of effort has been funneled into ensuring that these two goals are met. Unfortunately, in modern programming environments, these goals can conflict with each other in surprising ways.

First, we want to make sure that programs run quickly. At the high level, this involves data structure and algorithm design. At a lower level, this involves investing a great deal of time and effort reordering code to ensure that it is run in the most efficient way possible.

For modern compilers and processors, this lower level is crucial. Speculative execution and instruction reordering are two tools that have been integral to the explosion in computer power over the last several decades. Most compiler optimizations, including instruction scheduling, register allocation, common subexpression elimination and redundant read elimination, can be thought of as relying on instruction reordering to some extent. For example, when a redundant read is removed, the compiler is, in effect, moving the redundant read to where

the first read is performed. In similar ways, processor optimizations such as the use of write buffers and out-of-order completion / issue reflect reorderings.

Our second goal is to ensure that both programmers and computers understand what we are telling them to do. Programmers make dozens of assumptions about the way code is executed. Sometimes those assumptions are right, and sometimes they are wrong. In the background, at a lower level than our programming efforts, compilers and architectures are spending a lot of time creating an environment where our code is transformed; it is optimized to run on the system. Our assumptions about the way code is executing can be easily violated.

It is fairly easy (by comparison) for a processor, dealing with a single thread of instructions, to avoid messing around too much with our notions of how instructions are scheduled. It simply needs to ensure that when an instruction is performed early, that instruction doesn't affect any of the instructions past which it was moved. Programmers will generally not need to reason about potential reorderings when they write single-threaded programs. This model actually allows for a wide variety of program transformations.

The real difficulty comes when there is more than one thread of instructions executing at the same time, and those threads are interacting. The processor can ensure that the actions in each thread appear to happen in the correct order in isolation. However, if more than one thread is executing, the limiting factors we impose for single-threaded execution are not enough – because of the need for code optimization, we can start to see bizarre side effects.

This is not necessarily a problem. For most modern multithreaded programming patterns, the programmer ensures that there are ordering constraints by explicitly communicating between threads. Within these specified constraints,

Original code

Initially: `p == q`, `p.x == 0`

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r1.x;`	

Result: `r2 == r5 == 0, r4 == 3`

Valid compiler transformation

Initially: `p == q`, `p.x == 0`

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r2;`	

Result: `r2 == r5 == 0, r4 == 3`

Figure 1.1: Surprising results caused by forward substitution

it is possible to reorder code with a great deal of freedom. However, this begs the question of how that communication works, and what happens in a program when it is missing.

Obviously, we have to spend some time specifying these issues. We need a lingua franca, which we usually call a *memory model*, because it describes how programs interact with memory. The memory model for whatever language the programmer is using defines what kind of reorderings may be perceived, down to the level of machine code. In a high-level interpreted language, like Java or C#, it defines the reorderings the compiler can perform when translating to bytecode, the virtual machine can perform when translating to machine code, and the processor can perform when executing that machine code.

A simple example of how reorderings can be perceived can be seen in Figure 1.1 [Pug99]. One common compiler optimization involves having the value read for `r2` reused for `r5`: they are both reads of `r1.x` with no intervening write.

Now consider the case where the assignment to `r6.x` in Thread 2 happens

Initially, x == y == 0

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

Can r1 == r2 == 42?

Figure 1.2: Motivation for disallowing some cycles

between the first read of `r1.x` and the read of `r3.x` in Thread 1. If the compiler decides to reuse the value of `r2` for the `r5`, then `r2` and `r5` will have the value 0, and `r4` will have the value 3. From the perspective of the programmer, the value stored at `p.x` has changed from 0 to 3 and then changed back. It was this surprising example the first motivated this work.

This example is a relatively simple one; if the examples were all so simple, a memory model could be defined in terms of allowable reorderings. Not a great deal of subtlety would be required. However, this is not the case; we have to deal with the notion of a *causal loop*.

Consider Figure 1.2. We have two threads of execution, both of which read from one variable and then write to another. If these actions are executed out of their program order, we could imagine that, for example, the number 42 was written to x in Thread 2, then read from x in Thread 1, written to y in Thread 1, then read from y in Thread 2, justifying the initial write to x. Where did the value 42 come from?

This example may seem glib and unrealistic to some. However, a complete semantics for a programming language memory model would have to make a decision whether or not to allow it. In this case, the causal loop leads to an undesirable behavior: one which we wish to disallow.

Initially, `x == y == 0`

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`r2 = r1 | 1;`	`x = r3;`
`y = r2;`	

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

Figure 1.3: Compilers Can Think Hard about when Actions are Guaranteed to Occur

Another example of a causal loop – this time, one that describes an acceptable behavior – can be seen in Figure 1.3. In order to see the result `r1 == r2 == r3 == 1`, it would seem as if Thread 1 would need to write 1 to `y` before reading `x`. It also seems as if Thread 1 can't know what value `r2` will be until after `x` is read.

In fact, a compiler could 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 statement containing the logical or operator (—) will result 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`.

It is clear, therefore, that sometimes causal loops are acceptable, and other times, they are not. The fundamental problem with these sorts of examples is that there is no "first cause" from which we can reason. We have to think some more about cause and effect. The progress we have made in this area – the deep thinking about what sort of behaviors are desirable in a multithreaded program – is one of the most important contributions of this dissertation.

1.1 Why Solve This Problem?

In the past, multithreaded languages have not defined a full semantics for multithreaded code. Ada, for example, simply defines unsynchronized code as "erroneous" [Ada95]. The reasoning behind this is that since such code is incorrect (on some level), no guarantees should be made for that code. 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 rarely happens. 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.

1.2 Approach

The approach taken in this dissertation was done in two overlapping stages. First, we gathered requirements from an experienced community of professional software engineers, hardware architects and compiler / virtual machine designers. In the coming chapters, you may notice references to guarantees and optimizations that are "reasonable" or "unreasonable"; these notions came from long discus-

sions with this group. The requirements that were gathered are largely described in Chapters 2 and 3. When the requirements were gathered, we designed a model that reflected those requirements (Chapters 4 and 5), and verified that the final specification met the requirements (Chapter 6).

The requirements largely centered around two goals:

- We needed to define, in a careful way, how programmers could ensure that actions taken by one thread would be seen in a reasonable way by other threads.

- We needed to define, in a careful way, what could happen if programmers did not take care to ensure that threads were communicating in the proscribed manners.

The first bullet (arguably the more simple of the two goals) required a careful and complete definition of what constitutes *synchronization*, which allows threads to communicate. *Locking*, described in Chapter 2, is the most obvious way for two threads to communicate. However, other mechanisms are necessary. For example, it is often the case that we want threads to communicate without the overhead of mutual exclusion. To address this, we introduced the notion of a `volatile` field (Chapter 3) : this supplants the more traditional memory barrier as a basic memory coherence operation. Finally, we wanted to present a way that a programmer could ensure object immutability regardless of data races: the resulting semantics for `final` fields are outlined in Chapter 7.

The second bullet is somewhat more complex. The fundamental question is described above: what kind of causal loops are acceptable, and what kind must be disallowed? This question frames the discussion of causality (Chapter 4).

1.3 Development

A final note: this work has been done in the context of the Java programming language [GJS96]; Java is used as a running example. 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 was not defined in a way that allowed programmers and architects to understand the requirements for a Java system. For example, it disallowed the program transformation seen in Figure 1.1. For more details on this, see Chapter 8.

The work described in this dissertation has been adapted to become part of Java Specification Request (JSR) 133 [Jav04], a new memory model for Java, which is included in the 5.0 release of the Java programming language.

Chapter 2

Building Blocks

Time is an illusion. Lunchtime doubly so.

– Douglas Adams, The Hitchhiker's Guide to the Galaxy

In order to reason about cause and effect in a programming language, we have to do two things:

1. Provide a clear model for concurrent programmers to write correct code.
2. Provide a clear model for platform architects to optimize code.

There has been a lot of work in defining, conceptually, how threads interact through memory. In order to develop a memory model that fulfills out requirements, it is necessary to review some concepts and vocabulary first. This chapter provides that review.

2.1 Code is Reordered

The most commonly assumed memory model is *sequential consistency* [Lam79]. Sequential consistency specifies that memory actions will 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 prior to any

optimizations (the *program order*). This model basically reflects an interleaving of the actions in each thread; under sequential consistency, a read must return the value written most recently to the location being read in the total order.

While this is an intuitive extension of the single threaded model, sequential consistency restricts the use of many system optimizations. In general, a sequentially consistent system will not have much freedom to reorder memory statements within a thread, even if there are no conventional data or control dependences between the two statements.

Many of the 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.

Many different aspects of a system may affect reorderings. For example, the just-in-time compiler and the processor may perform reorderings. The memory hierarchy of the architecture on which a virtual machine is run may make it appear as if code is being reordered. Source code to bytecode transformation can also reorder and transform programs. For the purposes of simplicity, we shall simply refer to anything that can reorder code as being a compiler.

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

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 2.1: Behaves Surprisingly

Initially, x == y == 0

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

May return r2 == 2, r1 == 1

Figure 2.2: Figure 2.1, after reordering

transformations occurred at all.

However, it is much more difficult to maintain a sequentially consistent semantics while optimizing multithreaded code. Consider Figure 2.1. 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 (as seen in Figure 2.2. 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.

2.2 Synchronization and Happens-Before

To some programmers, the behavior demonstrated in Figure 2.1 may seem "broken". However, multithreaded programming languages provide built-in mechanisms to provide constraints when two threads interact: such mechanisms are referred to as *synchronization*. The code in Figure 2.1 does not use these mechanisms.

The Java programming language provides multiple mechanisms for synchronizing threads. The most common of these methods is *locking*, which is implemented using *monitors*. Each object in Java is associated with a monitor, which a thread can *lock* or *unlock*. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation.

There is a total order over all synchronization operations, called the *synchronization order*. We say that an unlock action on monitor m *synchronizes-with* all lock actions on m that come after it (or are *subsequent* to it) in the synchronization order.

We say that one action *happens-before* another [Lam78] in three cases:

- if the first action comes before the second in program order, or
- if the first action synchronizes-with the second, or
- if you can reach the second by following happens-before edges from the first (in other words, happens-before is transitive).

Note that all of this means that happens-before is a partial order: it is reflexive, transitive and anti-symmetric. There are more happens-before edges, as

described in Chapter 5 – we will just focus on these for now. We now have the key to understanding why the behavior in Figure 2.1 is legal. Here is what we observe:

- there is a write in one thread,
- there is a read of the same variable by another thread and
- the write and read are not ordered by happens-before.

In general, when there are two accesses (reads of or writes to) the same shared field or array element, and at least one of the accesses is a write, we say that the accesses *conflict* (regardless of whether there is a data race on those accesses) [SS88]. When two conflicting accesses are not ordered by happens-before, they are said to be in a *data race*. When code contains a data race, counterintuitive results are often possible.

We use the discussion of data races to define what it means for a program to be correctly synchronized. A program is *correctly synchronized* if and only if all sequentially consistent executions are free of data races. The code in Figure 2.1, for example, is incorrectly synchronized.

When a program is not correctly synchronized, errors tend to crop up in three intertwined ways. *Atomicity* deals with which actions and sets of actions have indivisible effects. This is the aspect of concurrency most familiar to programmers: it is usually thought of in terms of mutual exclusion. *Visibility* determines when the effects of one thread can be seen by another. *Ordering* determines when actions in one thread can be seen to occur out of order with respect to another. In the rest of this chapter, we discuss how those problems can arise.

2.3 Atomicity

If an action is (or a set of actions are) *atomic*, its result must be seen to happen "all at once", or indivisibly. Atomicity is the traditional bugbear of concurrent programming. Enforcing it means using locking to enforce mutual exclusion.

The Java programming language has some minor issues related to the atomicity of individual actions. Specifically, writes of 64-bit scalar values (longs and doubles) are allowed to take place in two separate atomic 32 bit "chunks". If a thread is scheduled between the separate writes, it may see the first half of the 64-bit value, but not the second. In other circumstances, we assume that all individual read and write actions take place atomically. This is not an aspect of the memory model that we shall address in any greater depth.

Atomicity can also be enforced on a sequence of actions. A program can be free from data races without having this form of atomicity. However, enforcing this kind of atomicity is frequently as important to program correctness as enforcing freedom from data races. Consider the Java code in Figure 2.3. Since all accesses to the shared variable `balance` are guarded by synchronization, the code is free of data races.

Now assume that one thread calls `deposit(5)` and another calls `withdraw(5)`; there is an initial balance of 10. Ideally, at the end of these two calls, there would still be a balance of 10. However, consider what would happen if:

- The `deposit()` method sees a value of 10 for the balance, then
- The `withdraw()` method sees a value of 10 for the balance **and** withdraws 5, leaving a balance of 5, and finally
- The `deposit()` method uses the balance it originally saw (10) to calculate

```
class BrokenBankAccount {
  private int balance;

  synchronized int getBalance() {
    return balance;
  }

  synchronized void setBalance(int x)
    throws IllegalStateException {
    balance = x;
    if (balance < 0) {
      throw new IllegalStateException("Negative Balance");
    }
  }

  void deposit(int x) {
    int b = getBalance();
    setBalance(b + x);
  }

  void withdraw(int x) {
    int b = getBalance();
    setBalance(b - x);
  }
}
```

Figure 2.3: Atomicity example

a new balance of 15.

As a result of this lack of atomicity, the balance is 15 instead of 10. This effect is often referred to as a *lost update* because the withdrawal is lost. A programmer writing multi-threaded code must use synchronization carefully to avoid this sort of error. In the Java programming language, if the `deposit()` and `withdraw()` methods are declared `synchronized`, it will ensure that locks are held for their duration: the actions of those methods will be seen to take place atomically.

Note that atomicity guarantees from synchronization are relative. By synchronizing on an object, you can guarantee that a sequence of actions is perceived to happen atomically by other threads that synchronize on the same object. But threads that do not synchronize on that object may not see the actions occur atomically.

Atomicity is the most well-studied problem when using synchronization. It is a common mistake to think that it is the only problem; it is not. In the rest of this chapter, we look at two other important issues that may crop up when writing multithreaded code.

2.4 Visibility

If an action in one thread is *visible* to another thread, then the result of that action can be observed by the second thread. In order to guarantee that the results of one action are observable to a second action, then the first must happen before the second.

Consider the code in Figure 2.4. Now imagine that two threads are created; one thread calls `work`, and at some point, the other thread calls `stopWork` on the same object. Because there is no happens-before relationship between the two

```
class LoopMayNeverEnd {
  boolean done = false;

  void work() {
    while (!done) {
      // do work
    }
  }

  void stopWork() {
    done = true;
  }
}
```

Figure 2.4: Visibility example

threads, the thread in the loop may never see the update to `done` performed by the other thread. In practice, this may happen if the compiler detects that no writes are performed to `done` in the first thread; the compiler may hoist the read of `done` out of the loop, transforming it into an infinite loop.

To ensure that this does not happen, there must be a happens-before relationship between the two threads. In `LoopMayNeverEnd`, this can be achieved by declaring `done` to be `volatile`. Conceptually, all actions on volatiles happen in a single total order, and each write to a volatile field happens-before any read of that volatile that occurs later in that order.

There is a side issue here; some architectures and virtual machines may execute this program without providing a guarantee that the thread that executes `work` will ever give up the CPU and allow other threads to execute. This would prevent the loop from ever terminating because of scheduling guarantees, not because of a lack of visibility guarantees. For more discussion of this issue, see Section 3.6.

2.5 Ordering

Ordering constraints govern the order in which multiple actions are seen to have happened. The ability to perceive ordering constraints among actions is only guaranteed to actions that share a happens-before relationship with them. We have already seen a violation of ordering in Figure 2.1; here is another.

The code in Figure 2.5 shows an example of where the lack of ordering constraints can produce surprising results. Consider what happens if `threadOne()` gets invoked in one thread and `threadTwo()` gets invoked on the same object in another. Would it be possible for `threadTwo()` to return the value `true`?

```
class BadlyOrdered {
  boolean a = false;
  boolean b = false;

  void threadOne() {
    a = true;
    b = true;
  }

  boolean threadTwo() {
    boolean r1 = b; // sees true
    boolean r2 = a; // sees false
    boolean r3 = a; // sees true
    return (r1 && !r2) && r3; // returns true
  }

}
```

Figure 2.5: Ordering example

If `threadTwo()` returns true, it means that the thread saw both updates by `threadOne`, but that it saw the change to `b` before the change to `a`.

The Java memory model allows this result, illustrating a violation of the ordering that a user might have expected. This code fragment is not correctly synchronized (the conflicting accesses are not ordered by a happens-before ordering).

If ordering is not guaranteed, then the assignments to `a` and `b` in `threadOne()` can be seen to be performed out of order. Compilers have substantial freedom to reorder code in the absence of synchronization. For example, the compiler is allowed to freely reorder the writes in `threadOne` or the reads in `threadTwo`.

To avoid this behavior, programmers must ensure that their code is correctly synchronized.

2.6 Discussion

The discussion in this chapter centers around some of the basic issues of memory semantics. We have given some form to the problem: we have outlined the basic ideas behind memory models, synchronization and happens-before, and we have seen how misinterpretation of how multithreading works can lead to problems with atomicity, visibility and ordering. However, this skirts the major issue: how can we define multithreaded semantics in a way that allows programmers to avoid these pitfalls, but still allows compiler designers and computer architects to optimize run time?

To answer this question, we still need to gather more requirements. What kinds of compiler optimizations do we need? What guarantees are sufficient for correctly synchronized code? What guarantees are sufficient for incorrectly

synchronized code? In the chapters ahead, we address these questions.

Chapter 3

In Which Some Motivations Are Given

The part can never be well unless the whole is well.

- Plato, Charmides

The last chapter outlined some informal properties that all concurrent programs have, and some ways in which those properties could be used incorrectly, leading to broken code. This chapter outlines some of the informal requirements that the memory model has – in other words, ways in which these broken programs can be tamed.

Many compiler writers and programmers contributed to the discussions that led to the conclusions in this chapter. These conclusions were only reached after 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, which makes predicting the results of a multithreaded program less straightforward.

At the end of this process, a consensus emerged as to what the informal re-

quirements for a programming language memory model are. As the requirements emerged, a memory model took shape. Most of these requirements have relatively simple to understand motivations – obvious guarantees that need to be made, optimizations that need to be allowed, and so on. These are documented in this chapter. The other requirements – those relating to causality – are more complicated. The causality requirements, together with the memory model, are described in full in Chapter 4.

3.1 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 to avoid having to do this sort of reasoning.

For example, in the code in Figure 3.2, 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.

Prior work [Adv93, Gha96, AH90] has shown that one of the best guarantees that can be provided is that reorderings should only be visible when code is incorrectly synchronized. This is a strong guarantee for programmers, whose assumptions about how multithreaded code behaves almost always include sequential consistency (as defined in Section 2.1). Our first guarantee for programmers (which we call DRF), therefore, applies to correctly synchronized programs (as defined in Section 2.2):

DRF Correctly synchronized programs have sequentially consistent semantics.

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 3.1: Surprising Correctly Synchronized Program

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 3.1 ([Adv93]) is correctly synchronized. This may seem surprising, since it doesn't perform any synchronization actions. Remember 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 to `x` or `y` will occur. Since no writes occur, there can be no data races: the program is correctly synchronized.

This disallows some subtle program transformations. For example, an aggressive write speculation could predict that the write to `y` of 42 was going to occur, allowing Thread 2's read of `y` to see that write. This would cause the write of 42 to `x` to occur, allowing the read of `x` to see 42; this would cause the write to `y` to occur, justifying the speculation! This sort of transformation is **not** legal; as a correctly synchronized program, only sequentially consistent results should be allowed.

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 3.2: Behaves Surprisingly

3.2 Simple Reordering

In earlier chapters, we outlined how important it is for compilers to reorder program actions – it is the engine that drives most optimizations. Our first requirement, therefore, is that we always allow statements that are not control or data dependent on each other in a program to be reordered:

Reorder1 Independent adjacent statements can be reordered.

Note that Reorder1 actually allows the reordering of statements that were not adjacent in the original program: it is simply necessary to perform repeated reorderings until the statement appears in the desired location.

In a multithreaded context, doing this may lead to counter-intuitive results, like the one in Figure 2.1 (reprinted here as Figure 3.2). Remember that the key notion behind this figure is that the actions in each thread are reordered; once this happens, even a machine that executes code in a sequentially consistent way can result in the behavior shown. However, it should be noted again that that code is improperly synchronized: there is no ordering of the accesses by synchronization. When synchronization is missing, weird and bizarre results are allowed.

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 3.3: Effects of Redundant Read Elimination

The Reorder1 guarantee ensures 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, actions cannot generally be reordered out of locking regions.

The reordering in Figure 3.2 does not interfere too heavily with our notion of cause and effect. Specifically, there is no reason to think that the first action in either of these threads has an effect on the second – causality is still served.

3.3 Transformations that Involve Dependencies

In Section 3.2, 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 3.3 is allowed. This behavior may seem cyclic, as the write to `y` is control dependent on the reads of `x`, which see a write to `x` that is data dependent on a read of `y` which must see the aforementioned write to `y`. However, 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.

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`. In that case, either `r1 == 0` and `r2 == 1`, or `r1 == 1` and `r2 == 0`.

Figure 3.4 shows another surprising behavior. In order to see the result `r1 == r2 == r3 == 1`, it would seem as if Thread 1 would need to write 1 to `y` before reading `x`. It also seems as if Thread 1 can't know what value `r2` will be until after `x` is read.

In fact, a compiler could 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 operation with the bitwise or will always return 1, resulting

Initially, `x == y == 0`

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`r2 = r1 | 1;`	`x = r3;`
`y = r2;`	

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

Figure 3.4: An Example of a More Sophisticated Analysis

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 3.5: Sometimes Dependencies are not Obvious

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`. This kind of 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 3.5. As it does for Figure 3.4, a compiler can determine that only the values 0 and 1 are ever written to `x`. As a result, the compiler can remove the dependency in Thread 2 and move the write to `x` to the start of that thread. If the resulting code were executed in a sequentially consistent way, it would result in the circular behavior described.

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 3.6: An Unexpected Reordering

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. As was mentioned before, actions can usually not be moved outside locking regions. Another example of this will be shown later (Figure 3.11).

Even though Reorder1 and Reorder2 are strong guarantees for compilers, they are not a complete set of allowed reorderings. They are simply a set that is always guaranteed to be allowed. More transformations are possible; it is simply necessary to ensure that the results are allowable by the memory model.

3.3.1 Reordering Not Visible to Current Thread

Figure 3.6 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

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 3.7: Simple Use of Volatile Variables

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.

3.4 Synchronization

We haven't really discussed how programmers can use explicit synchronization (in whatever form we give it) to make sure their code is correctly synchronized. In general, we use synchronization to enforce the happens-before relationships that we briefly discussed in Chapter 2. 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 example of their use can be seen in Figure 3.7. 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`. Communicating between threads in this way is clearly useful for non-blocking algorithms (such as, for example, wait free queues [MS96]). Volatiles are discussed in more detail in Section 3.5.2.

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, the *happens-before* property:

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* is defined as it was in Chapter 2 (reiterated here for your convenience). *Synchronization actions* include locks, unlocks, and reads of and writes to volatile variables. There is 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.

The *happens-before relationship* between two actions described in Chapter 2 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 notion may seem at odds with the results shown in many of our examples (for example, Figure 3.2).

However, a "reordering" is only visible in these examples if we are reasoning about both threads. The two threads do not share a happens-before relationship,

so we do not reason about both. The individual threads can only be examined in isolation. When this happens, no reordering is visible; the only mystery is 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 create happens-before relationships between threads. We call such relationships *synchronizes-with* relationships. 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`.

3.5 Additional Synchronization Issues

3.5.1 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.

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.

Synchronization Removal

There have been many optimizations proposed ([Ruf00, CGS+99]) 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

Before Coarsening	After Coarsening
`x = 1;`	`synchronized(someLock) {`
`synchronized(someLock) {`	`  x = 1;`
`  // lock contents`	`  // lock contents`
`}`	`  y = 2;`
`y = 2;`	`}`

Figure 3.8: Example of Lock Coarsening

that are already captured by the program order edges. This lock is redundant, and can therefore be removed.

Lock Coarsening

One transformation that is frequently effective in increasing concurrency is *computation lock coarsening* [DR98]. If a computation frequently acquires and releases the same lock, then computation lock coarsening can coalesce those multiple locking regions into a single region. This requires the ability to move accesses that occur outside a locking region inside of it, as seen in Figure 3.8.

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.

This is reflected in Figure 3.8. The write to `x` could have been scheduled before or after the last unlock of `someLock`. By moving it inside the `synchronized` block, the compiler is merely ensuring that it was scheduled after that unlock.

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 == 1, r3 == 2, r2 == r4 == 0` legal behavior?

Figure 3.9: Volatiles Must Occur In A Total Order

Similarly, the release ensures an ordering with a subsequent acquire. Consider an action that takes place after a release. It may or may not be visible to 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. This can also be seen in Figure 3.8, where the write to `y` is moved up inside the `synchronized` block.

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 – which of these is required, and on what platform, is discussed in greater detail in Chapter 8 and in [Lea04].

Figure 3.9 gives us another interesting glimpse into the guarantees we provide

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 3.10: Strong or Weak Volatiles?

to programmers. The reads of `v1` and `v2` should be seen in the same order by both Thread 3 and Thread 4; if they are not, the behavior `r1 == 1, r3 == 2, r2 == r4 == 0` can be observed. Specifically, Thread 3 sees the write to `v1`, but not the write to `v2`; Thread 4 sees the write to `v2`, but not the write to `v1`).

The memory model prohibits this behavior: it 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.

3.5.2 Additional Guarantees for Volatiles

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. This interpretation reflects the ordering constraints on synchronization variables in the memory model referred to as *weak ordering* [DSB86, AH90].

In Figure 3.10, 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 3.2.

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.

3.5.3 Optimizers Must Be Careful

Optimizers have to consider volatile accesses as carefully as they consider locking. In Figure 3.11, 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.

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 3.11: Another Surprising Correctly Synchronized Program

Even though we know that the write to `a` will always happen, we cannot reorder that write with the loop. If we did perform that reordering, Thread 1 would be able to see the value 1 for `a`, and perform the 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.

3.6 Infinite Executions, Fairness and Observable Behavior

The Java specification does not guarantee preemptive multithreading or any kind of fairness guarantee. There is no hard guarantee that any thread will surrender

Thread 1	Thread 2

`while (true)`
`  synchronized (o) {`
`  if (done) break;`
`  think`
`  }`
	`synchronized (o) {`
	`  done = true;`
	`}`

Figure 3.12: Lack of fairness allows Thread 1 to never surrender the CPU

Initially, v is volatile and v = false

Thread 1	Thread 2
`while (!v);`	`v = true;`
`print("Thread 1 done");`	`print("Thread 2 done");`

Figure 3.13: If we observe print message, Thread 1 must see write to `v` and terminate

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 implementations). Most Java 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.

To many, it may seem as if this is not a memory model issue. However, the issues are closely related. An example of their interrelation can be seen in Figure 3.12. Due to the lack of fairness, it is legal for the CPU running Thread 1 never to surrender the CPU to Thread 2; thus the program may never terminate. Since this behavior is legal, it is also legal for a compiler to hoist the `synchronized` block outside the while loop, which has the same effect.

This is a legal compiler transformation, but an undesirable one. As mentioned in Section 3.5.1, the compiler is allowed to perform lock coarsening (e.g., if the compiler sees two successive calls to synchronized methods on the same object, it doesn't have to give up the lock between calls). The exact tradeoffs here are subtle, and improved performance needs to be balanced by the longer delays threads will suffer in trying to acquire a lock.

However, there should be some limitations on compiler transformations that reduce fairness. For example, in Figure 3.13, 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. This motivates another requirement:

Observable The only reason that an action visible to the external world (e.g., a

Initially, `x == y == 0`

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

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

Figure 3.14: Correctly Synchronized Program

file read / write, program termination) might not be externally observable is if there is an infinite sequence of actions that might happen before it or come before it in the synchronization order.

3.6.1 Control Dependence

As a result of some of these requirements, the new Java memory model makes subtle but deep changes 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 3.14. 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 moved before the loops. However, this would be illegal in Java. This program is 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 3.1). A compiler must create a situation where the loops terminate.

The notion of control dependence that correctly encapsulates this is called *weak control dependence* [PC90] in the context of program verification. This property has also been restated as *loop control dependence* [BP96] in the context of program analysis and transformation.

Chapter 4

Causality – Approaching a Java Memory Model

It is an old maxim of mine that when you have excluded the impossible, whatever remains, however improbable, must be the truth.

- Sir Arthur Conan Doyle, The Adventure of the Beryl Coronet

In Section 2.1, we described sequential consistency. It is too strict for use as the Java memory model, because it forbids standard compiler and processor optimizations. We must formulate a better memory model. In this chapter, we address this need by carefully examining the necessary guarantees for unsynchronized code. In other words, we address the question: what is acceptable behavior for a multithreaded program? Answering this question allows us to synthesize our requirements and formulate a workable memory model.

4.1 Sequential Consistency Memory Model

Section 2.1 discusses the implications of sequential consistency. For convenience, it is presented again here, and formalized.

In sequential consistency, all actions occur in a total order (the execution order). The actions in the execution order occur in the same order they do in the program (that is to say, they are consistent with program order). Furthermore,

each read r of a variable `v` sees the value written by the write w to `v` such that:

- w comes before r in the execution order, and
- there is no other write w' such that w comes before w' and w' comes before r in the execution order.

4.2 Happens-Before Memory Model

We can describe a simple, interesting memory model using the HB guarantee introduced in Chapter 3 by abstracting a little from locks and unlocks. We call this model the *happens-before memory model*. Many of the requirements of our simple memory model are built out of the requirements in Chapters 2 and 3:

- There is a total order over all synchronization actions, called the *synchronization order* (Section 2.2).
- Synchronization actions induce *synchronizes-with* edges between matched actions. Together with program order, these two relationships form the *happens-before order* (Section 3.4).
- A volatile read sees the value written by the previous volatile write in the synchronization order (Section 3.4).

The only additional constraint on the Happens-Before Memory Model is that the value seen by a normal read is determined by *happens-before consistency*. Formally, we say that it is *happens-before consistent* for a read r of a variable v to see a write w to v if:

- w is ordered before r by happens-before and there is no intervening write w' to v (i.e., $w \xrightarrow{hb} r$ and there is no w' such that $w \xrightarrow{hb} w' \xrightarrow{hb} r$) or

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 4.1: Behaves Surprisingly

- w and r are not ordered by happens-before (i.e., it is not the case that $w \stackrel{hb}{\rightarrow} r$ or $r \stackrel{hb}{\rightarrow} w$).

Less formally, 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 to `x`, and the write of 2 happens-before a read of `x`, then that read cannot see the value 1. Alternatively, it can be happens-before consistent for the read to see the write if the read is not ordered by happens-before with the write.

As an example, consider Figure 4.1 (the same as Figures 3.2 and 2.1). The code in this figure has two writes and two reads; the reads are not ordered by happens-before with the write of the same variable. Therefore, it is happens-before consistent for the reads to see the writes.

If all of the reads in an execution see writes which it is happens-before consistent for them 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.

As far as first approximations go, the Happens-Before Memory Model is not a bad one. It provides some necessary guarantees. For example, the result of the

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 4.2: Surprising Correctly Synchronized Program

code in Figure 4.1 is happens-before consistent: the reads are not prevented from seeing the writes by any happens-before relationship.

Unfortunately, happens-before consistency is not a good memory model; it simply isn't sufficient. Figure 4.2 (the same as Figure 3.1) shows an example where happens-before consistency does not produce the desired results. Remember that this code is correctly synchronized: if it is executed under sequential consistency, neither the write to `y` nor the write to `x` can occur, because the reads of those variable will see the value 0. Therefore, there are no data races in this code. However, under the happens-before memory model, if both writes occur, it is happens-before consistent for the reads to see those writes. Even though this is a correctly synchronized program, under happens-before consistency, executions that are not sequentially consistent are legal. Therefore, happens-before consistent executions can violate our DRF guarantee.

Nevertheless, happens-before consistency provides a good outer bound for our model; based on HB, all executions must be happens-before consistent.

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 4.3: An Out Of Thin Air Result

4.3 Causality

Basic Notions

So, the happens-before memory model provides necessary constraints on our final memory model, but it is not complete. We saw why this was when examining Figure 4.2, in which the writes are control dependent on the reads. A similar example can be seen in Figure 4.3. In this case, the writes will always happen, and the values written are data dependent on the reads.

The happens-before memory model also allows the undesirable result in this case. Say that the value 42 was written to x in Thread 2. Then, under the happens-before memory model, it would be legal for the read of x in Thread 1 to see that value. The write to y in Thread 1 would then write the value 42. It would therefore be legal for the read of y in Thread 2 to see the value 42. This allows the value 42 to be written to x in Thread 2. The undesirable result justifies itself, using a circular sort of reasoning.

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*.

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 4.4: An Unexpected Reordering

In fact, the behavior shown in Figure 4.3 may be even more of a cause for concern than the behavior shown in Figure 4.2. 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 4.4. 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 4.3, 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 4.1. 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. The transformations we have examined all involve moving actions earlier than they would otherwise have occurred. For example, to get the out-of-thin-air result in Figure 4.3, we have, in some sense, moved the write of 42 to `y` in Thread 1 early, so that the read of `y` in Thread 2 can see it and allow the write to `x` to occur. The read of `x` in Thread 1 then sees the value 42, and justifies the execution of the write to `y`.

If we assume that the key to these issues is to consider when actions can be moved early, then we must consider this issue carefully. The question to answer is, what is it that causes an action to occur? When can the action be performed early? One potential answer to this question involves starting at the point where we want to execute the action, and then considering what would happen if we carried on the execution in a sequentially consistent way from that point. If we did, and it would have been possible for the action to have occurred afterward, then perhaps the action can be considered to be caused.

In the above case, we identify whether an action can be performed early by identifying some *well-behaved* execution in which it takes place, and using that execution to justify performing the action. Our model therefore builds an execution iteratively; it allows an action (or a group of actions) to be *committed*

(in essence, performed early) if it occurs in some well-behaved execution that also contains the actions committed so far. Obviously, this needs a base case: we simply assume that no actions have been committed.

The resulting model can therefore be described with two, iterative, phases. Starting with some committed set of actions, generate all the possible "well-behaved" executions. Then, use those well-behaved executions to determine further actions that can be reasonably performed early. Commit those actions. Rinse and repeat until all actions have been committed.

Identifying what entails a "well-behaved" execution is crucial to our model, and key to our notions of causality. If we had, for example, a write that was control dependent on the value of `r1 + r2` being equal to 3 in Figure 4.1, 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 notion of well-behavedness to our other example, as well. In Figure 4.1, the writes to `x` and `y` can occur first because they will always occur in sequentially consistent executions. In Figure 3.3, 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 4.3, 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 (but not only) "out of thin air" guarantee:

ThinAir1 A write can occur earlier in an execution than it appears in program order. However, that write must have been able to occur without the assumption that any subsequent reads see non-sequentially consistent values.

This is only a first approximation to causality: it, too, is a good starting point, but does not cover all of our bases.

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 4.5: 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 4.6: Can Threads 1 and 2 see 42, if Thread 4 didn't write to `x`?

4.3.1 When Actions Can Occur

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 4.3 provides an example of a result that is clearly unacceptable, but other examples may be less straightforward.

The examples in Figures 4.5 and 4.6 are similar to the examples in Figures 4.2 and 4.3, with one major distinction. In those examples, the value 42 could never be written to `x` in any sequentially consistent execution. Thus, our ThinAir1 guarantee prevented the value 42 from appearing. In the examples in Figures 4.5 and 4.6, 42 can be written to `x` in some sequentially consistent executions (specif-

ically, ones in which the write to `z` in Thread 3 occurs before the read of `z` in Thread 4). Should these new examples also get an out-of-thin-air guarantee, even though they are not covered by our previous guarantee? In other words, 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 4.5 and 4.6.

Allowing Other Results

Now consider the code in Figure 4.7. 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.

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 4.7: A Complicated Inference

This is a reasonable transformation that needs to be balanced with the out-of-thin-air requirement. Notice that the code in Figure 4.7 is quite similar to the code in Figures 4.5 and 4.6. 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 consistent results that the write does not occur. The ThinAir1 guarantee is therefore not strong enough to encapsulate this notion.

The examples in Figure 4.5 and 4.6 are significantly different from the example in Figure 4.7. One way of articulating this difference is that in Figure 4.7, 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 4.5 and 4.6, we need to reason about the outcome of a data race to determine that `r1 = x` can see 42.

This, then, is another, stronger way of differentiating out-of-thin-air reads. A read is not considered to be out-of-thin-air if you can determine whether it happens without considering data races that influenced its execution. This is also our second out-of-thin-air principle:

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 4.8: Another Out Of Thin Air Example

ThinAir2 An action can occur earlier in an execution than it appears in program order. However, that write must have been able to 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 Figures 4.5 and 4.6. We want to determine if a write of 42 to `y` can be performed earlier than its original place in the program. We therefore examine the actions that happen-before it. Those actions (a read of `x`, and the initial actions) do not allow for us to determine that a write of 42 to `y` occurred. Therefore, we cannot move the write early.

As another example, consider the code in Figure 4.8. In this example, the only way in which `r2` could be set to 1 is if `r1` was not 0. In this case, `r3` would have to be 1, which means `r2` would have to be 1. The value 1 in such an execution clearly comes out of thin air. 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.

The memory model that results from this causality constraint is our final

model. We build executions based on the notion that they must be "well-behaved" (as described in Section 4.3). However, the executions must be "well-behaved" based on the notion that additional reads cannot see any values from data races: a "well-behaved" execution requires that all reads see writes that happen-before them. The model is presented formally in the next chapter.

4.4 Some Things That Are Potentially Surprising about This Memory Model

Many of the requirements and goals for the Java memory model were straightforward and non-controversial (e.g., DRF). 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. This section describes some of the implications of the memory model that are not a result of requirements, but, rather, more artifacts of our reasoning.

4.4.1 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 considered 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 4.6. If we allowed the unacceptable execution, then we could say 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, 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.

Initially, `a = b = c = d = 0`

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

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

Figure 4.9: Behavior disallowed by semantics

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 4.10: Result of thread inlining of Figure 4.9; behavior allowed by semantics

4.4.2 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 4.9. An implementation that always scheduled Thread 1 before Thread 2 and Thread 2 before Thread 3 could conceivably 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 4.10, we have another example, similar to the one in Figure 4.9, where Threads 1, 2 and 3 are combined (or "inlined"). We can use the same reasoning that we were going to use for Figure 4.9 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 example.

As a result of this distinction, it is clear that when a compiler performs thread inlining, the resulting thread is not necessarily allowed to be treated in the same way as a thread that was written "inline" in the first place. Thus, a 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

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 4.11: A variant "bait-and-switch" behavior

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 4.12: Behavior that must be allowed

can execute.

Another example of how thread inlining is generally disallowed can be seen from Figures 4.11 and 4.12. The behavior in Figure 4.11 is very similar to that shown in Figure 4.6. Given that our isolation principle states that the behavior in Figure 4.6 is unacceptable, it seems reasonable to prohibit both.

Figure 4.12 shows a code fragment very similar to that of Figure 4.11. However, for the code in Figure 4.12, we must allow the behavior that was prohibited in Figure 4.11. We do this because that behavior can result from well understood

and reasonable compiler transformations.

- 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 4.11 is prohibited and the behavior in Figure 4.12 is allowed is, perhaps, surprising. However, this is another example of inlining: we could derive Figure 4.12 from Figure 4.11 by combining Threads 1 and 2 into a single thread.

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 per, for example, ThinAir1 and ThinAir2). The happens-before relationship is used to express causality between two actions; if an additional happens-before relationship is inserted, the causal relationships change.

Chapter 5

The Java Memory Model

Everything should be as simple as possible, but no simpler.

– Albert Einstein

This chapter provides the formal specification of the Java memory model (excluding final fields, which are described in Chapter 7).

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, (normal or non-volatile) read, (normal or non-volatile) write, lock, unlock or other synchronization action. Volatile reads, volatile writes, locks and unlocks are synchronization actions, as are the (synthetic) first and last action of a thread, actions that start a thread or detect that a thread has terminated, as described in Section 5.2. There are also external actions, and thread divergence actions

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.

$\overset{hb}{\rightarrow}$ - 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 5.5. 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 has 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 5.5.

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 number of additional thread divergence actions.

5.2 Definitions

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

   They are defined as follows:

   - An unlock action on monitor m synchronizes-with all subsequent lock

actions on m (where subsequent is defined according to the synchronization order).

- A write to a volatile variable v synchronizes-with all subsequent reads of v by any thread (where subsequent is defined according to the synchronization order).
- An action that starts a thread synchronizes-with the first action in the thread it starts.
- The final action in a thread T1 synchronizes-with any action in another thread T2 that detects that T1 has terminated. T2 may accomplish this by calling `T1.isAlive()` or doing a join action on T1.
- If thread T1 interrupts thread T2, the interrupt by T1 synchronizes-with any point where any other thread (including T2) determines that T2 has been interrupted (by invoking `Thread.interrupted`, invoking `Thread.isInterrupted`, or by having an `InterruptedException` thrown).
- The write of the default value (zero, false or null) to each variable synchronizes-with to the first action in every thread. Although it may seem a little strange to write a default value to a variable before the object containing the variable is allocated, conceptually every object is created at the start of the program with its default initialized values. Consequently, the default initialization of any object happens-before any other actions (other than default writes) of a program.
- At the invocation of a finalizer for an object, there is an implicit read of a reference to that object. There is a happens-before edge from the end of a constructor of an object to that read. Note that all freezes for

this object (see Section 7.1) happen-before the starting point of this happens-before edge .

2. **Definition of happens-before**. If we have two actions x and y, we use $x \xrightarrow{hb} y$ to mean that x *happens-before* y. If x and y are actions of the same thread and x comes before y in program order, then $x \xrightarrow{hb} y$. If an action x synchronizes-with a following action y, then we also have $x \xrightarrow{hb} y$. Further more, happens-before is transitively closed. In other words, if $x \xrightarrow{hb} y$ and $y \xrightarrow{hb} z$, then $x \xrightarrow{hb} z$.

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 is 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 properly 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 Java language specification 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.

  If A is finite, then the sequence $C_0, C_1, \ldots$ will 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 $E_i = \langle P, A_i, \overset{po_i}{\rightarrow}, \overset{so_i}{\rightarrow}, W_i, V_i, \overset{sw_i}{\rightarrow}, \overset{hb_i}{\rightarrow} \rangle$.

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. $\overset{hb_i}{\rightarrow}|_{C_i} = \overset{hb}{\rightarrow}|_{C_i}$
3. $\overset{so_i}{\rightarrow}|_{C_i} = \overset{so}{\rightarrow}|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. 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 $W_i(r) \overset{hb_i}{\rightarrow} r$
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 $\overset{ssw_i}{\rightarrow}$ be the $\overset{sw_i}{\rightarrow}$ edges that are in the transitive reduction of $\overset{hb_i}{\rightarrow}$ but not in $\overset{po_i}{\rightarrow}$. We call $\overset{ssw_i}{\rightarrow}$ the sufficient synchronizes-with edges for E_i. If $x \overset{ssw_i}{\rightarrow} y \overset{hb_i}{\rightarrow} z$ and $z \in C_i - C_{i-1}$, then $x \overset{sw_j}{\rightarrow} y$ 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 $x \overset{hb_i}{\rightarrow} y$, then $x \in C_i$.

5.5 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 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.

5.6 Formalizing Observable Behavior

To reason about observable behaviors, we need to talk about sets of observable actions. 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 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.

Initially, `x = y = 0`

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

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

Figure 5.1: A Standard Reordering

5.7 Examples

There are a number of examples of behaviors that are either allowed or prohibited by the Java memory model. Most of these are either deliberately prohibited examples that show violations of the causality rules, or permissible examples that seem to be a violation of causality, but can result from standard compiler optimizations. In this section, we examine how some of these examples can be worked through the formalism.

5.7.1 Simple Reordering

As a first example of how the memory model works, consider Figure 5.1. 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 be obtained 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.

The set of actions C_0 is the empty set, and there is no execution E_0. As a result of this, execution E_1 will be an execution where all reads see writes that happen-before them, as per Rule 6. 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.

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 5.2: Table of commit sets for Figure 5.1

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.

5.7.2 Correctly Synchronized Programs

It is easy to see how most of the guarantees that we wished to provide for volatile variables are fulfilled; Section 5.2 explicitly provides most of them. A slightly

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 5.3: A Correctly Synchronized Program

more subtle issue (mentioned in Section 3.5.3) is reproduced as Figure 5.3. This code is correctly synchronized: in all sequentially consistent executions, the read of `a` by Thread 1 sees the value 0; the volatile variable `v` is therefore written, and there is a happens-before relationship between that read of `a` by Thread 1 and the write to `a` in Thread 2.

In order for the read of `a` to see the value 1 (and therefore result in a non-sequentially consistent execution), the write to `a` must be committed before the read. We may commit that write first, in E_1. We then would try to commit the read of `a` by Thread 1, seeing the value 1. However, Rule 2 requires that the happens-before orderings between an action being committed and the actions already committed remain the same when the action is committed. In this case, we are trying to commit the read, and the write is already committed, so the read must happen-before the write in E_2. This makes it impossible for the read to see the write.

Note that in addition to this, Rule 8 states that any release-acquire pair that happens-before the write to `a` when that write is committed must be present in

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

Thread 1	Thread 2	Thread 3	Thread 4
join Thread 4	join Thread 4	v = 1	r3 = v
r1 = x	r2 = y		if (r3 == 1) {
y = r1	x = r2		x = 1
			y = 1
			}

Behavior in question:r1 = r2 = 1, r3 = 0

Figure 5.4: Must Disallow Behavior by Ensuring Happens-Before Order is Stable

C_i. This requirement implies that the volatile accesses would have to be added to C_i. Since they have to be present in the final execution for the write to take place early, the data race is not possible.

Rule 8 is not redundant; it has other implications. For example, consider the code in Figure 5.4. We wish to prohibit the behavior where the read of `v` returns 0, but Threads 1 and 2 return the value 1. Because `v` *is* volatile, Rule 8 protects us. In order to commit one of the writes in Threads 1 or 2, the release-acquire pair of (`v = 1`, `r3 = v`) must be committed before the reads in Threads 1 and 2 can see the value 1 – `r3` must be equal to 1.

Note that if `v` were not volatile, the read in Thread 4 would have to be committed to see the write in Thread 3 before the reads in Threads 1 and 2 could see the value 1 (by Rule 6).

5.7.3 Observable Actions

Figure 5.5 is the same as Figure 3.13 in Section 3.6. Our requirement for this program was that if we observe the print message from Thread 2, and no other

Initially, v is volatile and v = false

Thread 1	Thread 2
`while (!v);`	`v = true;`
`print("Thread 1 done");`	`print("Thread 2 done");`

Figure 5.5: If we observe print message, Thread 1 must see write to `v` and terminate

threads other than Threads 1 and 2 run, then Thread 1 must see the write to `v`, print its message and terminate. The compiler should not be able to hoist 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 described in Section 5.5. 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.

Chapter 6

Simulation and Verification

You cannot proceed formally from an informal specification

In Chapter 3, we looked at the informal requirements for our memory model, exposing the principles that guided its development. In Chapter 5, we provided a formal specification for the model. However, as the maxim goes, you cannot proceed formally from an informal specification. How do we know that the formal specification meets our informal criteria?

The answer is clearly verification. For the Java memory model, we did two types of verification. The first was a simulator, which was useful for determining that the behavior of the model was correct on individual test cases. The other was a set of proofs that the properties that we specified (in Chapter 3) for our memory model did, in fact, hold true for it.

This chapter is broken into two parts. Section 6.1 describes the simulator: how it works and its runtime complexity. Section 6.2 recapitulates the properties that we outlined for our memory model, and discusses how we ensured that those properties were realized.

6.1 The Simulator

As shown in Chapters 3 and 4, the informal requirements for our memory model were motivated by a dozen (or so) test cases. When developing the memory model, we actually formulated dozens of these test cases, each of which was created specifically to demonstrate a particular property that our model had to have. Every change we made to the model could conceivably affect any number of those properties; we needed a way to review the test cases systematically. Being computer scientists, we naturally developed the idea of simulating the model.

The simulator can be used in a number of ways. For example, you could feed a program P into the simulator to obtain a set of results R. Then, you can apply a compiler transformation by hand to convert P into P', and feed P' into the simulator, giving a set of results R'. The transformation from P to P' is legal if and only if $R' \subseteq R$. The insight here is that while a compiler can do transformations that eliminate possible behaviors (e.g., performing forward substitution or moving memory references inside a synchronized block), transformations must not introduce new behaviors.

The simulator provides three important benefits:

- It gives us confidence that the formal model means what we believe, and that we believe what it means.
- As we fine tuned and modified the formal model, we gained confidence that we were only changing the things we intended to change.
- The formal model is not easy to understand; it is likely that only a subset of the people who need to understand the Java memory model will understand the formal description of model. People such as JVM implementors and

authors of books and articles on thread programming may find the simulator a useful tool for understanding the memory model.

6.1.1 Simulating Causality

The simulator is built on top of a global system that executes one operation from one thread in each step, in program order. This is used to provide all possible interleavings of a given program, when the actions appear in the interleavings in the same order in which they occur in the program. This framework has been useful in simulating several versions of the model [MP01a, MP01b].

Although this is a useful framework, it does not allow for the full flexibility of the model. As discussed in Chapter 4, the difficulty in developing a sufficient model lies in determining which actions can occur early (out of program order). The notion of causality underlies this; how can an action be performed early (or *presciently*) while also ensuring that the action is not causing itself to occur?

A basic simulator that executes instructions in program order does not have to deal with causality, because no actions are performed early. However, it is also incomplete, because it does not provide results of executions with prescient actions. A naïve solution would be to attempt to perform all actions early, at every possible place where they might be performed. However, this is far too computationally expensive. The single greatest difficulty in developing a simulator is in determining where prescient actions may be executed.

An early version of the simulator was able to use some heuristics to decide where early reads could occur, but it largely relied on manual placement [MP02]. However, this is an incomplete approach; that simulator could not generate and verify all legal executions.

However, this is not as computationally expensive as it sounds. Remember that the formal model is given an execution E to verify, and takes an iterative, two phase approach (as mentioned in Chapters 4 and 5):

1. Starting with some set of actions that are ensured to occur (the "committed" actions), generate all possible executions that include those actions, and in which all reads that are not committed see writes that happen-before them.

2. Given those resulting executions, determine what actions can be added to the set of "committed" actions: these represent additional actions that can be performed early.

This process is repeated until there are no actions left to be added to the set of committed actions. If the resulting execution E' is the same as the execution E provided, then E' is a legal execution. The simulator therefore simply generates all sets of legal executions.

The tricky part, of course, is picking the next actions to commit, while still preventing the execution time from being too long. A simple answer to this can be found in the model itself. The "obvious" extension (as mentioned in the first bullet above) is to include all executions where reads see only writes that happen before those reads. The only executions that this does not produce are ones where a read r is in a data race with a write w, and r returns the value of w. It is therefore simply a matter of detecting writes that are involved in data races with r, and committing those writes so that r can see them. As an optimization, we only commit those writes that can actually allow the read to see a value that it did not see in the justifying execution.

To detect data races, the simulator uses a technique commonly used for dynamic on-the-fly data race detection [Sch89, NR86, DS90]. The technique is quite simple; it works by examining the partial ordering induced by the happens-before relationship (referred to as a *partially-ordered execution graph* in the literature). If two accesses in the graph are not strictly ordered with respect to each other, and one is a write, then they are in a data race. This is easily supported by the simulator, which needs to keep track of the happens-before relationships anyway.

The simulator simply commits writes that are in data races, then commits reads in the same data races that see those writes. It then executes all interleavings the program, ensuring that uncommitted reads can only see the result of writes that happen-before them (as described in Chapter 5). Additionally, it should be noted that when actions in a data race are committed, acquire and release actions that happen-before them also have to be committed (as per Rule 8 in Section 5.4).

Most of the other functional details are easy to extract from the definition of the model as found in Chapter 5. One additional point is worth noting. While the simulator is generating executions, it is important that it ensure that those executions obey the well-formedness criteria listed in Section 5.3. The simulator must, for example, ensure that volatile variables read the correct values and that intra-thread consistency is maintained.

6.1.2 Simulator Performance

In general, the performance of the simulator scales with the number of data races in a program. We can take some time to describe this in more detail. Consider a multithreaded program with t threads, where the i^{th} thread has n_i

instructions and the total number of instructions in all threads is n. The number of interleavings of a given multithreaded program can be determined by taking the total number of permutations of all instructions ($n!$) and removing those permutations where the instructions occur out of program order. Since, for each thread, there are $n_i!$ orderings in which the instructions occur out of program order, we have:

$$\frac{n!}{\prod_{i=1}^{n}(n_i!)}$$

Since the simulator works by committing all permutations of data races in the program, we have a bound on runtime of (where dr is the number of data races in the program):

$$O(dr!(\frac{n!}{\prod_{i=1}^{n}(n_i!)}))$$

The number dr is a very rough one, as the number of data races in a program is by no means fixed. For example, a data race may be control or data dependent on the value seen by another data race read.

This runtime looks ugly, but it isn't actually so bad, as most of the examples we care about are relatively short. As they say, the proof of the pudding is in the eating: what is the practical runtime?

The programs we ran through the simulator ranges in size from 2 to 5 threads, and in length from 2 to 17 instructions. Most of our examples contain 1-3 data races. A sampling of the results, which can be seen in Table 6.1, indicate that for the kind of examples we have been concerned about in this dissertation, performance has not been much of an issue. The tests were performed under Java 1.5.0 beta 3 on a 1.8 GHz Pentium 4 Xeon with 1 GB of RAM.

Test Case	Elapsed Time (Seconds)
Figure 3.1	0.48
Figure 3.3	1.23
Figure 3.11	.54
Figure 4.3	0.48
Figure 4.5	5.12
Figure 4.6	2.85
Figure 4.12	1.14
Figure 5.1	.54

Table 6.1: Simulator Timing Results

The results are worth some discussion. Note, for example, that the fastest examples are those with no data races (Figures 3.1 and 3.11). Figure 4.3 (the simplest out of thin air example) is also fast: the data races in it cannot allow the reads to see different values, so those data races need not be committed.

An increase in thread count (as evidenced in Figures 4.5 and 4.6) increases the run time, but an increase in data races does so more: Figure 4.5 is a data dependent version of Figure 4.6, so it more frequently evinces data races. Thus, its run time is substantially higher (although not so substantially that the run time is meaningful).

6.2 Proofs

The most important guarantee that we can make that the Java memory model has the properties that we wish for it, of course, is to perform a set of proofs. The fact that the Java memory model provides many of the properties (reproduced in

Name	Description	Exemplar
Guarantees for Optimizers		
Reorder1	Independent adjacent statements can be reordered.	Figure 2.1
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 3.3, 3.4, 3.5, 3.6, 4.7
RS	Synchronization actions that only introduce redundant happens-before edges can be treated as if they don't introduce any happens-before edges.	Section 3.5.1
Guarantees for Programmers		
DRF	Correctly synchronized programs have sequentially consistent semantics.	Figures 3.1, 3.11
HB	Volatile writes are ordered before subsequent volatile reads of the same variable. Unlocks are ordered before subsequent locks of the same monitor.	Figure 3.7
VolatileAtomicity	All accesses to volatile variables are performed in a total order.	Figure 3.9
StrongVolatile	There is a happens-before relationship from each write to each subsequent read of that volatile.	Figure 3.10
ThinAir1	A write can occur earlier in an execution than it appears in program order. However, that write must have been able to occur without the assumption that any subsequent reads see non-sequentially consistent values.	Figures 4.3, 4.4, 4.8

Table 6.2: Table of Requirements, Part 1

Name	Description	Exemplar
ThinAir2	An action can occur earlier in an execution than it appears in program order. However, that write must have been able to occur in the execution without assuming that any additional reads see values via a data race.	Figures 4.5, 4.6, 4.9, 4.10
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 4.5, 4.6
Observable	The only reason that an action visible to the external world (e.g., a file read / write, program termination) might not be observable is if there is an infinite sequence of actions that might happen before it or come before it in the synchronization order.	Figure 3.13

Table 6.3: Table of Requirements, Part 2

Tables 6.2 and 6.3) is fairly straightforward. For example, the VolatileAtomicity property, the StrongVolatile property and the HB property are all explicitly stated as parts of the memory model.

The ThinAir1 and ThinAir2 properties are trivial extensions of the causality description. Actions can only be committed early if they occur in an execution where reads only see writes that happen-before them. Since no out of thin air reads occur in executions where reads only see writes that happen-before them, no writes can be committed that are dependent on out of thin air reads. Thus, the cycles that cause out-of-thin-air executions are not possible.

There are only really two key properties that require proof. In many ways, these two properties exemplify the fundamental dichotomy we faced in constructing the memory model. The first of these properties is that the semantics allows for reordering: this is the cornerstone of understanding why program transformations and optimizations are legal under the model. The second of these properties is that correctly synchronized programs obey sequentially consistent semantics: this is the strongest of the guarantees that we give to programmers. The rest of this section outlines the proofs that the model obeys these properties.

6.2.1 Semantics Allows Reordering

We mentioned earlier that a key notion for program optimization was that of *reordering*. We demonstrated in Figure 2.1 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. In compilers, instruction scheduling, register allocation, common sub-expression elimination and redundant read elimination all involve reordering.

On processors, the use of write buffers, out-of-order completion and out-of-order issue all require the use of reordering.

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 very common sample of 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)*
- *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*
- *Reordering s_x and s_y does not hoist an action above an infinite loop.*
- *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 a sequence of executions, $E'_0, E'_1, \ldots$ 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. □

6.2.2 Considerations for Programmers

The most important property of the memory model that is provided for programmers is the notion that if a program is correctly synchronized, it is unnecessary to worry about reorderings. In this section, we prove this property holds of the Java memory model.

Correctly Synchronized Programs Exhibit Only Sequentially Consistent Behaviors

As described in Chapter 2, we say an execution has *sequentially consistent* (SC) behavior if there is a total order over all actions consistent with the program order of each thread such that each read returns the value of the most recent write to the same variable. Two memory accesses are *conflicting* if they access the same variable and one or both of them are writes. A program is *correctly synchronized* if and only if in all sequentially consistent executions, all conflicting accesses to non-volatile variables are ordered by happens-before edges.

The most important property of the memory model that is provided for pro-

grammers is the notion that if a program is 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.

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 according to the memory model, the execution is synchronization order consistent and happens-before 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 sort but instead sees w'. Let the topological sort of E be $\alpha w' \beta w \gamma r \delta$.

Let E' be an execution whose topological sort is $\alpha w' \beta w \gamma r' \delta'$. 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 $w' \stackrel{hb}{\rightarrow} w \stackrel{hb}{\rightarrow} r$, so P is not correctly synchronized.

Thus, no such r exists and the program has sequentially consistent behavior. $\Box$

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. □

Chapter 7

Immutable Objects

I am constant as the northern star,
Of whose true-fix'd and resting quality
There is no fellow in the firmament.

– William Shakespeare, Julius Caesar (III, i, 60 – 62)

In Java, a *final* field is (intuitively) written to once, in an object's constructor, and never changed. The original Java memory model contained no mention of final fields. However, programmers frequently treated them as immutable. This resulted in a situation where programmers passed references between threads to objects they thought were immutable without synchronization. In this chapter, we cover how our memory model deals with final fields.

One design goal for the Java memory model was to provide a mechanism whereby an object can be immutable if all of its fields are declared final. This immutable object could be passed from thread to thread without worrying about data races. This relatively simple goal proved remarkably difficult, as this chapter describes.

Figure 7.1 gives an example of a typical use of final fields in Java. An object of

type `FinalFieldExample` is created by the thread that invokes `writer()`. That thread then passes the reference to a reader thread without synchronization. A reader thread reads both fields `x` and `y` of the newly constructed object.

Under Java's original memory model, it was possible to reorder the write to `f` with the invocation of the constructor. Effectively, the code:

```
r1 = new FinalFieldExample;
r1.x = 3;
r1.y = 4;
f = r1;
```

would be changed to:

```
r1 = new FinalFieldExample;
f = r1;
r1.x = 3;
r1.y = 4;
```

This reordering allowed the reader thread to see the default value for the final field and for the non-final (or *normal* field). One requirement for Java's memory model is to make such transformations illegal; it is now required that the assignment to `f` take place *after* the constructor completes.

A more serious example of how this can affect a program is shown in Figure 7.2. `String` objects are intended to be immutable; methods invoked on `String`s do not perform synchronization. This class is often implemented as a pointer to a character array, an offset into that array, and a length. This approach allows a character array to be reused for multiple `String` objects. However, this can create a dangerous security hole.

```
class FinalFieldExample {

  final int x;
  int y;
  static FinalFieldExample f;

  public FinalFieldExample() {
    x = 3;
    y = 4;
  }

  static void writer() {
    f = new FinalFieldExample();
  }

  static void reader() {
    if (f != null) {
      int i = f.x;
      int j = f.y;
    }
  }
}
```

A reader must never see x == 0

Figure 7.1: Example Illustrating Final Field Semantics

Thread 1	Thread 2
	`String myS = Global.s;`
`Global.s = "/tmp/usr".substring(4);`	`if (myS.equals("/tmp"))`
	`System.out.println(myS);`

Figure 7.2: Without final fields or synchronization, it is possible for this code to print `/usr`

In particular, if the fields of the `String` class are not declared final, then it would be possible for Thread 2 initially to see the default value of 0 for the offset of the `String` object, allowing it to compare as equal to `"/tmp"`. A later operation on the `String` object might see the correct offset of 4, so that the `String` object is perceived as being `"/usr"`. This is clearly a danger; many security features of the Java programming language depend upon `String`s being perceived as truly immutable.

In the rest of this chapter, we discuss the way in which final fields were formalized in the Java memory model. Section 7.1 lays out the full, informal requirements for the semantics of final fields. The bulk of the document discusses the motivation; the full semantics of final fields are presented in Section 7.3. Finally, we present some implementation issues in Section 7.5.

7.1 Informal Semantics

The detailed semantics of final fields are somewhat different from those of normal fields. In particular, we provide the compiler with great freedom to move reads of final fields across synchronization barriers and calls to arbitrary or unknown methods. Correspondingly, we also allow the compiler to keep the value of a final field cached in a register and not reload it from memory in situations where a

non-final field would have to be reloaded.

Final fields also provide a way to create thread-safe immutable objects that do not require synchronization. A thread-safe immutable object is seen as immutable by all threads, even if a data race is used to pass references to the immutable object between threads.

The original Java semantics did not enforce an ordering between the writer and the reader of the final fields for the example in Figure 7.1. Thus, the read was not guaranteed to see the write.

In the abstract, the guarantees for final fields are as follows. When we say an object is "reachable" from a final field, that means that the field is a reference, and the object can be found by following a chain of references from that field. When we say the "correctly initialized" value of a final field, we mean both the value of the field itself, and, if it is a reference, all objects reachable from that field.

- At the end of an object's constructor, all of its final fields are "frozen" by an implicit "freeze" action. The freeze for a final field takes place at the end of the constructor in which it was set. In particular, if one constructor invokes another constructor, and the invoked constructor sets a final field, the freeze for the final field takes place at the end of the invoked constructor.

- If a thread only reads references to an object that were written after the last freeze of its final fields, that thread is always guaranteed to see the frozen value of the object's final fields. Such references are called *correctly published*, because they are published after the object is initialized. There may be objects that are reachable by following a chain of references from such a final field. Reads of those objects will see values at least as up to

date as they were when the freeze of the final field was performed.

- Conversely, if a thread reads a reference to an object written before a freeze, that thread is not automatically guaranteed to see the correctly initialized value of the object's final fields. Similarly, if a thread reads a reference to an object reachable from the final field without reaching it by following pointers from that final field, the thread is not automatically guaranteed to see the value of that object when the field was frozen.

- If a thread is not guaranteed to see a correct value for a final field or anything reachable from that field, the guarantees can be enforced by a normal happens-before relationship. In other words, those guarantees can be enforced by normal synchronization techniques.

- When you freeze a final which points to an object, then freeze a final field of that object, there is a happens-before relationship between the first freeze and the second.

7.1.1 Complications

Retrofitting the semantics to the existing Java programming language requires that we deal with a number of complications:

- *Serialization* is the writing of an object to an input or output stream, usually so that object can be stored or passed across a network. When the object is read, that is called *deserialization* . Using serialization in Java to read an object requires that the object first be constructed, then that the final fields of the object be initialized. After this, deserialization code is invoked to set the object to what is specified in the serialization stream. This means

that the semantics must allow for final fields to change after objects have been constructed.

Although our semantics allow for this, the guarantees we make are somewhat limited; they are specialized to deserialization. These guarantees are not intended to be part of a general and widely used mechanism for changing final fields. In particular, you cannot make a call to native code to modify final fields; the use of this technique will invalidate the semantics of the VM.

To formalize the semantics for multiple writes / initializations of a final field, we allow multiple freezes. A second freeze action might, for example, take place after deserialization is complete.

- `System.in`, `System.out` and `System.err` are static final fields that allow access to the system's stdin, stdout and stderr files. Since programs often redirect their input and output, these fields are defined to be mutable by public methods. Thus, we give these three fields (and only these three fields) different semantics. This is discussed in detail in Section 7.3.

7.2 Motivating Examples

7.2.1 A Simple Example

Consider Figure 7.3. We will not start out with the complications of multiple writes to final fields; a freeze, for the moment, is simply what happens at the end of a constructor. Although `r1`, `r2` and `r3` can see the value `null`, we will not concern ourselves with that; that just leads to a null pointer exception.

The reference `q` is correctly published after the end of `o`'s constructor. Our

f1 is a final field; its default value is 0

Thread 1	Thread 2	Thread 3
o.f1 = 42	r1 = p;	r3 = q;
p = o;	i = r.f1;	j = r3.f1;
freeze o.f1	r2 = q;	
q = o;	if (r2 == r)	
	k = r2.f1;	

We assume r1, r2 and r3 do not see the value null. i and k can be 0 or 42, and j must be 42.

Figure 7.3: Example of Simple Final Semantics

semantics guarantee that if a thread only sees correctly published references to o, that thread will see the correct value for o's final fields. We therefore want to construct a special happens-before relationship between the freeze of o.f1 and the read of it as q.f1 in Thread 3.

The read of p.f1 in Thread 2 is a different case. Thread 2 sees p, an incorrectly published reference to object o; it was made visible before the end of o's constructor. A read of p.f1 could easily see the default value for that field, if a compiler decided to reorder the write to p with the write to o.f1. No read of p.f1 should be guaranteed to see the correctly constructed value of the final field.

What about the read of q.f1 in Thread 2? Is that guaranteed to see the correct value for the final field? A compiler could determine that p and q point to the same object, and therefore reuse the same value for both p.f1 and q.f1 for that thread. We want to allow the compiler to remove redundant reads of final fields wherever possible, so we allow k to see the value 0.

One way to conceptualize this is by thinking of an object being "tainted' for a thread if that thread reads an incorrectly published reference to the object. If an object is tainted for a thread, the thread is never guaranteed to see the object's correctly constructed final fields. More generally, if a thread t reads an incorrectly published reference to an object `o`, thread t forever sees a tainted version of `o` without any guarantees of seeing the correct value for the final fields of `o`.

In Figure 7.3, the object is not tainted for Thread 3, because Thread 3 only sees the object through `p`. Thread 2 sees the object through both both the `q` reference and the `p` reference, so it is tainted.

7.2.2 Informal Guarantees for Objects Reachable from Final Fields

In Figure 7.4, the final field `o.f2` is a reference instead of being a scalar (as it was in Section 7.2.1). It would not be very useful if we only guaranteed that the values read for references were correct, without also making some guarantees about the objects pointed to by those references. In this case, we need to make guarantees for `f2`, the object `p` to which it points and the array pointed to by `p.b`. Thread 1 executes `threadOne`, Thread 2 executed `threadTwo`, and Thread 3 executed `threadThree`.

We make a very simple guarantee: if a final reference is published correctly, and its correct value was guaranteed to be seen by an accessing thread (as described in Section 7.2.1), everything *transitively reachable* from that final reference is also guaranteed to be up to date as of the freeze. In Figure 7.4, `o`'s reference to `p`, `p`'s reference to `a` and the contents of `a` are all guaranteed to be seen by Thread 3. We call this idiom a *dereference chain*.

We make one exception to this rule. In Figure 7.4, Thread 2 reads `a[0]`

```
class First {

  final Second f2;

  static Second p = new Second();
  static First pub;
  static int [] a = {0};

  public First() {
    f2 = p;
    p.b = a;
    a[0] = 42;
  }
  public void threadOne() {
    pub = new First();
  }
  // First continues ...
  public void threadTwo() {
    int i = a[0];
    Second r1 = pub.f2;
    int [] r2 = r1.b;
    int r3 = r2[0];
  }
  public void threadThree() {
    Second s1 = pub.f2;
    int [] s2 = s1.b;
    int s3 = s2[0];
  }
}

class Second {
  int [] b;
}
```

We assume `r1` and `s1` do not see the value null.

`r2` and `s2` must both see the correct pointer to array `a`.

`s3` must be 42, but `r3` does not have to be 42.

Figure 7.4: Example of Transitive Final Semantics

Thread 1	Thread 2
o.f = p;	i = pub.x;
p.g = 42;	j = pub.y;
pub.x = o;	k = j.f;
freeze p.g;	l = k.g;
pub.y = o;	

Figure 7.5: Example of Reachability

through two different references. A compiler might determine that these references are the same, and reuse `i` for `r3`. Here, a reference reachable from a final field is read by a thread in a way that does not provide guarantees; it is not read through the final field. If this happens, the thread "gives up" its guarantees from that point in the dereference chain; the address is now tainted. In this example, the read of `a[0]` in Thread 2 can return the value 0.

The definition of reachability is a little more subtle than might immediately be obvious. Consider Figure 7.5. It may seem that the final field `p.g`, read as `k.g` in Thread 2, can only be reached through one dereference chain. However, consider the read of `pub.x`. A global analysis may indicate that it is feasible to reuse its value for `j`. `o.f` and `p.g` may then be read without the guarantees that are provided when they are reached from `pub.y`. As with normal fields, an apparent dependency can be broken by a compiler analysis (see Section 3.3 for more discussion of how this affects normal fields).

The upshot of this is that a reachability chain is not solely based on syntactic rules about where dereferences occur. There is a link in a dereference chain from any dynamic read of a value to any action that dereferences that value, no matter where the dereference occurs in the code.

Thread 1	Thread 2	Thread 3
`o.f1 = 42;`	`r1 = Global.a;`	`s1 = Global.b;`
`freeze o.f1;`	`Global.b = r2;`	`s2 = s1.f1;`
`Global.a = o;`		

`s2` is guaranteed to see 42, if `s1` is a reference to `o`.

Figure 7.6: Freezes are Passed Between Threads

7.2.3 Additional Freeze Passing

In this section, we will discuss some of the other ways that a read can be guaranteed to see a freeze.

Freezes Are Passed Between Threads

Figure 7.6 gives an example of another guarantee we provide. If `s1` is a reference to `o`, should `s2` have to see 42? The answer to this lies in the way in which Thread 3 saw the reference to `o`.

Thread 1 correctly published a reference to `o`, which Thread 2 then observed. Had Thread 2 then read a final field of `o`, it would have seen the correct value for that field; the thread would have to have ensured that it saw all of the updates made by Thread 1. To do this on SMP systems, Thread 2 does not need to know that it was Thread 1 that performed the writes to the final variable, it needs only to know that updates were performed. On systems with weaker memory constraints (such as DSMs), Thread 2 would need this information; we shall discuss implementation issues for these machines later.

How does this impact Thread 3? Like Thread 2, Thread 3 cannot see a reference to `o` until the freeze has occurred. Any implementation that allows

p.x is initialized to 42.

o.f is final.

Thread 1	Thread 2	Thread 3
lock m;	lock m;	k = Global.b;
Global.a = o;	i = Global.a;	l = k.x;
o.f = p;	Global.b = i;	
freeze o.f;	j = i.x;	
unlock m	unlock m;	

If the unlock in Thread 1 happens-before the unlock in Thread 2:

i will see o.

j will see 42.

k will see o or null.

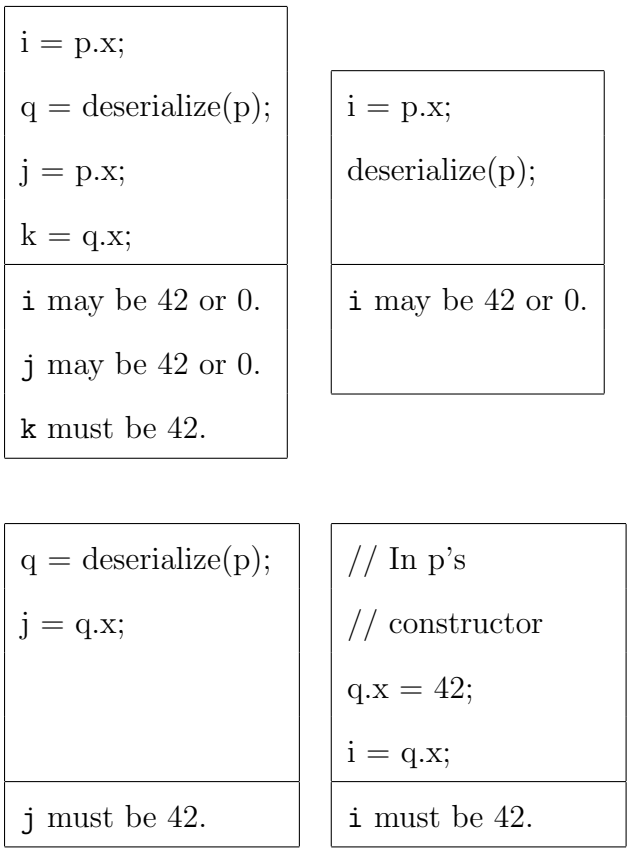
l will see 42 or throw a null pointer exception..

Figure 7.7: Example of Happens-Before Interaction

Thread 2 to see the writes to o that occurred prior to the freeze will therefore allow Thread 3 to see all of the writes prior to the freeze. There is therefore no reason not to provide Thread 3 with the same guarantees with which we provide Thread 2.

Semantics' Interaction with Happens-Before Edges

Now consider Figure 7.7. We want to describe the interaction between ordinary happens-before relationships and final field guarantees. In Thread 1, o is published incorrectly (before the freeze). However, if the code in Thread 1 happens-before the code in Thread 2, the normal happens-before relationships ensure that Thread 2 will see all of the correctly published values. As a result, j will be 42.



The `deserialize()` method sets the final field `p.x` to 42 and then performs a freeze on `p.x`. It passes back a reference to the `p` object. This is done in native code.

Figure 7.8: Four Examples of Final Field Optimization

What about the reads in Thread 3? We assume that `k` does not see a null value: should the normal guarantees for final fields be made? We can answer this by noting that the write to `Global.b` in Thread 2 is the same as a correct publication of `o`, as it is guaranteed to happen after the freeze. We therefore make the same guarantees for any read of `Global.b` that sees `o` as we do for a read of any other correct publication of `o`.

7.2.4 Reads and Writes of Final Fields in the Same Thread

Up to this point, we have only made guarantees about the contents of final fields for reads that have seen freezes of those final fields. This implies that a read of a final field in the same thread as the write, but before a freeze, might not see the correctly constructed value of that field.

Sometimes this behavior is acceptable, and sometimes it is not. We have four examples of how such reads could occur in Figure 7.8. In three of the examples, a final field is written via deserialization; in one, it is written in a constructor.

We wish to preserve the ability of compiler writers to optimize reads of final fields wherever possible. When the programs shown in Figure 7.8 access `p.x` before calling the `deserialize()` method, they may see the uninitialized value of `p.x`. However, because the compiler may wish to reorder reads of final fields around method calls, we allow reads of `p.x` to see either 0 or 42, the correctly written value.

On the other hand, we do want to maintain the programmer's ability to see the correctly constructed results of writes to final fields. We have a simple metric: if the reference through which you are accessing the final field was not used before the method that sets the final field, then you are guaranteed to see the last write to the final field. We call such a reference a *new* reference to the object.

This rule allows us to see the correctly constructed value for `q.x`. Because the reference `deserialize()` returns is a new reference to the same object, it provides the correct guarantees.

For cases where a final field is set once in the constructor, the rules are simple: the reads and writes of the final field in the constructing thread are ordered according to program order.

q.this$0 and `p.x` are final

Thread 1	Thread 2
`// in constructor for`	`r = Global.b;`
`// p`	`s = r.this$0;`
`q.this$0 = p;`	`t = s.x;`
`freeze q.this$0;`	
`p.x = 42;`	
`freeze p.x;`	
`Global.b = p;`	

`t` should be 42

Figure 7.9: Guarantees Should be made via the Enclosing Object

We must treat the cases (such as deserialization) where a final field can be modified after the constructor is completed a little differently.

7.2.5 Guarantees Made by Enclosing Objects

Consider Figure 7.9. In Java, objects can be logically nested inside each other – when an inner object is constructed, it is given a reference to the outer object, which is denoted in bytecode by `$0`. In Thread 1, the inner object `q` is constructed inside the constructor for `p`. This allows a reference to `p` to be written before the freeze of `p.x`. The reference is now tainted, according to the intuition we have built up so far: no other thread reading it will be guaranteed to see the correctly constructed values for `p.x`.

However, Thread 2 is guaranteed not to see the final fields of `p` until after `p`'s constructor completes, because it can only see them through the correctly published variable `Global.b`. Therefore, it is not unreasonable to allow this

Initially, a.ptr points to b, and b.ptr points to a. a.o, b.o and obj.x are all final.

Thread 1	Thread 2
b.o = obj;	r1 = A.ptr;
freeze b.o;	r2 = r1.o;
a.o = obj;	r3 = r2.x
freeze a.o;	
obj.x = 42;	
freeze obj.x;	s1 = B.ptr;
A = a;	s2 = s1.o;
B = b;	s3 = s2.x;

Figure 7.10: Cyclic Definition Causes Problems

thread to be guaranteed to see the correct value for p.x.

In general, we the semantics to reflect the notion that a freeze for an object o is seen by a thread reading a final field o.f if o is only read through a dereference chain starting at a reference that was written after the freeze of o.f.

7.3 Full Semantics

The semantics for final fields are as follows. A *freeze* action on a final field f of an object o takes place when a constructor for o in which f is written exits, either normally or abruptly (because of an exception).

Reflection and other special mechanisms (such as deserialization) can be used to change final fields after the constructor for the object completes. The set(...) method of the Field class in java.lang.reflect may be used to this effect. If the underlying field is final, this method throws an IllegalAccessException

unless `setAccessible(true)` has succeeded for this field and the field is non-static. If a final field is changed via such a special mechanism, a freeze of that field is considered to occur immediately after the modification.

Final Field Safe Contexts

An implementation may provide a way to execute a block of code in a *final field safe context*. Actions executed in a final field safe context are considered to occur in a separate thread for the purposes of Section 7.3, although not with respect to other aspects of the semantics. The actions performed within a final field safe context are immediately followed in program order by a synthetic action marking the end of the final field safe context.

Replacement and/or Supplemental Ordering Constraints

For each execution, the behavior of reads is influenced by two additional partial orders, dereference chain ($\overset{dc}{\rightarrow}$) and memory chain ($\overset{mc}{\rightarrow}$), which are considered to be part of the execution (and thus, fixed for any particular execution). These partial orders must satisfy the following constraints (which need not have a unique solution):

- **Dereference Chain** If an action a is a read or write of a field or element of an object o by a thread t that did not construct o, then there must exist some read r by thread t that sees the address of o such that $r \overset{dc}{\rightarrow} a$.
- **Memory Chain** There are several constraints on the memory chain ordering:
  a) If r is a read that sees a write w, then it must be the case that $w \overset{mc}{\rightarrow} r$.

b) If r and a are actions such that $r \overset{dc}{\rightarrow} a$, then it must be the case that $r \overset{mc}{\rightarrow} a$.

c) If w is a write of the address of an object o by a thread t that did not construct o, then there must exist some read r by thread t that sees the address of o such that $r \overset{mc}{\rightarrow} w$.

d) If r is a read of a final instance field of an object constructed within a final field safe context ending with the synthetic action a such that $a \overset{po}{\rightarrow} r$, then it must be the case that $a \overset{mc}{\rightarrow} r$.

With the addition of the semantics for final fields, we use a different set of ordering constraints for determining which writes occur before a read, for purposes of determining which writes can be seen by a read.

We start with normal happens-before orderings, except in cases where the read is a read of a final instance field and either the write occurs in a different thread from the read or the write occurs via a special mechanism such as reflection.

In addition, we use orderings derived from the use of final instance fields. Given a write w, a freeze f, an action a (that is not a read of a final field), a read r_1 of the final field frozen by f and a read r_2 such that $w \overset{hb}{\rightarrow} f \overset{hb}{\rightarrow} a \overset{mc}{\rightarrow} r_1 \overset{dc}{\rightarrow} r_2$, then when determining which values can be seen by r_2, we consider $w \overset{hb}{\rightarrow} r_2$ (but these orderings do not transitively close with other $\overset{hb}{\rightarrow}$ orderings). Note that the $\overset{dc}{\rightarrow}$ order is reflexive, and r_1 can be the same as r_2. Note that these constraints can arise regardless of whether r_2 is a read of a final or non-final field.

We use these orderings in the normal way to determine which writes can be seen by a read: a read r can see a write w if r is ordered before w, and there is no intervening write w' ordered after w but before r.

Static Final Fields

The rules for class initialization ensure that any thread that reads a *static* field will be synchronized with the static initialization of that class, which is the only place where static final fields can be set. Thus, no special rules in the JMM are needed for static final fields.

Static final fields may only be modified in the class initializer that defines them, with the exception of the `java.lang.System.in`, `java.lang.System.out`, and `java.lang.System.err` static fields, which can be modified respectively by the `java.lang.System.setIn`, `java.lang.System.setOut`, and `java.lang.System.setErr` methods.

7.4 Illustrative Test Cases and Behaviors of Final Fields

In order to determine if a read of a final field is guaranteed to see the initialized value of that field, you must determine that there is no way to construct a partial order $\stackrel{mc}{\rightarrow}$ without providing the chain $f \stackrel{hb}{\rightarrow} a \stackrel{mc}{\rightarrow} r_1$ from the freeze f of that field to the read r_1 of that field.

An example of where this can go wrong can be seen in Figure 7.11. An object o is constructed in Thread 1 and read by Threads 2 and 3. Dereference and memory chains for the read of `r4.f` in Thread 2 can pass through any reads by Thread 2 of a reference to o. On the chain that goes through the global variable `p`, there is no action that is ordered after the freeze operation. If this chain is used, the read of `r4.f` will not be correctly ordered with regards to the freeze operation. Therefore, `r5` is not guaranteed to see the correctly constructed value for the final field.

The fact that `r5` does not get this guarantee reflects legal transformations

`f` is a final field; its default value is 0

Thread 1	Thread 2	Thread 3
`r1.f = 42;`	`r2 = p;`	`r6 = q;`
`p = r1;`	`r3 = r2.f;`	`r7 = r6.f;`
`freeze r1.f;`	`r4 = q;`	
`q = r1;`	`if (r2 == r4)`	
	`r5 = r4.f;`	

We assume `r2`, `r4` and `r6` do not see the value null. `r3` and `r5` can be 0 or 42, and `r7` must be 42.

Figure 7.11: Final field example where reference to object is read twice

by the compiler. A compiler can analyze this code and determine that `r2.f` and `r4.f` are reads of the same final field. Since final fields are not supposed to change, it could replace `r5 = r4.f` with `r5 = r3` in Thread 2.

Formally, this is reflected by the dereference chain ordering (`r2 = p`) $\overset{dc}{\rightarrow}$ (`r5 = r4.f`), but *not* ordering (`r4 = q`) $\overset{dc}{\rightarrow}$ (`r5 = r4.f`). An alternate partial order, where the dereference chain does order (`r4 = q`) $\overset{dc}{\rightarrow}$ (`r5 = r4.f`) is also valid. However, in order to get a guarantee that a final field read will see the correct value, you must ensure the proper ordering for all possible dereference and memory chains.

In Thread 3, unlike Thread 2, all possible chains for the read of `r6.f` include the write to `q` in Thread 1. The read is therefore correctly ordered with respect to the freeze operation, and guaranteed to see the correct value.

In general, if a read R of a final field x in thread t_2 is correctly ordered with respect to a freeze F in thread t_1 via memory chains, dereference chains, and happens-before, then the read is guaranteed to see the value of x set before the

`a` is a final field of a class A

Thread 1	Thread 2
`r1 = new A;`	`r3 = p;`
`r2 = new int[1];`	`r4 = r3.a;`
`r1.a = r2;`	`r5 = r4[0]`
`r2[0] = 42`	
`freeze r1.a;`	
`p = r1;`	

Assuming Thread 2 read of `p` sees the write by Thread 1, Thread 2 reads of `r3.a` and `r4[0]` are guaranteed to see the writes to Thread 1.

Figure 7.12: Transitive guarantees from final fields

freeze F. Furthermore any reads of elements of objects that were only reached in thread t_2 by following a reference loaded from x are guaranteed to occur after all writes w such that $w \xrightarrow{hb} F$.

Figure 7.12 shows an example of the transitive guarantees provided by final fields. For this example, there is no dereference chain in Thread 2 that would permit the reads through `a` to be traced back to an incorrect publication of `p`. Since the final field `a` must be read correctly, the program is not only guaranteed to see the correct value for `a`, but also guaranteed to see the correct value for contents of the array.

Figure 7.13 shows two interesting characteristics of one example. First, a reference to an object with a final field is stored (by `r2.x = r1`) into the heap before the final field is frozen. Since the object referenced by `r2` isn't reachable until the store `p = r2`, which comes after the freeze, the object is correctly published, and guarantees for its final fields apply.

f is a final field; x is non-final

Thread 1	Thread 2	Thread 3
r1 = new ;	r3 = p;	r5 = q;
r2 = new ;	r4 = r3.x;	r6 = r5.f;
r2.x = r1;	q = r4;	
r1.f = 42;		
freeze r1.f;		
p = r2;		

Assuming that Thread 2 sees the writes by Thread 1, and Thread 3's read of q sees the write by Thread 2, r6 is guaranteed to see 42.

Figure 7.13: Yet Another Final Field Example

This example also shows the use of rule (c) for memory chains. The memory chain that guarantees that Thread 3 sees the correctly initialized value for f passes through Thread 2. In general, this allows for immutability to be guaranteed for an object regardless of which thread writes out the reference to that object.

7.5 Permissible Optimizations

The question for optimizing final fields is the same one we address when optimizing normal fields: what reorderings can a compiler writer prise out of these semantics? To be more precise, we must address two issues: first, what transformations can be performed on normal fields but not final fields? Next, what transformations can be performed on final fields but not on normal fields?

7.5.1 Prohibited Reorderings

The most important guarantee that we make for the use of final fields is that if an object is only made visible to other threads after its constructor ends, then those other threads will see the correctly initialized values for its final fields.

This can be easily derived from the semantics. Freezes occur at the end of constructors; the only way for another thread to read a final field of an object that has been properly constructed is by a combination of dereference and memory chains from the point at which that object was properly published (i.e., after the constructor ends). Thus, the requirement that the write in the chain $w \overset{hb}{\rightarrow} f \overset{hb}{\rightarrow} a \overset{mc}{\rightarrow} r_1 \overset{dc}{\rightarrow} r_2$ must be seen by the read is trivially true.

It is therefore of paramount importance that a write of a reference to an object where it might become visible to another thread never be reordered with respect to a write to a final field of the object pointed to by the reference. In addition, such a write should never be reordered with anything reachable from a final field that was written before the end of the constructor.

As an example of this, we look back at Figure 7.4. In that figure, the write to `pub` in Thread 1 must never be reordered with respect to anything that takes place before the read. If such a reordering occurred, then Thread 3 might be able to see the reference to the object without seeing the correctly initialized final fields.

7.5.2 Enabled Reorderings

There is one principles that guides whether a reordering is legal for final fields: the notion that "all references are created equal". If a thread reads a final field via multiple references to its containing object, it doesn't matter which one of

ready is a boolean volatile field, initialized to false.

a is an array.

Thread 1	Thread 2
a = {1,2,3};	i = pub.x;
ready = true;	j = i.f;
o.f = a;	if (ready) {
pub.x = o;	k = j[0];
freeze o.f;	}

Figure 7.14: Happens-Before Does Matter

those references is used to access the final field. None of those references will make more guarantees about the contents of that final field than any other. The upshot of this is that as soon as a thread sees a reference to an object, it may load all of that object's final fields, and reuse those values regardless of intervening control flow, data flow, or synchronization operations.

Consider once more the code in Figure 7.5. As soon as the read of pub.x occurs, all of the loads of o's final fields may occur; the reference pub.x of o is "equal" to the reference pub.y of o. This might cause uninitialized values to be seen for p.g and o.f, as the read of pub.x can occur before the freeze of p.g.

This should not be taken to mean that normal fields reachable through final fields can always be treated in the same way. Consider Figure 7.14. As a reminder, a happens-before ordering is enforced between a write to and a read of a volatile. In this figure, the volatile enforces a happens-before ordering between the write to the array and the read of j[0]: assuming that the other reads see the correct values (which is not guaranteed by the rest of the semantics), then k is required to have the value 1.

7.6 Implementation on Weak Memory Orders

One of the problems with guaranteeing where writes are seen without explicit lock and unlock actions to provide ordering is that it is not always immediately obvious how the implementation will work. One useful thing to do is consider how this approach might be implemented on a system where few guarantees are given about memory coherence. On such machines, it is often the case that the hardware will reorder your actions, spoiling some of the guarantees we want to give to final fields.

In this section, we discuss sample implementation strategies that allow implementation of final fields on symmetric multiprocessor systems under various architectures and under lazy release consistent distributed shared memory systems.

7.6.1 Weak Processor Architectures

To implement the semantics as we have described them, a processor must make guarantees for both reading and writing threads. For the writer thread, a processor must not allow stores to final fields to be seen to be reordered with later stores of the object that contains the final field. On many architectures, the processor must be explicitly prevented from doing this by some sort of explicit memory barrier instruction, which can be performed after the constructor finishes.

Many processor architectures, including SPARC TSO [WG94] and Intel x86, do not require this memory barrier – there is an implicit barrier between stores in the program. Other, more relaxed architectures, such as Intel's IA-64 [Int02] and Alpha [Com98] architectures, do require an explicit memory barrier.

On the reader side, there needs to be an ordering between a dereference of

an object, and a dereference of final fields of that object. Most architectures do not require such a memory barrier; generally, ordering is preserved between dependent loads.

The notable exception to this is the Alpha architecture. On the Alpha, no such ordering is provided. One simple strategy would be to insert a memory barrier after each load of an object that has a final field. This, however, is an extremely heavyweight strategy, and is not recommended.

Before we proceed with alternate strategies, it is worth mentioning that there are no plans for future versions of Java to be implemented on Alpha processors. The Intel IA-64 platform was originally an exception to this as well, but discussions between Intel and the JSR group overseeing the new Java memory model caused a rewrite in the appropriate section of the IA-64 memory model.

An alternative is to maintain an invariant that if a thread or stack has a reference to a heap allocated object, then the thread has done the memory barriers required to see the constructed version of that object. The simple implementation of this is to put a memory barrier after each operation that loads a reference (getfield, getstatic or aaload). Since we only need to put a barrier after getfields that load a reference, as opposed to putting one before all getfields, this is somewhat better than the simple strategy.

A further optimization is possible if we note that it is not necessary to perform a memory barrier after a getfield of a final field. The reasoning here is that if a thread is up to date with respect to an object X, and then loads a final field of X that references Y, then Y must be older than X and the thread must therefore be up to date with respect to Y.

Finally, if the reference we load is null, we can skip the memory barrier (this

Memory Barriers At Which Action?	Number of Required Memory Barriers
Without Making Fields Final	
getfields	202,869,185
loads of refs	64,993,844
nonfinal loads of refs	64,809,454
nonfinal nonnull loads of refs	50,011,951
After Making Fields Final	
nonfinal loads of refs	22,628,420
nonfinal nonnull loads of refs	7,832,852

Figure 7.15: Number of Required Memory Barriers on Weak Memory Orders

would involve a runtime test to see if the ref is null, and doing the memory barrier only if it is not).

To test the overhead for both of these solutions, we instrumented Sun's JDK 1.2.2 JVM (a relatively early VM with an easily instrumented architecture). Final fields are rarely used in standard benchmarks; for some tests we adjusted the SPECjvm98 benchmarks so that all fields that are never modified were made into final fields. The total number of required memory barriers over all benchmarks can be seen in Figure 7.15.

More implementation strategies for final fields on the Alpha, that decrease the number of required memory barriers further, are discussed in [Man01].

7.6.2 Distributed Shared Memory Based Systems

Imagine a Lazy Release Consistent (LRC) machine (see [KCZ92]): a processor acquires data when a lock action occurs, and releases it when an unlock action occurs. The data "piggyback" on the lock acquire and release messages in the

form of "diffs", a listing of the differences made to a given page since the last acquire of that memory location.

Let us assume that each object with a final field is allocated in space that had previously been free. The only way for a second processor to see a pointer to that object at all is to perform an acquire after the processor constructing the object performed a release. If the release and the acquire do not happen, the second processor will never see a pointer to that object: in this case, neither the object's final fields nor anything reachable in a dereference chain from its final fields will appear to be incorrectly initialized.

Let us now assume that the acquire and release do happen. As long as these actions take place after object has been constructed (and there is no code motion around the end of the constructor), the diffs that the second processor acquires are guaranteed to reflect the correctly constructed object. This property makes implementation of final fields on a LRC-based DSM possible.

Chapter 8

Related Work

A scientist will never show any kindness for a theory which he did not start himself.

- Mark Twain, A Tramp Abroad

8.1 Architectural Memory Models

Most work on memory models has been done because of the need to verify properties of hardware architectures. An excellent primer for this work is available [AG96]. An early discussion of memory models can be found in [Lam78], which provides the widely used definition for sequential consistency.

The needs of memory models for hardware architectures differ from the needs of programming language memory models. The most obvious difference is the lack of a need for language specific features, such as immutability support, type safety, class initialization, and finalization. The design of architecture-based models must focus on issues like when writes get sent to a cache or to main memory, whether instructions are allowed to issue out of their original order, whether reads must block waiting for their result, and so on. There is very little discussion of the interaction of compiler technology and processor architectures in much of

the literature. Perhaps the greatest difference between the research that has been done for architectures and our research is the lack of a full treatment for causality (as in Section 4.3).

Architecture memory models are generally classified along a continuum from strong (which does not allow the results of instructions to be viewed out of order) to weak (which allows for much more speculation and reordering). Our work has to provide a "catch-all" for these memory models; Java should run efficiently on as many machines as possible. Our memory model can therefore be categorized as "relaxed".

8.2 Processor Memory Models

Each memory model provides synchronization operations that allow a strengthening of the model for that operation. On processors, there is customarily processor support for a "memory barrier" (membar), an operation that makes guarantees about whether instructions are finished. In addition, there may also be completion flags to the memory barriers, which determine what it means for an instruction to be finished. Finally, individual load and store operations for a processor may have special memory semantics. When these operations are needed depend on the strength of the memory model. Here, we examine the requirements of several different processors.

SPARC

The SPARC processor has three different memory models: Total Store Order (TSO), Partial Store Order (PSO) and Relaxed Memory Order (RMO) [WG94]. Sun's Solaris operating system is implemented in TSO; this mode allows the

Initially, a = b = 0;

Thread 1	Thread 2	Thread 3
a = 1;	r1 = a;	r2 = b;
	if (r1 == 1)	if (r2 == 1)
	b = 1;	r3 = a;

Figure 8.1: Can r2 = 1, r3 = 0?

system to execute a write followed by a read out of program order, but forbids other reorderings.

One interesting additional relaxation that TSO allows is illustrated by Figure 8.2 (as seen in [AG96]). If all three threads are on the same processor, the value for a written in Thread 1 can be seen early by Thread 2, but not by Thread 3. In the mean time, the write to b can occur, and be seen by Thread 3. The result of this would be $b == 1$, $r1 == 0$. TSO allows this relaxation, which is sometimes called *write atomicity relaxation.*

An example of the way in which a SPARC membar can affect memory is the modifier #StoreLoad, which ensures that all stores before the memory barrier complete before any loads after it start. There are also #LoadStore, #StoreStore and #LoadLoad modifiers, which have the obvious related implications.

Alpha

The Alpha makes very few guarantees about the ordering of instructions. A read may be reordered with a write, and a write may be reordered with another write, as long as they are not to the same memory location. The Alpha does enforce ordering between two reads of the same memory location, however [Com98]. Fi-

nally, the Alpha allows for write atomicity relaxation if all threads are on a single processor.

On the Alpha 21264, the MB instruction ensures that no other processor will see a memory operation after it before they see all of the memory operations before it. The strength of this operation is important because of the weak nature of the Alpha memory model. The Alpha also has a WMB (Write Memory Barrier), which prevents two writes from being reordered.

IA-64

The IA-64 memory model is similar to that of the Alpha. The IA-64 memory fence instruction (mf) takes two forms. First, the mf instruction (ordering_form) ensures that "all data memory accesses are made visible prior to any subsequent memory accesses". [Int99].

The IA-64 allows one relaxation that the Alpha does not allow: writes can be seen early by other processors, not just the one that performed the write. This relaxation may be a little obscure, but it is simply another reflection of Figure 8.2. We have already mentioned that the result $b = 1, r1 = 0$ can happen on most architectures, including Alpha, if the threads are all on the same processor. IA-64, unlike Alpha, allows this result to be seen if the threads are all running on different processors.

Other

There are a number of other memory orders, of both the purely theoretical and the implemented variety. The PowerPC model, for example, is substantially similar to the IA-64 model, including the write atomicity relaxation for multiple

processors.

Release Consistency

Distributed shared memory systems, because of the high cost of communication between the processors, tend to have very weak memory models. A common model, called *release consistency* [GLL+90], allows a processor to defer making its writes available to other threads/processors until it performs a release. This allows arbitrary reordering of code, but must respect all synchronization actions; all acquires and releases affect global memory.

A relaxation of release consistency, called *lazy release consistency* [KCZ92], is similar to our model. Not only does lazy release consistency allow a processor to defer making its writes available to other processors until it performs a release operation, but, in fact, the publication of the writes can be deferred until the other processor performs an acquire operation on the same lock that was released by the first processor. Because release consistency requires that all writes be released immediately, it is sometimes called *eager* release consistency.

Location consistency [GS00a] is also similar to our model. Each memory location is said to have a partial order of writes and synchronization actions associated with it. If a processor sees an acquire for a memory location l, it can read the last write made to l before the last release on l, or any write made to l after that point. This is similar to lazy release consistency; the main difference is that acquires and releases only affect a single memory location.

8.3 Programming Language Memory Models

Our work focuses on memory models for programming languages, whose needs differ significantly from those of hardware memory models. Memory models for programming languages need the ability to deal with programming language level constructs, such as final fields and volatile variables. In addition, unlike processor models, programming language models need to be applicable to a wide variety of architectures: any architecture which runs programs written in that language needs to be able to support the memory model.

8.3.1 Single-Threaded Languages

Programming languages that are either single-threaded or only support multiple threads through application libraries do not require their own memory models. C [KR88] and C++ [Str97] are prime examples of this: they use libraries such as POSIX threads (pthreads) [LB98] to support multi-threading. If a system uses pthreads, there is no guarantee of portability across all platforms that support that language.

In a single-threaded language, there is no concept equivalent to the one for which we use volatile fields. The sole purpose of a volatile field in our semantics is to enable efficient interthread communication; in an environment where there is no interthread communication, there is no meaningful use for volatile.

C and C++ do, of course, have a volatile modifier. It is traditionally used for preventing code reordering when dealing with hardware-dependent issues like memory mapped I/O. According to the C++ standard [Str97, §r7.1.6], there are "no implementation-independent semantics for volatile objects".

Similarly, it is not necessary for a single-threaded language to provide multi-

threaded semantics for immutable fields. The `const` qualifier in C and C++ enforces single-threaded immutability, but there is no way for a `const` field to imply that any object or structure reachable through that field is visible to another thread.

8.3.2 Multi-Threaded Languages

Multi-threaded languages generally fall into one of two categories, *message passing* and *shared memory*. Each has its own specific memory model needs.

Message Passing Languages

In message passing languages, data are shared between threads by explicit communication. If one thread (a *sender*) wants to communicate with another (a *receiver*), it sends a message to it. The receiver then accepts the message.

Because all communication between threads in message passing languages is explicit, they have no need for a detailed memory model. If a thread receives the data from the sender, it can be safely assumed that communication has taken place.

One example of a message passing language is Occam [Jon87], which uses a message passing framework similar to that of CSP [Hoa85]. Ada [Ada95] allows for message passing between threads using a *rendezvous*, but it also allows shared memory parallelism using locking and fork/join parallelism (see below).

Shared Memory Languages

Because they can rely on both implicit and explicit communication, memory models for shared memory languages are more complicated. Nearly every language

with built-in multithreading has made some attempt to define what communication between threads entails. Many multi-threaded languages contain some sort of lock construct; when a thread a acquires a lock released by a thread b, the memory values visible to a at the time it released the lock must be visible to b. Languages with built in locking include Java [GJS96], Ada [Ada95], Cilk [Gro00] and most of the other multithreaded languages.

The language Cilk employs a form of *fork/join parallelism*. A parent thread can spawn (or fork) child processes. If the parent wishes, it can execute a `sync` statement (or a join) - the parent will then wait for all of the spawned threads to complete and continue. The forked and joined threads form a DAG; writes are visible from a parent in the DAG to its children. This provides an easy to understand memory model which is sometimes referred to as DAG-consistency [BFJ+96]

Hiding the Memory Model

A certain amount of work has been done to try to hide the fact that the memory model for programming languages is not sequential consistency. A trivial way of performing this on many architectures is to insert memory and compiler barriers between every memory access; not only is this obviously tremendously expensive in and of itself, but it also negates many of the design features that provide for efficient execution.

Most programming language implementations make the guarantee that correctly synchronized programs have sequentially consistent semantics. This means that non-sequentially consistent results are only visible when data races are present. In fact, Shasha and Snir [SS88] show that non-sequentially consistent

results can only occur when there is a cycle in the graph of happens-before and conflict edges (if two accesses conflict, there is a conflict edge between them). Therefore, detecting where to place memory barriers is a matter of detecting these cycles, and placing memory barriers accordingly.

Midkiff, Padua and Cytron [MPC90] extended this work to apply to arrays. Shasha and Snir's original analysis did not take locking and other explicit synchronization into account; this was addressed by the compiler analysis of Krishnamurthy and Yelick [KY95]. Lee and Padua [LP01] developed a similar compiler analysis, and showed that minimizing the number of memory barriers using their technique was NP-complete.

The work in this area places constraints on compilers, and does not apply to correctly written code. Our work takes the position that there are few, if any, correct programming patterns for high-level languages that involve data races. We therefore provide the minimum number of guarantees for such code to ensure basic safety properties. We do not constrain compilers or architectures to sacrifice performance or complexity for programs that are likely incorrect (usually because of a misunderstanding of the semantics of multithreaded code).

Semantics of Multithreaded Java

Java's original memory model [GJS96, §17] is extremely difficult to understand; there have been several attempts to formalize it [CKRW97, CKRW98, GS97]. Unfortunately, due to its complexity, the resulting formalisms were contradictory and difficult to understand, at best. Gontmakher and Schuster [GS97, GS98] believed that simple reordering of independent statements was illegal. However, they did not consider prescient stores (as described in the original JLS [GJS96]).

They later presented a revised model [GS00b].

The same authors [GS97, GS00b] proved that the original Java memory model required a property called *coherence* (as originally described in [ABJ+93]). This innocent sounding property requires that the order in which program actions occur in memory is seen to be the same on each thread. Specifically, coherence disallows a common case for reordering reads when those reads can see writes that occur in another thread via a data race [Pug99].

The language specification was, therefore, obviously not complete. A language's memory model must take into account that fact that both processors and compilers can perform optimizations that may change the nature of the code, and that if consistent behavior is desired, a specification must account for what may happen when synchronization is not used to emulate sequential consistency.

A number of proposals for replacement memory models for Java have emerged since the deficiencies of the original model became apparent. Most of these are based around modeling techniques originally used for architecture memory models; most are also operational.

Commit / Reconcile / Fence Approach Maessen, Arvind and Shen [AMS00] used the Commit / Reconcile / Fence protocol [SAR99] to propose a memory model for Java. That proposal has several major weaknesses. First, it does not distinguish between writes to final fields and writes to non-final fields; final field semantics are guaranteed through the use of a memory barrier at the end of a constructor. This means that writes to any fields of an object a in a's constructor must be guaranteed to be seen by other threads if they access a via a reference published after that memory barrier. This is an onerous burden for architectures with weak memory models and disallows optimizations that might be performed

Initially: a = 0

Thread 1	Thread 2
a = 1;	a = 2;
i = a;	j = a;

Figure 8.2: CRF does not allow i == 2 and j == 1

on objects with no final fields.

Additionally, problems arise in their approach because all communication is performed through a single, global memory. For example, in Figure 8.3.2 the result i = 2 and j = 1 is prohibited. This is because CRF requires an ordering over writes to global memory; one thread cannot perceive that a = 1 happens before a = 2 while the other thread perceives that a = 2 happens before a = 1.

Finally, their model does not allow for as much elimination of "useless" synchronization operations. The CRF-based specification provides a special rule to skip the communication actions associated with a lock action if that thread is the last one that released the lock. However, there is no symmetric rule for unlock actions or volatile variables, so the communication associated with them can never be eliminated. Finally, even though the inter-thread communications may be skipped for lock actions, it is unclear that instructions can be moved around thread-local locks.

Operational Semantics The work that led to development of the model in this dissertation [MP01b, MP01a] originally used an operational semantics to describe a memory model for Java. The semantics in those papers describes a machine that can take as input a Java program, and produce as output any legal execution of that program.

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

Thread 1	Thread 2	Thread 3
`z = 1;`	`r1 = y`	`r3 = x;`
	`r2 = z;`	`y = 1;`
	`x = r2`	

We must allow `r1 == r2 == r3 == 1`

Figure 8.3: A Purely Operational Approach Does Not Work

One of the major problems with that model is the operational approach it takes. The final Java memory model allows a write to occur early if an execution in which that write occurs can be demonstrated. The original, operational approach we described allows a write to be performed early if three conditions were fulfilled:

1. The write would occur (to the same variable, with the same value),
2. The write would not be seen by the thread that performed it before the point at which it occurred originally, and
3. The write would not appear to occur out of thin air.

A major problem with this approach is illustrated by the example in Figure 8.3. In order to allow the reads of `y` and `x` to return the value 1, they need to be able to be reordered with the writes to `x` and `y`, respectively. This is similar to the sorts of reorderings we have seen.

The approach allows the writes to occur early, but *only if the write is guaranteed to occur to write out the same value.* In this case, the write is **not** guaranteed to occur with the same value: the read of `z` may return either 0 or 1, so either

value may be written. Thus, the approach in that paper does not allow this result.

Unified Memory Model approach Yang, Gopalakrishnan and Lindstrom [YGL01, YGL02] attempt to characterize the approaches taken in [MP01b, MP01a, AMS00] using a single, unified memory model (called UMM). They [YGL02, Yan04] also use this notation to propose a third memory model.

The UMM based approach does not attempt to solve some of the issues that a memory model for Java is required to solve. The paper suggests that it is more important to formulate a simple memory model than one that fulfills the informal requirements laid out by the Java community. First, they do not allow the result in Figures 3.3 and 8.4; an action in their model is only allowed to be reordered around a control or data dependence if the action is guaranteed to occur in all executions. This disallows, for example, redundant load elimination, an extremely important compiler optimization.

Their treatment of final fields is equally lacking. First, they do not allow for transitive visibility effects for finals. For the example of an immutable String object with an integer `length` field and a reference to an array of characters `contents`, the integer would be correct, the reference would point to the correct array, but the array's contents would not be guaranteed to be up to date. This simplifies the model, but does not satisfy safety conditions.

Second, their rules for final have the effect that once the field has been frozen (at the end of a constructor), all other threads are guaranteed to see the final value rather than the default value. It is difficult to see how this could be implemented efficiently; this has therefore never been part of the agreed-upon safety guarantees for final fields. They do not address this issue.

Initially, `x == y == 0`

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`r2 = r1 | 1;`	`x = r3;`
`y = r2;`	

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

Figure 8.4: Compilers Can Think Hard About When Actions Are Guaranteed to Occur

Other Kotrajaras [Kot01] 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. It also fails to describe adequately what happens in the case of a cycle in control dependency. Furthermore, their proposed model violates the informal rules for final fields by disallowing references to the object being constructed from escaping their constructors.

Saraswat [Sar04] 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 3.3; 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 8.4; in that example, given the binding of reads to writes, there are multiple fixed-point solutions.

8.4 Final Fields

The C# language [ECM02a] has a `readonly` keyword that is similar to Java's `final` modifier. Fields marked as `readonly` do not have any additional threading semantics in C#'s runtime environment, the Common Language Infrastructure [ECM02b]; they must, as in earlier revisions of Java, be treated as normal fields for the purposes of synchronization.

Chapter 9

Future Work

Answers are easy. It's asking the right questions which is hard.

- Chris Boucher, The Face of Evil

This work is the first that has explored programming language memory models in depth. While it has resolved the issue of what the Java programming language's memory model is to be, there are many potential areas for future work. This chapter explores some of these areas.

9.1 Performance Testing and Optimization

We have spent a lot of time in this dissertation detailing the results a Java program may have. We have also pointed out the fact that as long as one of those results is obtained, any implementation is allowed. For example, we have seen how the Java memory model allows a wide variety of optimizations disallowed by the previous memory model.

Although we can never enumerate all possible implementations (there are an infinite number of choices that can be made), we do have a clearly defined notion of what is **not** allowed in an implementation. We have strengthened both volatile and final fields – having done so, it is important to measure the performance

impact of that strengthening.

This is a more difficult task than it may appear to be at first blush. There are several questions that need to be answered. First is the question of what we use for comparison. The answer to this question probably must be an alternative model that does not strengthen volatile or final fields. This may come in the form of, say, an earlier implementation of Java, but older implementations do not include as many performance optimizations as newer ones, so direct comparisons are difficult.

More important than the basis for comparison is the problem of choosing benchmarks. In order to measure the impact of volatiles and finals effectively, you have to choose a benchmark that will be correct under both the old and new implementations. This has different implications for final and volatile fields.

For final fields, the impact is going to be that of a memory / compiler barrier at the end of a constructor of an object with at least one final field. We have performed some analysis of the impact of these barriers, as discussed in Chapter 7. In general, however, non-static final fields are not all that common in widely used benchmarks; our benchmarks made fields final that were not final previously. Furthermore, barriers are only needed at the end of a constructor when a reference to the object is passed to another thread via a data race. This is an uncommon case; more future work may eliminate a substantial number of barriers.

It is more difficult to evaluate the impact of volatile fields. All of the known algorithms that employ volatile fields require the semantics that we gave them. In developing the memory model, we found that programmers who were using volatile fields were invariably either tacitly assuming our semantics, or had code that wouldn't have done what they wanted it to do. As a result of this, existing

benchmarks that employ volatile require our semantics for correctness. Thus, it is useless to run those benchmarks on virtual machines that do not implement volatile correctly.

Future work that evaluates the performance impact of the new memory model must also take into account the new ways of improving performance that are allowed. That means exploiting newly available techniques to their fullest. This includes, but is not limited to:

- Newly permissible optimizations on monitors, such as synchronization elimination and lock coarsening.
- Newly available programming idioms, including lock- and wait-free algorithms that use volatile.
- Optimizations that reduce the cost of final and volatile variables, such as escape analysis to determine objects that do not need barriers at the end of constructors, and elimination of barriers from accesses to volatile fields of thread-local objects.

9.2 Better Programming Models

The need for backwards compatibility had a tremendous impact on the design of the Java memory model. Examples of this abound. Consider the semantics of final fields; they are substantially more complex simply because of the possibility that final fields can be changed by deserialization. Another example is the presence of potential data races in the language; this brings with it the need for causality semantics.

Java programmers should not have to deal with the full complexity introduced by the model. There are two ways of mitigating this complexity:

- Changes can be made to Java to reduce the difficulty of dealing with concurrency, or
- New programming languages and paradigms can be introduced with less complexity.

These two approaches are intertwined; features introduced in a new language (like lightweight transactions [HF03], for example) may be introduced in an older language by way of new APIs and system calls. Actual language changes in a language like Java must be made very carefully, because of backwards compatibility issues. New APIs are frequently used to introduce significant functionality (as per Isolates or Real-Time extensions [Jav02, Jav03]).

However, deciding what features to develop is a difficult task. For example, many attempts have been made to design a language that is completely free from data races [BST00, FF00]. However, because they eliminate data races by enforcing mutual exclusion, such languages only allow limited support for wait- and lock-free data structures.

Some language designers are attempting to introduce the notion of atomicity (in the form of lightweight transactions, for example) – however, such languages are still prototypical, and it is still unclear whether they can be implemented effectively and efficiently.

9.3 Other Programming Languages

Many of the issues for other languages were described in Chapter 8. There is some potential work to do for those languages that does not reflect existing work.

As we have mentioned, most languages do not incorporate a well-designed memory model. Those that do tend to have limited options for concurrency control. It is quite easy to define semantics for languages where, for example, unsynchronized access to shared variables is impossible.

On the other hand, these issues may be more difficult for popular languages like C and C++. C and C++, for example, are single threaded languages; this implies that there is no consistent specification that prohibits reordering synchronization operations with non-synchronization operations. The volatile modifier, for example, is defined to be "implementation dependent". This also implies that the semantics change depending on the threading library used and on the compiler / platform combination.

This leads to some serious potential anomalies when writing multithreaded code on these platforms. Consider the following C code:

```
static short sharedArr [4];

void t1() {
  sharedArr[1] = 1;
  sharedArr[3] = 3;
}

void t2() {
```

```
  sharedArr[2] = 2;
  sharedArr[4] = 4;
}
```

Consider two threads, each acquiring and releasing a different lock, and executing `t1` and `t2` concurrently. One thread may read the array as a single, 32 bit value, update it in a register and block. The other thread may read the array as a single, 32 bit value, update it in a register and write out its changes. The first thread might then continue, and write out the value of the array stored in its register. This would cause the update in `t2` to be lost. Java disallows this, but C does not – in fact, the pthread specification allows it.

In general, many of the most important questions for C and C++ are the same kind of questions we answered for Java. For example, should the `volatile` modifier have the same semantics? It is possible to make it stronger – for example, perhaps all writes that occur to any volatile should be visible after a read of any other subsequent volatile. Decisions on this would affect the semantics of all synchronization mechanisms, including the use of locking and explicit memory barriers.

Another example is that of word tearing. In Java, reads and writes of 32-bit values are always atomic, while 64-bit values may be written or read in 32-bit chunks. C and C++ may require fewer guarantees. For example, if a 32-bit value is aligned improperly, it may be more difficult to provide atomicity guarantees.

Chapter 10

Conclusion

In this dissertation, 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:

- It provides a clear and simple semantics that explains how threads can use locking to interact through memory, providing crucial ordering and visibility properties.
- 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.
- It refines the concept of `volatile` variables, which can be used in place of explicit memory barriers to design lock- and wait-free algorithms, while still providing the aforementioned crucial ordering and visibility properties.

- It strengthens the concept of `final` variables, which can now be used to provide thread-safe immutability regardless of the presence of data races.

This dissertation (and the model contained within) clarified and formalized these needs, balancing them carefully with a wide variety of optimizations and program transformations commonly performed by compilers and processor architectures. It also provided verification techniques to ensure that the model reflects this balancing act accurately and carefully.

It is these two things in concert: both the balance, and the degree to which the model has been verified (both by peer review and proof techniques), that has allowed this model to be used for practical applications. The designers of the Itanium memory model, for example, changed their specification to account for the need for final fields to appear immutable without additional memory barriers. The architects of C and C++ are now looking into adapting some of the work that has been done for this model to their languages. And finally, of course, this model has been adopted as the foundation for concurrent programming in the Java programming language.

Appendix A

Finalization

"Dying is a very dull, dreary affair and my advice to you is to have nothing whatever to do with it."

– W. Somerset Maugham

In the Java programming language, the `Object` class has a protected method called `finalize`; this method can be overridden by other classes. The particular definition of `finalize` that can be invoked for an object is called the *finalizer* of that object. Before the storage for an object is reclaimed by the garbage collector, the Java virtual machine will invoke the finalizer of that object.

Finalizers provide a chance to free up resources that cannot be freed automatically by an automatic storage manager. In such situations, simply reclaiming the memory used by an object would not guarantee that the resources it held would be reclaimed.

The JLS does not specify how soon a finalizer will be invoked, except to say that it will occur before the storage for the object is reused. Also, the language does not specify which thread will invoke the finalizer for any given object. It is guaranteed, however, that the thread that invokes the finalizer will not be holding any user-visible synchronization locks when the finalizer is invoked. If

an uncaught exception is thrown during the finalization, the exception is ignored and finalization of that object terminates.

It should also be noted that the completion of an object's constructor (including all of its freezes for final fields) happens-before the execution of its `finalize` method (in the formal sense of happens-before).

Finalizers interact in a special way with final field semantics. Each finalizer takes place in a final field safe context. Additionally, as described in Section 5.2, each finalizer begins with a logical read of a reference to the object being finalized. The memory chain for that final field (see Section 7.1) passes through that read, providing immutability guarantees to the finalizer.

It is important to note that many finalizer threads may be active (this is sometimes needed on large SMPs), and that if a large connected data structure becomes garbage, all of the finalize methods for every object in that data structure could be invoked at the same time, each finalizer invocation running in a different thread.

The `finalize` method declared in class Object takes no action.

The fact that class `Object` declares a `finalize` method means that the `finalize` method for any class can always invoke the `finalize` method for its superclass. This should always be done, unless it is the programmer's intent to nullify the actions of the finalizer in the superclass. Unlike constructors, finalizers do not automatically invoke the finalizer for the superclass; such an invocation must be coded explicitly.)

For efficiency, an implementation may keep track of classes that do not override the `finalize` method of class Object, or override it in a trivial way, such as:

```
protected void finalize() throws Throwable {
   super.finalize();
}
```

We encourage implementations to treat such objects as having a finalizer that is not overridden, and to finalize them more efficiently, as described in Section A.1.

A finalizer may be invoked explicitly, just like any other method.

The package `java.lang.ref` describes weak references, which interact with garbage collection and finalization. As with any API that has special interactions with the language, implementors must be cognizant of any requirements imposed by the java.lang.ref API. This specification does not discuss weak references in any way. Readers are referred to the API documentation for details.

A.1 Implementing Finalization

Every object has two attributes: it may be *reachable*, *finalizer-reachable*, or *unreachable*, and it may also be *unfinalized*, *finalizable*, or *finalized*.

A *reachable* object is any object that can be accessed in any potential continuing computation from any live thread. Any object that may be referenced from a field or array element of a reachable object is reachable. Finally, if a reference to an object is passed to a JNI method, then the object must be considered reachable until that method completes.

Optimizing transformations of a program can be designed that reduce the number of objects that are reachable to be less than those which would naïvely be considered reachable. For example, a compiler or code generator may choose

to set a variable or parameter that will no longer be used to null to cause the storage for such an object to be potentially reclaimable sooner.

Another example of this occurs if the values in an object's fields are stored in registers. The program then may access the registers instead of the object, and never access the object again. This would imply that the object is garbage.

Note that this sort of optimization is only allowed if references are on the stack, not stored in the heap. For example, consider the *Finalizer Guardian* pattern [Blo01]:

```
class Foo {
  private final Object finalizerGuardian = new Object() {
    protected void finalize() throws Throwable {
      /* finalize outer Foo object */
    }
  }
}
```

The finalizer guardian forces a `super.finalize` to be called if a subclass overrides finalize and does not explicitly call `super.finalize`.

If these optimizations are allowed for references that are stored on the heap, then the compiler can detect that the *finalizerGuardian* field is never read, null it out, collect the object immediately, and call the finalizer early. This runs counter to the intent: the programmer probably wanted to call the `Foo` finalizer when the `Foo` instance became unreachable. This sort of transformation is therefore not legal: the inner class object should be reachable for as long as the outer class object is reachable.

Transformations of this sort may result in invocations of the `finalize` method occurring earlier than might be otherwise expected. In order to allow the user to prevent this, we enforce the notion that synchronization may keep the object alive. *If an object's finalizer can result in synchronization on that object, then that object must be alive and considered reachable whenever a lock is held on it.*

Note that this does not prevent synchronization elimination: synchronization only keeps an object alive if a finalizer might synchronize on it. Since the finalizer occurs in another thread, in many cases the synchronization could not be removed anyway.

A *finalizer-reachable* object can be reached from a finalizable object through some chain of references, but not from any live thread. An *unreachable* object cannot be reached by either means.

An *unfinalized* object has never had its finalizer automatically invoked; a *finalized* object has had its finalizer automatically invoked. A *finalizable* object has never had its finalizer automatically invoked, but the Java virtual machine may eventually automatically invoke its finalizer. An object cannot be considered finalizable until its constructor has finished. Every pre-finalization write to a field of an object must be visible to the finalization of that object. Furthermore, none of the pre-finalization reads of fields of that object may see writes that occur after finalization of that object is initiated.

A.2 Interaction with the Memory Model

It must be possible for the memory model to decide when it can commit actions that take place in a finalizer. This section describes the interaction of finalization with the memory model.

Each execution has a number of *reachability decision points*, labeled d_i. Each action either *comes-before* d_i or *comes-after* d_i. Other than as explicitly mentioned, *comes before* in this section is unrelated to all other orderings in the memory model.

If r is a read that sees a write w and r comes-before d_i, then w must come-before d_i. If x and y are synchronization actions on the same variable or monitor such that $x \xrightarrow{so} y$ and y comes-before d_i, then x must come-before d_i.

At each reachability decision point, some set of objects are marked as unreachable, and some subset of those objects are marked as finalizable. These reachability decision points are also the points at which References are checked, enqueued and cleared according to the rules provided in the specifications for `java.lang.ref`.

Reachability

The only objects that are considered definitely reachable at a point d_i are those that can be shown to be reachable by the application of these rules:

- An object B is definitely reachable at d_i from static fields if there exists there is a write w_1 to a static field v of a class C such that the value written by w_1 is a reference to B, the class C is loaded by a reachable classloader and there does not exist a write w_2 to v s.t. $\neg(w_2 \xrightarrow{hb} w_1)$, and both w_1 and w_2 come-before d_i.

- An object B is definitely reachable from A at d_i if there is a write w_1 to an element v of A such that the value written by w_1 is a reference to B and there does not exist a write w_2 to v s.t. $\neg(w_2 \xrightarrow{hb} w_1)$, and both w_1 and w_2 come-before d_i.

- If an object C is definitely reachable from an object B, object B is definitely reachable from an object A, then C is definitely reachable from A.

An action a is an active use of X if and only if

- it reads or writes an element of X
- it locks or unlocks X and there is a lock action on X that happens-after the invocation of the finalizer for X.
- it writes a reference to X
- it is an active use of an object Y, and X is definitely reachable from Y

If an object X is marked as unreachable at d_i,

- X must not be definitely reachable at d_i from static fields,
- All active uses of X in a thread t that come-after d_i must occur in the finalizer invocation for X or as a result of thread t performing a read that comes-after d_i of a reference to X.
- All reads that come-after d_i that see a reference to X must see writes to elements of objects that were unreachable at d_i, or see writes that came after d_i.

If an object X marked as finalizable at d_i, then

- X must be marked as unreachable at d_i,
- d_i must be the only place where X is marked as finalizable,
- actions that happen-after the finalizer invocation must come-after d_i

BIBLIOGRAPHY

[ABJ+93] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The Power of Processor Consistency. In *The Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1993.

[Ada95] Ada Joint Program Office. *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, Massachusetts, 1995.

[Adv93] Sarita Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin, Madison, December 1993.

[AG96] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AH90] Sarita Adve and Mark Hill. Weak ordering—A new definition. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 2–14, 1990.

[AMS00] Arvind, Jan-Willem Maessen, and Xiaowei Shen. Improving the Java memory model using CRF. In *Object Oriented Programming Systems, Languages and Applications*, pages 1–12, October 2000.

[BFJ+96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. DAG-consistent distributed shared memory. In *10th International Parallel Processing Symposium (IPPS '96)*, pages 132–141, Honolulu, Hawaii, April 1996.

[Blo01] Joshua Bloch. *Effective Java programming language guide*. Sun Microsystems, Inc., 2001.

[BP96] 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.

[BST00] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of java without data races. In *Object Oriented Programming Systems, Languages and Applications*, pages 382–400, 2000.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Object Oriented Programming Systems, Languages and Applications*, pages 1–19, 1999.

[CKRW97] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From Sequential to Multi-Threaded Java: An Event Based Operational Semantics. In *Sixth International Conference on Algebraic Methodology and Software Technology*, October 1997.

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

[Com98] Compaq Computer Corporation. *Alpha Architecture Handbook, version 4*. Compaq Computer Corporation, Houston, TX, USA, October 1998.

[DR98] Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *The Journal of Parallel and Distributed Computing*, 49(2):218–244, 1998.

[DS90] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access an omaly detection. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, June 1990.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the Thirteenth International Symposium on Computer Architecture*, pages 434–442. IEEE Computer Society Press, 1986.

[ECM02a] ECMA. C# Language Specification, December 2002. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[ECM02b] ECMA. Common Language Infrastructure (CLI), December 2002. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[Gha96] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1996.

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

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

[Gro00] Supercomputing Technologies Group. *Cilk 5.3.1 Reference Manual*. MIT Laboratory for Computer Science, Cambridge, Massachusetts, 2000.

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

[GS98] Alex Gontmakher and Assaf Schuster. Characterization for Java Memory Behavior. In *Twelfth International Parallel Processing Symposium and Ninth Symposium on Parallel and Distributed Processing*, pages 682–686, 1998.

[GS00a] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency proto col. *IEEE Transactions on Computers*, 49(8):798–813, 2000.

[GS00b] Alex Gontmakher and Assaf Schuster. Java Consistency: Nonoperational Characterizations for Java Memory Behavior. *ACM Transactions on Computer Systems*, 18(4):333–386, 2000.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object Oriented Programming Systems, Languages and Applications*, October 2003.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Int99] Intel Corporation. *IA-64 Application Developer's Architecture Guide, version 1.* Intel Corporation, Santa Clara, CA, USA, May 1999.

[Int02] Intel Corporation. *IA-64 Architecture Software Developer's Manual*, volume 2. Intel Corporation, Santa Clara, CA, USA, October 2002.

[Jav02] Java Specification Request (JSR) 1. Real-time Specification for Java, 2002. http: //jcp.org/jsr/detail/121.jsp.

[Jav03] Java Specification Request (JSR) 121. Application Isolation API Specification, 2003. http: //jcp.org/jsr/detail/121.jsp.

[Jav04] Java Specification Request (JSR) 133. Java Memory Model and Thread Specification Revision, 2004. http: //jcp.org/jsr/detail/133.jsp.

[Jon87] Geraint Jones. *Programming in Occam*. Prentice-Hall International, 1987.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Kot01] Vishnu Kotrajaras. *Towards an Improved Memory Model for Java*. PhD thesis, Department of Computing, Imperial College, August 2001.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[KY95] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 196 – 204, La Jolla, California, June 1995.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.

[LB98] Bil Lewis and Daniel J. Berg. *Multithreaded programming with pthreads*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1998.

[Lea04] Doug Lea. JSR-133 Cookbook, 2004. Available from http: //gee.cs.oswego.edu/dl/jmm/cookbook.html.

[LP01] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 2001.

[Man01] Jeremy Manson. Virtual Machine Implementation Issues for the Revised Java Me mory Model, December 2001. M.S. Thesis, Available from http: //www.cs.umd.edu/users/jmanson/java.html.

[MP01a] Jeremy Manson and William Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.

[MP01b] 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.

[MP02] Jeremy Manson and William Pugh. The Java Memory Model Simulator. In *Workshop on Formal Techniques for Java Programs, in association with ECOOP 2002*, June 2002.

[MPC90] Samuel P. Midkiff, David A. Padua, and Ron G. Cytron. Compiling programs with user parallelism. In August, editor, *Proceedings of the 3rd Workshop on Languages and Compilers for Parallel Computing*, 1990.

[MS96] Maged Michael and Michael Scott. Simple, fast and practical non-blocking and blocking concurrent queue algorithms. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, 1996.

[NR86] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[PC90] 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.

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

[Ruf00] Eric Ruf. Effective Synchronization Removal for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC Canada, June 2000.

[SAR99] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture*, 1999.

[Sar04] Vijay Saraswat. Concurrent Constraint-based Memory Machines: A framework for Java Memory Models. Technical report, IBM TJ Watson Research Center, March 2004.

[Sch89] Edith Schonberg. On-the-fly detection of access anomalies. In *ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 285–297, 1989.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, April 1988.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, Reading Mass. USA, 3rd edition, 1997.

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

[Yan04] Yue Yang. *Formalizing Shared Memory Consistency Models for Program Analysis*. PhD thesis, University of Utah, 2004.

[YGL01] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Specifying Java thread semantics using a uniform memory model. In *ACM Java Grande Conference*, November 2001.

[YGL02] 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.