# SPECIAL POPL ISSUE
# The Java Memory Model

Jeremy Manson
Purdue University
and
William Pugh
University of Maryland, College Park
and
Sarita Adve
University of Illinois at Urbana-Champaign

---

Java's original *memory model*, which describes how threads interact through memory, was fundamentally flawed. Some language features, like the volatile field modifier, were under-specified: their treatment was so weak as to render them useless. Other features, including reads and writes of ordinary object fields, were unintentionally over-specified: the memory model prevented 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.

The new Java memory model, described in this article, fixes these problems. It also defines what it means for a program to be correctly synchronized, and provides a simple interface for such programs: they exhibit sequential consistency. Its novel contribution is the requirement that the behavior of incorrectly synchronized programs be bounded by a well defined notion of causality. The causality requirement is strong enough to respect the safety and security properties of Java and weak enough to allow standard compiler and hardware optimizations. To our knowledge other models are either too weak, because they do not provide sufficient safety and security guarantees, or are too strong, because they rely on a notion of data and control dependences that precludes some standard compiler transformations.

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

These issues had never been addressed for any programming language: in addressing them for Java, we provide a framework for all multithreaded languages. The work described in this article has been incorporated into the version 5.0 of the Java programming language. In addition to this, we believe the model described here could prove to be a useful basis for reasoning about other programming languages that currently lack well-defined models, such as C++ and C#.

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

---

## 1. INTRODUCTION

Much of the work done in modern computer science focuses on one of two goals. 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 easily be violated.

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

It is fairly easy (by comparison) for a processor to maintain the intra-thread semantics of a program when dealing with a single-threaded program. 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 allows for a wide variety of program transformations.

The real difficulty comes when there are multiple threads of instructions executing at the same time, and those threads are interacting. In this case, the limiting factors we impose for single-threaded execution are not enough – program transformations can result in bizarre side effects. Compilers that do not take into account the existence of multiple threads can introduce data races or perform other transformations that would violate the semantics of even correctly synchronized programs [Boehm 2005].

In most cases, this is not a major problem. For most modern multithreaded programming patterns, the programmer specifies ordering constraints by defining how threads communicate with explicit mechanisms. Within these specified constraints, 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 a multithreaded system specifies how memory actions (e.g., reads and writes) in a program will appear to execute to the programmer, and specifically, which value each read of a memory location may return.

Every hardware and software interface of a system that admits multithreaded access to shared memory requires a memory model. The model determines the transformations that the system (compiler, virtual machine, or hardware) can apply to a program written at that interface. For example, given a program in machine language, the memory model for the machine language / hardware interface will determine the optimizations the hardware can perform.

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

The model also impacts the programmer; the transformations it allows (or disallows) determine the possible outcomes of a program, which in turn determines which design patterns for communicating between threads are legal. Without a well-specified memory model for a programming language, it is impossible to know what the legal results are for a program in that language. For example, a memory model is required to determine what, if any, variants of double checked locking [Schmidt and Harrison 1996] are valid within a particular language.

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. As we shall see, it disallowed many standard compiler transformations. For more details on this, see Section 8.

The work described in this article has been adapted to become part of Java Specification Request (JSR) 133 [Java Specification Request (JSR) 133 2004], a new memory model for Java; it is included in the 5.0 release of the Java programming language. It incorporates and extends previous work by the same authors [Manson et al. 2005].

### 1.1 Sequential Consistency

The memory model must strike a balance between ease-of-use for programmers and implementation flexibility for system designers. The model that is most commonly assumed and easiest to understand is *sequential consistency* [Lamport 1979]. 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 optimiza-

Initially, x == y == 0

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

r1 == r2 == r3 == 0 is legal behavior

Fig. 1. Incorrect Synchronization Can Lead To Surprising Results

tions – this is an informal definition of what we shall refer to as 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 it 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. This can be a serious limitation, since many important optimizations involve reordering program statements.

Figure 1 provides a simple example of how a lack of sequential consistency can result in surprising behavior. As we shall see, this program is incorrectly synchronized – shared variables are accessed concurrently without synchronization. Under sequential consistency, either the write or `x` or the write to `y` must occur before the reads in statements 3 and 5. Thus, it would seem that either `r2` or `r3` should be 1. One common compiler optimization involves having the value read for `r1` reused for `r2`: they are both reads of `x` with no intervening write by Thread 1. If this optimization is applied, then the behavior `r1 == r2 == r3 == 0` could result.

Recently, hardware designers have developed techniques that alleviate some of the limitations of sequential consistency by reordering accesses speculatively [Gharachorloo et al. 1991; Ranganathan et al. 1997]. The reordered accesses are committed only when they are known to not be visible to the programmer. Compiler techniques to determine when reorderings are safe have also been proposed [Shasha and Snir 1988; Sura et al. 2002], but are not yet comprehensively evaluated or implemented in commercial compilers.

A common method to overcome the limitations of sequential consistency is the use of relaxed memory models, which allow more optimizations [Dubois et al. 1986b; Gharachorloo et al. 1990a; IBM 1983; May et al. 1994; Sites and Witek 1995; Weaver and Germond 1994]. Many of these models were originally motivated by hardware optimizations and are described in terms of low-level system attributes such as buffers and caches. Generally, these models have been hard to reason with and, as we show later, do not allow enough implementation flexibility either. Having said this, Java's memory model is a relaxed one.

To achieve both programming simplicity and implementation flexibility, alternative models, referred to as *data-race-free* models [Adve 1993; Adve and Hill 1990; 1993] or *properly-labeled* models [Gharachorloo 1996; Gharachorloo et al. 1992], have been proposed. This approach exploits the observation that good programming practice dictates that programs be correctly synchronized (data-race-free); a data race is often a symptom of a bug. The data-race-free models formalize correct

Initially, x == y == 0

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

Can r1 == r2 == 42?

Fig. 2. A Motivation for disallowing some cycles

Initially, `x == y == 0`

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

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

Fig. 3. Compilers Can Think Hard about when Actions are Guaranteed to Occur

programs as those that are data-race-free, and guarantee the simple semantics of sequential consistency for such programs. For full implementation flexibility, these models do not provide any guarantees for programs that contain data races. This approach allows standard hardware and compiler optimizations, while providing a simple model for programs written according to widely accepted practice.

The example in Figure 1 is a relatively simple one; if the examples were all so simple, a memory model could be defined in terms of allowable reorderings. A great deal of subtlety would not be required. However, this is not the case; we have to deal with the notion of a *causal loop*.

## 1.2 Causal Loops

Figure 2 contains a example of an program for which more careful semantics are required. Like Figure 1, it is incorrectly synchronized. Given this, do we wish to ensure that a value such as 42 will not be assigned to `r1` or `r2`? Since the value is not explicitly written in the program, we say that it appears *out of thin air*; a discussion of how value prediction and speculative execution could lead to this result is presented in Section 3.3.

For a number of reasons having to do with security and providing some level of meaning to incorrect programs, Java's semantics says that this behavior is illegal. If a system could generate an integer value out of thin air as in this example, it might also be able to generate a object reference out of thin air in similar examples, preventing any guarantees about object confidentiality or confinement.

Characterizing precisely what constitutes an out-of-thin-air violation is complicated, and is the core of this paper's contribution. Not all speculations that appear to be self-justifying are security violations, and some can even arise out of current compiler transformations. How we characterize the appropriate set of violations is at the core of the new Java model, and can be found in Section 3.3.

Another example of a causal loop – this time, one that describes an acceptable behavior – can be seen in Figure 3. In order to see the result `r1 == r2 == r3 == 1`, one of the threads must perform its write before it performs its read. But

each write seems dependent upon the read above it. While this value might also seem to come out of thin air, it does not and can result from standard compiler transformations, as discussed in Section 2.2.2.

It is clear, that determining what constitutes an unsafe causal loop is complicated and subtle. The fundamental problem with these sorts of examples is that we have not defined a way to reason from a "first cause". 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 article.

### 1.3 Why Solve This Problem?

The bulk of the effort for the revision, and our focus here, was on understanding the requirements for incorrectly synchronized code. The previous strategy of leaving the semantics for incorrect programs unspecified is inconsistent with Java's security and safety guarantees. Such a strategy has been used in some prior languages. For example, Ada simply defines unsynchronized code as "erroneous" [Ada Joint Program Office 1995]. Reynolds [Reynolds 2004] describes data races as "catastrophic", any programs that contain them as "wrong", and states that if a program is "wrong", then no possibilities other than "wrong" for its results must be considered.

The reasoning behind this is that since such code is incorrect (on some level), no guarantees should be made when it occurs. This is similar to the strategy that some languages take with array bounds overflow – unpredictable results may occur, and it is the programmer's responsibility to avoid these scenarios.

The above approach does not lend itself to the writing of secure and safe code. In an ideal world, every programmer would write correct code all of the time. In our world, programs frequently contain errors; not only does this cause code to misbehave, but it can also allows attackers to violate safety assumptions in a program (as is true with buffer overflows). Our earlier work has described the dangers of such scenarios in more detail [Manson and Pugh 2001a].

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, they won't be able to know what their code is doing wrong.

An alternative is to to design a language that is completely free from data races [Reynolds 1978; Bacon et al. 2000; Flanagan and Freund 2000]. However, because they eliminate data races by enforcing mutual exclusion, such languages only allow limited support for wait- and lock-free data structures [Michael and Scott 1996].

### 1.4 Approach

The approach taken in this article 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. These requirements 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. It was also necessary to ensure that the model was largely compatible with existing coding practice and JVM implementations. At the end of this process, a consensus emerged as to what the informal requirements for a programming language memory model are. As the requirements emerged, a memory model took shape.

In the coming sections, you may notice references to guarantees and optimizations that we describe as "reasonable" or "unreasonable"; these notions came from long discussions with this group. The requirements that were gathered are largely described in Section 2, with additional material in Section 5. When this process was finished, we were able to settle on a model that reflected those requirements (Sections 3 and 4), and verified that the final specification met the requirements (Section 6).

We focused primarily 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 Section 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 refined the notion of a `volatile` field (Section 2) to supplant 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 Section 7.

The second bullet is somewhat more complex. The fundamental question is described above: what kinds of causal loops are acceptable, and what kinds must be disallowed? This question frames the discussion of causality (Section 3), the most important contribution of this article.

## 2. BASIC REQUIREMENTS FOR THE JAVA MEMORY MODEL

Of all the properties desired in the Java Memory Model that were not supported by the original Java Memory Model, three properties are most important, and are discussed in this section:

—The consistency guarantees for programmers that are made for correctly written (or correctly synchronized) programs.

—The guarantee that compiler writers and architects can perform basic code transformations / optimizations.

—The guarantee that programmers will be able to write code that uses non-blocking synchronization (using volatile variables).

The other primary requirements relate to causality; as they are the major contribution of this paper to memory model literature, they are discussed in a separate

section. The causality requirements, together with the full memory model, are described in full in Sections 3 and 4. More in-depth discussion of several other informal issues can be found in Section 5.

### 2.1 Guarantees for Correctly Synchronized Programs

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* (the monitor may be used to protect more than just that object, of course). 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. The lock and unlock actions are referred to as *synchronization actions*; other synchronization actions are described in Section 4.3.

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

There is a partial order called *program order* that, for each thread, provides a total order over the actions performed by that thread. There is also a total order over all synchronization actions, consistent with program order, 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 [Lamport 1978] 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 Section 4 – we will just focus on these for now. We now have the key to understanding why the behavior in Figure 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) [Shasha and Snir 1988]. 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.

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

Fig. 4. Surprising Correctly Synchronized Program

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 1, for example, is incorrectly synchronized.

Prior work [Adve 1993; Gharachorloo 1996; Adve and Hill 1990; Gharachorloo et al. 1990a] 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 1.1). Thus, we make a guarantee (which we call DRF) about the possible behavior of correctly synchronized programs.

*DRF.* Correctly synchronized programs have sequentially consistent semantics.

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

This requirement leads to some interesting corner cases. For example, the code shown in Figure 4 [Adve 1993] 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.

DRF disallows some subtle program transformations. For example, an aggressive write speculation cannot predict that a write to `y` of 42 will 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.

## 2.2 Allowed Code Transformations

In earlier sections, we outlined how important it is for compilers and processors to reorder program actions – it is the engine that drives most optimizations. Absent a convincing reason otherwise [Boehm 2005], in the absence of synchronization operations, we want to allow compilers and processors to perform the optimizations and transformations that they would apply to single threaded code.

Before 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 = r1.x;` | |

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

After 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`

Fig. 5. Optimization Disallowed By Original Java Memory Model

In the following, we describe some situations in which we wish to allow the compiler freedom to perform reorderings that might be surprising to someone expecting a sequentially consistent execution. This is not intended to be a complete list of such situations, but only an illustrative set of examples.

2.2.1 *Reordering Independent Statements.* We say two statements are *independent* if they are not control or data dependent on each other. In general, we want to allow independent statements in a program that are not involved in synchronization to be reordered. This property is formalized in Section 6.1.1, where we also provide a proof that the model described in Section 4 obeys it. Note that our reordering guarantee actually allows the reordering of statements that were not adjacent in the original program: to achieve this, it is simply necessary to perform repeated reorderings, transitively, until the statement appears in the desired location.

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

Figure 5 shows a behavior that results from standard compiler optimization and is allowed by our model. This behavior was unintentionally prohibited by the original Java memory model [Pugh 1999]; that model did not allow reads of the same variable to be reordered within a thread. However, reusing the value of `r2` for the second read of `r1.x` in Thread 1 has the logical effect of reordering the reads of `r3.x` and `r1.x`. This is a standard compiler optimization, and must be permitted.

2.2.2 *Dependence Breaking Analysis and Transformations.* The reordering discussed in Section 2.2.1 does not interfere too heavily with our notion of cause and effect; the actions being reordered are clearly independent. In this section, we discuss reordering of actions that might seem to be dependent and violate causality.

Compilers can perform a number of analyses and transformations that have the effect of removing dependencies, and many of these analyses and transformations are ones that need to be allowed by the Java memory model. For example, the Java memory model allows the results shown in Figure 6. This behavior may seem cyclic, as the write to `b` is control dependent on the reads of `a`, which see a write to `a` that is data dependent on a read of `b`; that read of `b` must see the aforementioned write to `b`. However, the compiler should be allowed to

—eliminate the redundant read of `a`, replacing `r2 = a` with `r2 = r1`, then

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

Fig. 6. Effects of Redundant Read Elimination

Initially, `x == y == 0`

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

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

Fig. 7. Sometimes Dependencies are not Obvious

—determine that the expression `r1 == r2` is always true, eliminating the conditional branch `3`, and finally

—move the write `4: b = 2` early.

If the compiler performs this transformation, the assignment of 2 to `b` will always be guaranteed to happen; the second read of `a` will always return the same value as the first. Without information about the redundant read, 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`.

A different situation is shown in Figure 7, in which we allow the behavior `r1 == r2 == 1`. This seems unexpected, since it seems as though the write in each thread is dependent upon the read in each thread, and thus one of the reads would have to be performed before either of the writes, preventing this behavior. But a compiler can easily determine that the only values written to `y` are the constant values 0 and 1. As a result, the compiler can determine that `r2` is either 0 or 1, and that thread 2 always writes 1 to `x` and remove the dependence from the read of `y` to the write of `x` in thread 2. Having removed the dependency, the compiler is allowed to move the write of `x` to the start of thread 2. If the resulting code were executed in a sequentially consistent way, it would result in the circular behavior described.

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.

Fig. 8. Simple Use of Volatile Variables

Figure 3 shows a similar but even more surprising behavior. In this case, the compiler would have to perform a more slightly deeper analysis to determine that the values of `x` and `y` are guaranteed to be either 0 or 1. One possible such analysis would be an analysis that determines the bit width required to represent integer values in a program [Stephenson et al. 2000].

It is clear, then, that compilers can perform many analyses and transformations that remove dependencies. In this case, if a compiler can determine that an action will always happen (with the same value written to the same variable), the action can be performed in advance of things it seems to be dependent upon.

### 2.3 Volatiles and Non-Blocking Concurrency

We have talked about using locks to establish happens-before edges and synchronize information between threads. In order to allow for non-blocking techniques that communicate between threads, we also want to allow the use of *volatile* variables to synchronization information between threads.

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 8. 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 [Michael and Scott 1996]). Other issues involving volatiles are discussed in more detail in Section 5.2.2.

We expand the notion of a synchronization action, as defined in Section 2.1, to include reads of and writes to volatile variables. The resulting property reflects the way synchronization is used to communicate between threads. As with unlocks and subsequent locks, volatile writes are ordered before subsequent volatile reads of the same variable.

The happens-before relation is how we express the requirements on volatile variables, as we can see from Figure 8. Assume Thread 2's read of `ready` sees the write to `ready`. The write must therefore occur before the read in synchronization order. Thread 1's write of `x` must happen-before Thread 2's read of `x`, because program order and synchronization order transitively compose to provide the happens-before order. As a result of this, Thread 2's read must return the new value of x. In more detail: the initial value of x is assumed to happen-before all actions. There is a path of happens-before edges that leads from the initial write, through the write to `x`, to the read of `x`. On this path, the initial write is seen to be overwritten. Thus, if the read of `ready` in Thread 2 sees `true`, then the read of `x` must see the value `1`.

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

Note that the new semantics for volatile fields constitute a change from the original Java memory model. In the original model, while volatile reads and writes needed to be performed directly to shared memory and not cached, it also allowed volatile and non-volatile memory actions to be freely reordered. The new model does not allow this: the happens-before edge from a volatile write to a volatile read ensure that non-volatile memory actions cannot be arbitrarily reordered past volatile writes or before volatile reads.

In addition to operations on locks and volatile variables, other actions, such as starting or joining on a thread, can induce happens-before edges. These are listed in detail in Section 4.3.

## 3. CAUSALITY IN A MEMORY MODEL

In Section 1.1, 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 section, we discuss the necessary guarantees for unsynchronized code. In other words, we address the question: what is acceptable behavior for a incorrectly synchronized multithreaded program? Answering this question allows us to synthesize our requirements and formulate a workable memory model.

### 3.1 Sequential Consistency Memory Model

Section 1.1 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.

### 3.2 Happens-Before Memory Model

We can describe a simple, interesting memory model using the happens-before relation introduced in Section 2 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 Section 2.

Each legal execution under this model has several properties/requirements (Section 4 provides a more formal version of the requirements):

—There is a synchronization order over synchronization actions, synchronization actions induce synchronizes-with edges between matched actions, and the transitive closure of the synchronizes-with edges and program order gives an order known as the happens-before order (as described in Section 2.1).

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

Fig. 9. Surprising Correctly Synchronized Program

—For each thread $t$, the actions $t$ performs in the execution are the same as those that $t$ would generate in program order in isolation, given the values seen by the reads of $t$ in the execution.

—A rule known as *happens-before consistency* (see below) determines the values a non-volatile read can see.

—A rule known as synchronization order consistency (see below) determines the value a volatile read can see.

Synchronization order consistency says that (i) synchronization order is consistent with program order and (ii) each read $r$ of a volatile variable $v$ sees the last write to $v$ to come before it in the synchronization order (Section 2.3).

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

—$r$ does not happen-before $w$ (i.e., it is not the case that $r \stackrel{hb}{\rightarrow} w$) – a read cannot see a write that happens-after it, and

—there is no intervening write $w'$ to $v$, ordered by happens-before (i.e., no write $w'$ to $v$ such that $w \stackrel{hb}{\rightarrow} w' \stackrel{hb}{\rightarrow} r$) – the write $w$ is not overwritten along a happens-before path.

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 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 code in Figure 1 is happens-before consistent. 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.

### 3.3 Causality

Although it is simple, the happens-before memory model allows unacceptable behaviors. Notice that the behavior we want to disallow in Figure 9 (the same as Figure 4) is consistent with the happens-before memory model. 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. Under the happens-before memory model, if both write actions write out the value 42, and both reads see those writes, then both reads see values that they are allowed to see. Even though this is a correctly synchronized program, executions that are not sequentially consistent are legal. Therefore, executions that are legal under the happens-before memory model can violate our DRF guarantee.

Nevertheless, the happens-before memory model provides a good outer bound for our model; all executions must be happens-before consistent.

An example that is similar to the one in Figure 9 can be seen in Figure 2. In this case, the writes will always happen; there is, therefore, a data race in this code. The values written are data dependent on the reads. A system that performed aggressive value prediction and speculation could predict that Thread 1 will read 42 for `x` and speculatively write the value 42 to `y`. This would allow Thread 2 to read 42 for `y` and write it out to `x`, which would allow Thread 1 to read 42 for `x`, and justify its original speculative write of 42 for `y`. A self-justifying write speculation, or *causal loop*, like that one can create serious security violations and needs to be disallowed.

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.

Unlike the code in figure 9, this is not correctly synchronized program, because there is a data race between Thread 1 and Thread 2. However, as it is in many ways a very similar example, we would like to provide a similar guarantee. In this case, we say that the value 42 cannot appear *out of thin air*.

In fact, the behavior shown in Figure 2 may be even more of a cause for concern than the behavior shown in Figure 9. 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.

Another example of this sort of behavior can be seen in Figure 10. 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.

The missing link between our desired semantics and the happens-before memory

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

Fig. 10. An Unexpected Reordering

model is one of *causality*. In Figures 2, 9 and 10, the actions which caused the illegal writes to occur were, themselves, caused by the writes. This kind of circular causality is undesirable in general, and these examples must be prohibited in the Java memory model.

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

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 2, no read can ever see it.

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

One way to think about these issues is to consider when actions can occur in an execution. 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 2, 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

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

Fig. 11. A Complicated Inference

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 as an iterative process. Starting with some committed set of actions, generate all the possible "well-behaved" executions containing these actions. Then, use those well-behaved executions to determine further actions that can be reasonably performed early. Commit those actions. Repeat this process 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 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 1, the writes to `x` and `y` can occur first because they will always occur in sequentially consistent executions. In Figure 6, 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 2, the writes of 42 to `y` and `x` cannot happen, because they do not occur in any sequentially consistent execution. So, as a first approximation to an out-of-thin-air guarantee, we could state that a write can only occur earlier in an execution that it would have given the program order if it would have been able to occur in a sequentially consistent execution. This is only a first approximation to causality: it is not a bad starting point, but does not cover all of our bases.

So, if a write's being caused (in some sense) by its occurrence in a sequentially consistent execution is not enough, what is? 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 2 provides an example of a result that is clearly unacceptable, but other examples may be less straightforward.

In this section we explore a more subtle notion of what defines causality; as usual, we base this notion on some subtle examples of the compiler optimizations we would like to allow.

3.3.1 *A result that should be allowed.* First, consider the code in Figure 11. 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

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?

Fig. 12. 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?

Fig. 13. Can Threads 1 and 2 see 42, if Thread 4 didn't write to `x`?

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. It could then, quite reasonably, change `r1 = x` to `r1 = 42`; this would allow `y = r1` to be transformed to `y = 42` and performed earlier, resulting in the behavior in question.

This is a reasonable transformation; a compiler should be able to perform a whole-program analysis that can determine what the possible values assigned to a variable will be. This sort of transformation needs to be balanced with the out-of-thin-air requirement.

3.3.2 *Results that must be prohibited.* The examples in Figures 12 and 13 are similar to the examples in Figures 2 and 9, with one major distinction. In those examples, the value 42 could never be written to `x` in any sequentially consistent execution. Thus, our initial out-of-thin-air guarantee prevented the value 42 from appearing. In the examples in Figures 12 and 13, 42 can be written to `x` in some sequentially consistent executions (specifically, 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 12 and 13.

Notice that the code in Figure 11 is quite similar to the code in Figures 12 and

13. 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 out-of-thin-air guarantee based on sequential consistency is therefore not strong enough to encapsulate this notion.

The examples in Figure 12 and 13 are significantly different from the example in Figure 11. One way of articulating this difference is that in Figure 11, 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 12 and 13, we need to reason about the outcome of a data race to determine that `r1 = x` can see 42.

3.3.3 *Our approach to causality.* The key observation here is that early execution of an action does not result in an undesirable causal cycle *if its occurrence is not dependent on a read returning a value from a data race.* This insight led to our final notion of a "well-behaved execution": a read that is not yet committed must return the value of a write that is ordered before it by happens-before.

As a result of this, we revise our out-of-thin-air guarantee. We do not reason about whether an action can happen based on whether it happens in a sequentially consistent execution. Instead, we base our reasoning on whether it can occur if no values are seen because of a data race. In short, *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 a "well-behaved" execution to "justify" further executions where writes occur early. Given a well-behaved execution, we may commit any of the uncommitted writes that occur in it (with the same address and value). We also may commit any uncommitted reads that occur in such an execution, but additionally require that the read return the value of a previously committed write in both the justifying execution and the execution being justified (we allow these writes to be different).

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

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

We can use this out-of-thin-air guarantee as a basic principle to reason about multithreaded programs. Consider Figures 12 and 13. 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 14. 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

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

Fig. 14. Another Out Of Thin Air Example

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. In Section 4, we formalize this model.

## 4. THE JAVA MEMORY MODEL

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

The Java memory model determines whether a set of inter-thread actions (e.g., reads and writes of shared variables, locks and unlocks of monitors) represents a legal execution. It relies upon being able to determine whether a set of inter-thread actions is consistent with the intra-thread semantics of the code executed by that thread, but does not define intra-thread semantics. As the Java memory model is only concerned with inter-thread actions, many details of the Java language (such as method invocation) are immaterial to it.

The ultimate result of the Java memory model is to define (Section 4.7) the set of observable behaviors of a program. This can be used, for examine, to determine if a program transformation is legal: a transformation is legal if it does not introduce any new observable behaviors.

### 4.1 Variables

A *variable* refers to a static variable of a loaded class, a field of an allocated object, or element of an allocated array. A variable may contain a primitive value (e.g., an integer), or a reference to an object. The system must maintain the following properties with regards to variables and the memory manager:

—It must be impossible for any thread to see a variable before it has been initialized to the default value for the type of the variable.

—The fact that a garbage collection may relocate a variable to a new memory location or reuse a memory location is immaterial and invisible to the semantics.

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

Java variables local to a method are not stored in shared memory and operations on them are not governed by the Java memory model.

### 4.2 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 4.3. 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, a partial order over actions, 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 4.6. 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 4.6.

*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.

### 4.3 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 $x \stackrel{hb}{\rightarrow} y$ 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 $x \stackrel{hb}{\rightarrow} y$. If an action $x$ synchronizes-with a subsequent action $y$, then we also have $x \stackrel{hb}{\rightarrow} y$. Further more, happens-before is transitively closed. In other words, if $x \stackrel{hb}{\rightarrow} y$ and $y \stackrel{hb}{\rightarrow} z$, then $x \stackrel{hb}{\rightarrow} z$.

(3) **Definition of sufficient synchronization edges**. A set of synchronizes-with 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 $\stackrel{e}{\rightarrow}|_d$ to represent the restriction of the partial order $\stackrel{e}{\rightarrow}$ to the elements in $d$: for all $x, y \in d$, $x \stackrel{e}{\rightarrow} y$ if and only if $x \stackrel{e}{\rightarrow}|_d y$. If either $x \notin d$ or $y \notin d$, then it is not the case that $x \stackrel{e}{\rightarrow}|_d y$.

### 4.4 Well-Formed Executions

We only consider well-formed executions. An execution $E = \langle P, A, \stackrel{po}{\rightarrow}, \stackrel{so}{\rightarrow}, W, V, \stackrel{sw}{\rightarrow}, \stackrel{hb}{\rightarrow}\rangle$ 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) **The synchronization order has an order less than or equal to omega** This means that for each synchronization action $x$, only a finite number of synchronization actions occur before $x$ in the synchronization order.

(3) **Synchronization order is consistent with program order** This 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.

(4) **Lock operations are consistent with mutual exclusion** This is defined as requiring that for each lock operation $l$ on a monitor $m$ by thread $T$, the number of unlock operations performed by a thread $T' \neq T$ on $m$ that come before $l$ in the synchronization order is equal to the number of lock operations performed by a thread $T'$ on $m$ that come before $l$ in the synchronization order.

(5) **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.

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

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

### 4.5 Causality Requirements for Executions

A well-formed execution

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

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 $A_i$. 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 $\overset{po_i}{\rightarrow}$ (the transitive reduction of $\overset{hb_i}{\rightarrow}$ is the minimum relation that has the same transitive closure as $\overset{hb_i}{\rightarrow}$). 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 $x \in C_i$.

### 4.6 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. More informal discussion of these issues can be found in Section 5.3.

The observable behavior of a program is defined by the sets of external actions that the program may perform during a finite period. 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. The behavior of printing "Hello" an infinite number of times is not an observable behavior, since we can not observe an infinite number of prints.

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 external *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.

### 4.7 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

Initially, `x = y = 0`

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

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

Fig. 15. A Standard Reordering

$A$, 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 $x \xrightarrow{hb} y$ or $x \xrightarrow{so} y$, then $x \in O$.

Note that a set of observable actions is not restricted to external actions. Informally, the observable actions represent all the actions that occur during the period the execution is observed. External actions that are in the set of observable actions represent the observable external behavior of a program.

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.

## 4.8 Examples

We have seen a number of examples of behaviors for which the Java memory model must have specific results. 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.

4.8.1 *Simple Reordering.* As a first example of how the memory model works, consider Figure 15. 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

| 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 | $C_4$ | $E$ |

Fig. 16. Table of commit sets for Figure 15

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

Fig. 17. A Correctly Synchronized Program

happens-before memory model, so we only have to ensure that it complies with the causality rules in Section 4.5.

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 $C_1$.

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 $C_3$.

4.8.2 *Correctly Synchronized Programs.* It is easy to see how most of the guarantees that we wished to provide for volatile variables are fulfilled; Section 4.3 explicitly provides most of them. A slightly more subtle issue is shown in Figure 17. 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 the that read of `a` by Thread 1 and the write to `a` in Thread 2 are ordered by happens-before

In order for the read of `a` to see the value 1 (and therefore result in a non-

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

Fig. 18. Must Disallow Behavior by Ensuring Happens-Before Order is Stable

Initially, v is volatile and v = false

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

Fig. 19. If we observe print message, Thread 1 must see write to v and terminate

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 $C_i$. 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 18. 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; thus 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).

4.8.3 *Observable Actions.* Figure 19 is the same as Figure 24 in Section 5.3. Our requirement for this program was that if we observe the print message from Thread 2, and no other 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 4.6. 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.

## 4.9 Finalization

In Java, each object has a protected method called `finalize` (or, a *finalizer*) which is called when the garbage collector reclaims the allocated space for that object. A thread that invokes a finalizer is guaranteed not to be holding any user-visible locks (a lock that can be explicitly acquired by the programmer).

The Java language specification makes very weak guarantees about when a finalizer is invoked. For example, if the only references to an object are located on the stack, then the fields of that object may be allocated in registers. As a result, there would be no references to that object, and it can be finalized.

An object cannot be considered finalizable until all of its constructors have finished. The constructor for class `Object` must be invoked and complete normally in order for the object to be finalizable; other constructors may terminate by throwing exceptions. 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.

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.

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 set of *reachability decision points*. For any given reachability decision points $d$, each action either *comes-before* $d$ or *comes-after* $d$. 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 $d$, then $w$ must come-before $d$. If $x$ and $y$ are synchronization actions on the same variable or monitor such that $x \stackrel{so}{\rightarrow} y$ and $y$ comes-before $d$, then $x$ must come-before $d$.

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 $d$ are those that can be shown to be reachable by the application of these rules:

—An object $B$ is definitely reachable at $d$ 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. $\neg(w_2 \stackrel{hb}{\rightarrow} w_1)$, and both $w_1$ and $w_2$ come-before $d$.

—An object $B$ is definitely reachable from $A$ at $d$ 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. $\neg(w_2 \stackrel{hb}{\rightarrow} w_1)$, and both $w_1$ and $w_2$ come-before $d$.

—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 $d$,

—$X$ must not be definitely reachable at $d$ from static fields,

—All active uses of $X$ in a thread $t$ that come-after $d$ must occur in the finalizer invocation for $X$ or as a result of thread $t$ performing a read that comes-after $d$ of a reference to $X$.

—All reads that come-after $d$ that see a reference to $X$ must see writes to elements of objects that were unreachable at $d$, or see writes that came after $d$.

If an object $X$ marked as finalizable at $d$, then

—$X$ must be marked as unreachable at $d$,

—$d$ must be the only place where $X$ is marked as finalizable,

—actions that happen-after the finalizer invocation must come-after $d$

## 5. OTHER MOTIVATIONAL ISSUES

In Section 2, we developed an informal motivation for much of the Java memory model. However, there were many decisions that are reflected in the formalism that were not covered in that section. This section provides more detailed coverage for some of the more minor details of the model, including:

—Additional opportunities for reordering, including synchronization elimination,

—Additional requirements for volatile variables, to allow programmers to write non-blocking synchronization, and

—Requirements for the observable behavior of a program.

### 5.1 Synchronization Elimination

One possible memory model would require that all synchronization actions have happens-before edges with all other synchronization actions. Our model does not have this property: lock and unlock actions only have happens-before edges with other lock and unlock actions **on the same monitor**. Similarly, accesses to a volatile variable only create happens-before edges with accesses to the same volatile variable. This property provides several new opportunities for program transformation.

| Before Coarsening | After Coarsening |
|---|---|
| `x = 1;`<br>`synchronized(someLock) {`<br>`  // lock contents`<br>`}`<br>`y = 2;` | `synchronized(someLock) {`<br>`  x = 1;`<br>`  // lock contents`<br>`  y = 2;`<br>`}` |

Fig. 20. Example of Lock Coarsening

There have, for example, been many optimizations proposed [Ruf 2000; Choi et al. 1999] 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.

If we forced all synchronization actions to have happens-before edges 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: synchronization actions that only introduce redundant happens-before edges can be treated as if they don't impose any reordering barriers on memory reads or writes.

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

5.1.1 *Lock Coarsening. Computation lock coarsening* is closely related to synchronization elimination. 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 20. Lock coarsening has been shown to be effective in increasing concurrency [Diniz and Rinard 1998].

As we have mentioned, an acquire ensures an ordering with a previous release in the synchronization order. Consider an action that takes place before an acquire. It may or may not have been visible to actions that happen-before the previous release, depending on how the threads were 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 20. 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.

Similarly, the release ensures an ordering with a subsequent acquire. Consider an action that takes place after a release. It may or may not have been visible to particular actions that happen-after the subsequent acquire, depending on how the threads were 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 20, where the write to `y` is moved up inside the `synchronized` block.

Initially, `v1 == v2 == 0`, `v1` and `v2` are volatile.

| 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?

Fig. 21. Volatiles Must Occur In A Total Order

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

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

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

Fig. 22. Strong or Weak Volatiles?

All of this is simply a roundabout way of saying that accesses to normal variables can be reordered with a following volatile read or lock acquisition, or a preceding volatile write or lock release. This implies that normal accesses can be moved inside locking regions, but (for the most part) 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 Section 8 and in [Lea 2004].

## 5.2 Additional Constraints on Volatile Fields

In Section 2.3, we discussed how, to support non-blocking concurrency, we introduced constraints on fields marked `volatile`. In this section, we discuss additional constraints on these fields that help with writing safe concurrent code.

5.2.1 *Operations on Volatiles are sequentially consistent.* Figure 21 gives us an interesting glimpse into the guarantees we provide to programmers. The writes 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: all accesses to volatile variables are performed as if they were sequentially consistent with respect to each other.

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.

| Thread 1 | Thread 2 |
|---|---|
| while (true)<br>  synchronized (o) {<br>  if (done) break;<br>  think<br>  } | synchronized (o) {<br>  done = true;<br>} |

Fig. 23. Lack of fairness allows Thread 1 to never surrender the CPU

Initially, v is volatile and v = false

| Thread 1 | Thread 2 |
|---|---|
| while (!v);<br>print("Thread 1 done"); | v = true;<br>print("Thread 2 done"); |

Fig. 24. If we observe print message, Thread 1 must see write to v and terminate

5.2.2 *Volatile writes synchronize will all following matching reads.* Another issue that arises with volatiles came within the JMM community 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 edge from each write to each subsequent read of that volatile.

—**Weak interpretation** There is a happens-before edge 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 that is referred to as *weak ordering* [Dubois et al. 1986a; Adve and Hill 1990].

In Figure 22, under the weak interpretation, the read of v in each thread might see its own volatile write, allowing the behavior `r1 == r3 == 1`.

The strong interpretation allows a number of atypical but useful coding idioms. Since there does not seem to be any additional implementation cost for enforcing the strong interpretation, it was decided that the Java memory model should support the strong interpretation.

## 5.3 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 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 [Java Specification Request (JSR) 1 2002]). 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 fairness is not a memory model issue. However, the issues are closely related. An example of their interrelation can be seen in Figure 23. 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.

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

Fig. 25. Correctly Synchronized Program

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 5.1.1, 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 24, 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. In terms of our model, the only reason that an action visible to the external world (e.g., a 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.

5.3.1 *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 25. 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 4).

The notion of control dependence that correctly encapsulates this is called *weak control dependence* [Podgurski and Clarke 1990] in the context of program verification. This property has also been restated as *loop control dependence* [Bilardi and Pingali 1996] in the context of program analysis and transformation.

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

Fig. 26. 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

Fig. 27. Behavior that must be allowed

## 5.4 Thread Inlining

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.

The behavior in Figure 26 is very similar to that shown in Figure 13. Given that we have decided the behavior in Figure 13 is unacceptable, it seems reasonable to prohibit both.

Figure 27 shows a code fragment very similar to that of Figure 26. However, for the code in Figure 27, we must allow the behavior that was prohibited in Figure 26. 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 `r2 = x` on line 4 with `r2 = 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.

A program transformation called *thread inlining* could result in the code in Figure 26 being changed into the code in Figure 27: we could derive Figure 27 from

Figure 26 by combining Threads 1 and 2 into a single thread. Alternatively, this change could be effected by an implementation that always scheduled Thread 2 immediately after Thread 1.

The fact that the behavior in Figure 26 is prohibited and the behavior in Figure 27 is allowed is, perhaps, surprising. The Java memory model makes this distinction because it reflects the necessary causality guarantees.

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 can execute.

This is an example where adding happens-before edges 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. The happens-before relation is used to express causality between two actions; if an additional happens-before edge is inserted, the causal relationships change.

## 6. VERIFICATION

In Section 2, we looked at the informal requirements for our memory model, exposing the principles that guided its development. In Section 4, 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 Section 2) for our memory model did, in fact, hold true for it.

Information on the simulator can be found in other publications [Manson and Pugh 2001a; 2001b; Manson 2004]. The rest of this section formalizes some of the properties that we outlined for our memory model, and discusses how we ensured that those properties were realized.

### 6.1 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 fulfills many of the requirements found in Sections 2 and 5 is fairly straightforward. For example, the volatile properties are all explicitly stated as parts of the memory model.

The out of thin air properties are 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.

6.1.1 *Semantics Allows Reordering.* We mentioned earlier that a key notion for program optimization was that of *reordering*. We demonstrated in Figure 1 that standard compiler reorderings, unobservable in single threaded programs, can have observable effects in multithreaded programs. However, reorderings are crucial in many common code optimizations. 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 relation 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 $x \stackrel{hb}{\rightarrow} y$

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. □

6.1.2 *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.

6.1.2.1 *Correctly Synchronized Programs Exhibit Only Sequentially Consistent Behaviors.* As described in Section 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 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. 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 $w$; $\delta'$ 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. □

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 $w \stackrel{hb}{\rightarrow} r$ in $E$, then we didn't have $w \stackrel{hb}{\rightarrow} r$ 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. □

## 7. IMMUTABLE OBJECTS

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 section, 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 section describes.

Figure 28 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 29. `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.

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 section, 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.

```
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

Fig. 28. Example Illustrating Final Field Semantics

Thread 1

```
Global.s = "/tmp/usr".substring(4);
```

Thread 2

```
String myS = Global.s;
if (myS.equals("/tmp"))
  System.out.println(myS);
```

Fig. 29. Without final fields or synchronization, it is possible for this code to print `/usr`

## 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 28. 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 edge. 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 edge 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

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.

Fig. 30. Example of Simple Final Semantics

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 7.3.0.3.

## 7.2 Motivating Examples

7.2.1 *A Simple Example.* Consider Figure 30. 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 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 edge 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 30, 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 31, the final field o.f2 is a reference instead of being a scalar (as it was in

```
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.

Fig. 31. Example of Transitive Final Semantics

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 31, `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 31, Thread 2 reads `a[0]` 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 32. 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 2.2.2 for more discussion of how this affects normal fields).

| 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; | |

Fig. 32. Example of Reachability

| 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.

Fig. 33. Freezes are Passed Between Threads

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.

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.

7.2.3.1 *Freezes Are Passed Between Threads.* Figure 33 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 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.

7.2.3.2 *Semantics' Interaction with Happens-Before Edges.* Now consider Figure 34. We want to describe the interaction between ordinary happens-before edges 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 edges ensure that Thread 2 will see all of the correctly published values. As a result, j will be 42.

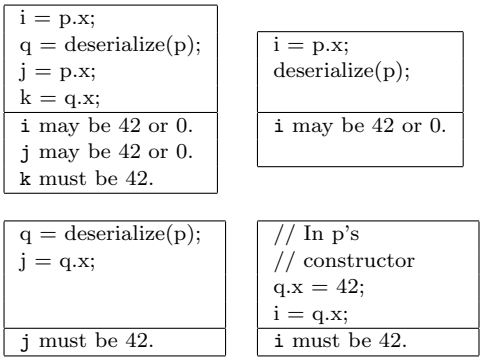
What about the reads in Thread 3? We assume that k does not see a null value: should the normal guarantees for final fields be made? We can answer this by

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..

Fig. 34. Example of Happens-Before Interaction




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.

Fig. 35. Four Examples of Final Field Optimization

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 35. 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 35 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,

*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

Fig. 36. Guarantees Should be made via the Enclosing Object

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.

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 36. 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 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

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;` |

Fig. 37. Cyclic Definition Causes Problems

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.

7.3.0.1 *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 7.3.0.2, 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.

7.3.0.2 *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 $r$.

7.3.0.3 *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 $\overset{mc}{\rightarrow}$ without providing the chain $f \overset{hb}{\rightarrow} a \overset{mc}{\rightarrow} r_1$ 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 38. 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 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

`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.

Fig. 38. Final field example where reference to object is read twice

`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.

Fig. 39. Transitive guarantees from final fields

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 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 39 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 40 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.

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

`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.

Fig. 40. Yet Another Final Field Example

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 31. 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 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

`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;` | `}` |

Fig. 41. Happens-Before Does Matter

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 32. 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 41. 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 [Weaver and Germond 1994] 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 [Intel Corporation 2002] and Alpha [Compaq Computer Corporation 1998] 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. It is interesting to note that Intel's IA-64 originally required a barrier in this case, but the authors were able to convince them that such a barrier was not desirable for good programming practice.

The notable exception to this is the Alpha processor. However, there are no plans for future versions of Java to be implemented on Alpha processors. We shall therefore not discuss alternative implementation strategies here. Those with more interest can consult other sources [Manson 2004].

7.6.2 *Distributed Shared Memory Based Systems.* Imagine a Lazy Release Consistent (LRC) machine (see [Keleher et al. 1992]): 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.

## 8. RELATED WORK

### 8.1 Architectural Memory Models

Most work on memory models has been done because of the need to verify properties of hardware architectures. Adve and Gharachorloo provide a primer for this work [Adve and Gharachorloo 1996]. An early discussion of memory models can be found in [Lamport 1978], 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 3.3).

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; |

Fig. 42. Can r2 = 1, r3 = 0?

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.

8.2.0.1 *SPARC.* The SPARC processor has three different memory models: Total Store Order (TSO), Partial Store Order (PSO) and Relaxed Memory Order (RMO) [Weaver and Germond 1994]. Sun's Solaris operating system is implemented in TSO; this mode allows the 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 8.2.0.1 (as seen in [Adve and Gharachorloo 1996]). 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.

8.2.0.2 *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 [Compaq Computer Corporation 1998]. Finally, 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.

8.2.0.3 *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". [Intel Corporation 1999].

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 8.2.0.1. 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.

8.2.0.4 *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.

8.2.0.5 *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* [Gharachorloo et al. 1990b], 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* [Keleher et al. 1992], 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 [Gao and Sarkar 2000] 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.

There have been a number of other relaxed models proposed in academic work [Adve and Hill 1990; 1993; Dubois et al. 1986b; Gharachorloo et al. 1990a; IBM 1983; May et al. 1994; Sites and Witek 1995]. The reader is advised to consult the primer mentioned above [Adve and Gharachorloo 1996] for further information.

## 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 [Kernighan and Ritchie 1988] and C++ [Stroustrup 1997] are prime examples of this: they use libraries such as POSIX threads (pthreads) [Lewis and Berg 1998] to support multi-threading.

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 [Stroustrup 1997, §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.

More recently, efforts have been made to address some of the limitations of threading definitions in C and C++ [Boehm 2005]. This work is still in progress; as of this date, it has several goals:

—First, they intend to strengthen the meaning of `volatile` in C so it provides some (but not necessarily all) of the guarantees it provides in Java.
—Second, they intend to restrict the addition of stores to the code by compilers. They do not intend to restrict them to the extent that they are restricted in Java; specifically, they do not intend to address causality issues at all.
—They intend to ensure that function-scope static variables are guaranteed to be initialized exactly once, even in multithreaded code.
—Finally, they intend to add common APIs for threading and atomic operations (such as compare-and-swap and load-linked / store-conditional operations)

8.3.2 *Multi-Threaded Languages.* Nearly every language with built-in multi-threading 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 [Gosling et al. 1996], Ada [Ada Joint Program Office 1995], Cilk [Group 2000] 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

Initially: a = 0

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

Fig. 43. CRF does not allow i == 2 and j == 1

memory model which is sometimes referred to as DAG-consistency [Blumofe et al. 1996]

8.3.2.1 *Semantics of Multithreaded Java.* Java's original memory model [Gosling et al. 1996, §17] is extremely difficult to understand; there have been several attempts to formalize it [Cenciarelli et al. 1997; 1998; Gontmakher and Schuster 1997]. Unfortunately, due to its complexity, the resulting formalisms were contradictory and difficult to understand, at best. Gontmakher and Schuster [Gontmakher and Schuster 1997; 1998] believed that simple reordering of independent statements was illegal. However, they did not consider prescient stores (as described in the original JLS [Gosling et al. 1996]). They later presented a revised model [Gontmakher and Schuster 2000].

The same authors [Gontmakher and Schuster 1997; 2000] proved that the original Java memory model required a property called *coherence* (as originally described in [Ahamad et al. 1993]). 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 [Pugh 1999].

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.

8.3.2.2 *Commit / Reconcile / Fence Approach.* Maessen, Arvind and Shen [Arvind et al. 2000] used the Commit / Reconcile / Fence protocol [Shen et al. 1999] 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 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 43, the result i = 2 and j = 1 is prohibited. This is because CRF requires an ordering over writes to

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`

Fig. 44. A Result That Must Be Permitted

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.

8.3.2.3 *Earlier Work.* Earlier work by some of the authors of this paper [Manson and Pugh 2001b; 2001a] used an operational semantics to describe a memory model for Java. Our current approach is non-operational. 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.

The Java memory model, as it stands, 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 44. 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.

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.

Another approach, SC-, was also by one of the authors of this paper [Adve 2004]. It is closer to the final model, but does not handle some of the more subtle out-of-thin-air requirements.

8.3.2.4 *Unified Memory Model approach.* Yang, Gopalakrishnan and Lindstrom [Yang et al. 2001; 2002] attempt to characterize the approaches taken in [Manson and Pugh 2001b; 2001a; Arvind et al. 2000] using a single, unified memory model (called UMM). They also use this notation to propose a third memory model [Yang et al. 2002; Yang 2004].

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 and 6; 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.

8.3.2.5 *Other.* Kotrajaras [Kotrajaras 2001] 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 [Saraswat 2004] 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 6; 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 3; in that example, given the binding of reads to writes, there are multiple fixed-point solutions.

The ECMA specification for the Common Language Infrastructure (CLI) provides a memory model [ECMA 2002b]. However, it is vague and informal; as a result it seems impossible to determine whether that model allows or disallows the non-sequentially-consistent behaviors displayed by the unsynchronized examples we have provided.

It is also worth noting that some of the Microsoft engineers have published articles [Brumme 2003] in which they claim that the CLI specification is too relaxed, and that they have written code as part of Microsoft's core libraries that won't work according to the ECMA spec.

8.3.3 *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 [Shasha and Snir 1988] 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 [Midkiff et al. 1990] 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 [Krishnamurthy and Yelick 1995]. Lee and Padua [Lee and Padua 2001] 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 require 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).

It should also be noted that if a program is executed in a sequentially consistent way, then volatile annotations become redundant. However, the volatile annotation is an important one anyway – it informs the programmer to be on the lookout for concurrent access to that variable. In addition, it has been our experience that programmers using volatile do so for performance reasons; a system that provided sequential consistency would have to be extremely well-optimized to match the efficiency of a trained programmer using the volatile annotation.

### 8.4 Final Fields

The C# language [ECMA 2002a] 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 [ECMA 2002b]; they must, as in earlier revisions of Java, be treated as normal fields for the purposes of synchronization. The C++ language has a `const` modifier, which has an equal lack of runtime semantic value.

More recently, there has been an effort to design immutability semantics for Java [Tschantz and Ernst 2005]. Focus has been on the description of the type system, but not on the concurrency semantics.

## 9. CONCLUSION

In this article, 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 article (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.

## ACKNOWLEDGMENTS

## REFERENCES


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

Adve, S. 1993. Designing memory consistency models for shared-memory multiprocessors. Ph.D. thesis, University of Wisconsin, Madison. Ph.D. Thesis.

Adve, S. and Gharachorloo, K. 1996. Shared memory consistency models: A tutorial. *IEEE Computer 29,* 12, 66–76.

Adve, S. and Hill, M. 1990. Weak ordering—A new definition. In *Proceedings of the Seventeenth International Symposium on Computer Architecture.* IEEE Computer Society Press, Seattle, WA, 2–14.

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

Adve, S. V. and Hill, M. D. June 1993. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems 4,* 6, 613–624.

Ahamad, M., Bazzi, R. A., John, R., Kohli, P., and Neiger, G. 1993. The Power of Processor Consistency. In *The Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA).*

Arvind, Maessen, J.-W., and Shen, X. 2000. Improving the Java memory model using CRF. In *Object Oriented Programming Systems, Languages and Applications* (Minneapolis, MN). 1–12.

Bacon, D. F., Strom, R. E., and Tarafdar, A. 2000. Guava: A dialect of java without data races. In *Object Oriented Programming Systems, Languages and Applications*. 382–400.

Bilardi, G. and Pingali, K. 1996. A Framework for Generalized Control Dependence. In *ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. Philadelphia, Pennsylvania, United States.

Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E., and Randall, K. H. 1996. DAG-consistent distributed shared memory. In *10th International Parallel Processing Symposium (IPPS '96)*. Honolulu, Hawaii, 132–141.

Boehm, H.-J. 2005. Threads Cannot Be Implemented as a Library. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. Chicago, IL, United States.

Brumme, C. 2003. C# memory model. http://blogs.msdn.com/cbrumme/archive/2003/05/17/51445.apx.

Cenciarelli, P., Knapp, A., Reus, B., and Wirsing, M. 1997. From Sequential to Multi-Threaded Java: An Event Based Operational Semantics. In *Sixth International Conference on Algebraic Methodology and Software Technology* (Berlin).

Cenciarelli, P., Knapp, A., Reus, B., and Wirsing, M. 1998. *Formal Syntax and Semantics of Java*. Springer-Verlag.

Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. 1999. Escape analysis for Java. In *Object Oriented Programming Systems, Languages and Applications*. 1–19.

Compaq Computer Corporation. 1998. *Alpha Architecture Handbook, version 4*. Compaq Computer Corporation, Houston, TX, USA.

Diniz, P. C. and Rinard, M. C. 1998. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *The Journal of Parallel and Distributed Computing 49,* 2, 218–244.

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

Dubois, M., Scheurich, C., and Briggs, F. A. 1986b. Memory access buffering in multiprocessors. In *Proc. 13th Ann. Intl. Symp. on Computer Architecture*. 434–442.

ECMA. 2002a. C# Language Specification. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

ECMA. 2002b. Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm.

Flanagan, C. and Freund, S. N. 2000. Type-based race detection for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. 219–232.

Gao, G. R. and Sarkar, V. 2000. Location consistency-a new memory model and cache consistency proto col. *IEEE Transactions on Computers 49,* 8, 798–813.

Gharachorloo, K. 1996. Memory consistency models for shared-memory multiprocessors. Ph.D. thesis, Stanford University.

Gharachorloo, K., Adve, S. V., Gupta, A., Hennessy, J. L., and Hill, M. D. August 1992. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing 15,* 4, 399–407.

Gharachorloo, K., Gupta, A., and Hennessy, J. 1991. Two techniques to enhance the performance of memory consistency models. In *Proc. Intl. Conf. on Parallel Processing*. I355–I364.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. May 1990a. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*. 15–26.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. L. 1990b. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture* (Seattle, WA). IEEE Computer Society Press, 15–26.

Gontmakher, A. and Schuster, A. 1997. Java consistency: Non-operational characterizations for the Java memory behavior. Tech. Rep. CS0922, Department of Computer Science, Technion. Nov.

Gontmakher, A. and Schuster, A. 1998. Characterization for Java Memory Behavior. In *Twelfth International Parallel Processing Symposium and Ninth Symposium on Parallel and Distributed Processing*. 682–686.

Gontmakher, A. and Schuster, A. 2000. Java Consistency: Nonoperational Characterizations for Java Memory Behavior. *ACM Transactions on Computer Systems 18,* 4, 333–386.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison Wesley.

Group, S. T. 2000. *Cilk 5.3.1 Reference Manual*. MIT Laboratory for Computer Science, Cambridge, Massachusetts.

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

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

Intel Corporation. 2002. *IA-64 Architecture Software Developer's Manual*. Vol. 2. Intel Corporation, Santa Clara, CA, USA.

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

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

Keleher, P., Cox, A. L., and Zwaenepoel, W. 1992. Lazy release consistency for software distributed shared memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*. IEEE Computer Society Press, 13–21.

Kernighan, B. W. and Ritchie, D. M. 1988. *The C Programming Language*, second ed. Prentice Hall.

Kotrajaras, V. 2001. Towards an improved memory model for Java. Ph.D. thesis, Department of Computing, Imperial College.

Krishnamurthy, A. and Yelick, K. 1995. Optimizing parallel programs with explicit synchronization. In *ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. La Jolla, California, 196 – 204.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21,* 7, 558–564.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers 9,* 29, 690–691.

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

Lee, J. and Padua, D. A. 2001. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*.

Lewis, B. and Berg, D. J. 1998. *Multithreaded programming with pthreads*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA.

Manson, J. 2004. The java memory model. Ph.D. thesis, University of Maryland, College Park.

Manson, J. and Pugh, W. 2001a. Core semantics of multithreaded Java. In *ACM Java Grande Conference*.

Manson, J. and Pugh, W. 2001b. Semantics of Multithreaded Java. Tech. Rep. CS-TR-4215, Dept. of Computer Science, University of Maryland, College Park. Mar.

Manson, J., Pugh, W., and Adve, S. V. 2005. The Java Memory Model. In *Proceedings of the Thirty-Second ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 378–391.

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

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

Midkiff, S. P., Padua, D. A., and Cytron, R. G. 1990. Compiling programs with user parallelism. In *Proceedings of the 3rd Workshop on Languages and Compilers for Parallel Computing*, Aug., Ed.

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

Pugh, W. 1999. Fixing the Java memory model. In *ACM Java Grande Conference*.

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

Reynolds, J. C. 1978. Syntactic Control of Interference. In *Proceedings of the Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 39–46.

Reynolds, J. C. 2004. Toward a Grainless Semantics for Shared-Variable Concurrency. In *Proceedings of the Twenty-Fourth Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag.

Ruf, E. 2000. Effective Synchronization Removal for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. Vancouver, BC Canada.

Saraswat, V. 2004. Concurrent Constraint-based Memory Machines: A framework for Java Memory Models. Tech. rep., IBM TJ Watson Research Center. Mar.

Schmidt, D. and Harrison, T. 1996. Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects. In *Third Annual Pattern Languages of Program Design Conference*.

Shasha, D. and Snir, M. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS) 10,* 2 (Apr.), 282–312.

Shen, X., Arvind, and Rudolph, L. 1999. Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture*, 150–161.

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

Stephenson, M., Babb, J., and Amarasinghe, S. 2000. Bidwidth analysis with application to silicon compilation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 108–120.

Stroustrup, B. 1997. *The C++ Programming Language*, 3rd ed. Addison-Wesley Longman, Reading Mass. USA.

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

Tschantz, M. S. and Ernst, M. D. 2005. Javari: Adding reference immutability to Java. In *Object Oriented Programming Systems, Languages and Applications*. San Diego, CA, USA.

Weaver, D. and Germond, T. 1994. *The SPARC Architecture Manual, version 9*. Prentice-Hall.

Yang, Y. 2004. Formalizing shared memory consistency models for program analysis. Ph.D. thesis, University of Utah.

Yang, Y., Gopalakrishnan, G., and Lindstrom, G. 2001. Specifying Java thread semantics using a uniform memory model. In *ACM Java Grande Conference*.

Yang, Y., Gopalakrishnan, G., and Lindstrom, G. 2002. Formalizing the Java Memory Model for Multithreaded Program Correctness and Optimization. Tech. Rep. UUCS-02-011, University of Utah. Apr.

May 2005, revised June 2005.