LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Improving the Java Memory Model Using CRF

Computation Structures Group Memo 428
October 6, 2000

Jan-Willem Maessen, Arvind, and Xiaowei Shen

jmaessen@lcs.mit.edu,arvind@lcs.mit.edu,xwshen@us.ibm.com. This paper appears in the proceedings of OOPSLA 2000. It describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by grants from Microsoft and NSF.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Improving the Java Memory Model Using CRF *

Jan-Willem Maessen
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
jmaessen@lcs.mit.edu

Arvind
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
arvind@lcs.mit.edu

Xiaowei Shen
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
xwshen@us.ibm.com

ABSTRACT

This paper describes alternative memory semantics for Java programs using an enriched version of the Commit/Reconcile/Fence (CRF) memory model [16]. It outlines a set of reasonable practices for safe multithreaded programming in Java. Our semantics allow a number of optimizations such as load reordering that are currently prohibited. Simple thread-local algebraic rules express the effects of optimizations at the source or bytecode level. The rules focus on reordering source-level operations; they yield a simple dependency analysis algorithm for Java. An instruction-by-instruction translation of Java memory operations into CRF operations captures thread interactions precisely. The fine-grained synchronization of CRF means the algebraic rules are easily derived from the translation. CRF can be mapped directly to a modern architecture, and is thus a suitable target for optimizing memory coherence during code generation.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; C.1.2 [**Processor Architectures**]: Multiprocessors; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Languages, standardization

Keywords

Memory models, Java, commit/reconcile/fence, compilation

*This paper describes research conducted at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by grants from Microsoft and NSF.

1. INTRODUCTION

Java is the first widely-accepted computer language to contain language-level support for multithreaded programs running on multiple processors. Java objects are shared between all threads; reads and writes performed in one thread must be visible in another thread. Thus, like any multithreaded language Java requires a *memory model*, which explains the behavior of objects which are read and written by multiple threads. In this paper we present new semantics for Java memory operations in terms of an enriched version of the Commit/Reconcile/Fence (CRF) memory model. These semantics are both simpler and more formal than the current Java semantics. At the same time, it retains a constructive, program-like flavor, *describing* rather than *prescribing* program behavior.

It should be emphasized that the details of the Java Memory Model are relevant to every single Java programmer. The memory model affects every object which may be visible to more than one thread during program execution. Writing Java, however, should not require a detailed understanding of the intricacies of the Java Memory Model. Instead, a set of high-level practices and guarantees should permit the programmer to write code that is both efficient and correct. We propose one possible set of rules in Section 4. We also give a set of rules to permit source-level reasoning about Java programs.

Reordering constraints are critical to understanding any memory model. There is an important difference between the reordering constraints in a language-level memory model and in an architecture-level memory model. Architecture-level memory semantics are very concrete—they describe the *dynamic* behavior of loads and stores on particular memory addresses. Instructions are reordered on the fly, and only when an instruction cannot be processed immediately. A programmer or a compiler, on the other hand, sees a *static* program where each variable may represent many different addresses dynamically. Furthermore, two different variables may both point to the same memory location (alias) during a particular program run. Optimizing compilers will reorder load operations, even when the variables referred to by those load operations may prove to alias at run time. A realistic memory model must permit such reordering.

In the CRF model, instruction ordering is resolved dynamically, and is expressed compactly by a reordering table. For Java, the constraints on the order of memory operations must be expressible in terms of program variables. This permits the programmer to systematically juggle the order of operations in a Java program to obtain a program which makes a particular order of memory operations evident. The algebraic rules we develop in Section 4.2 formalize reordering constraints for Java at the source and bytecode levels. The detailed CRF semantics make it clear that the static reorderings are correct with respect to the permitted dynamic reorderings. They also show how to efficiently map the high-level model onto a modern processor architecture.

This paper is the first detailed presentation of compilation using CRF. We must enrich CRF by adding monitor-like locks and a freeze operation to capture the write-once semantics of final fields. Our semantics for final fields are of broad interest, as many systems contain a large body of data which is written at most once—for example, object metadata in Java, or data structures in purely-functional programming languages such as Concurrent ML [14] or pH [2].

Java can be compiled efficiently by referring to the algebraic rules (for example while compiling source code to bytecode) or by using CRF operations directly (to represent memory operations during machine code generation). In either case, the compiler must reason statically about memory—where the dynamic memory semantics require that two addresses be distinct, the compiler must require that the corresponding variables cannot alias. Many of the optimizations on modern programs rely on the use of a dependency graph. A graph with fewer edges gives the compiler more flexibility in choosing instruction order, and frequently also reduces the algorithmic complexity of program analysis. The reordering tables we use yield straightforward dependency analysis algorithms for memory operations. The dynamic side conditions on reordering are captured statically by compiler analyses such as escape analysis, alias analysis, and pointer analysis.

There are a number of different declarations which affect the visibility of variables in different parts of a Java program. In this paper we address the behavior of the memory itself, and not issues of naming or scoping, so we do not distinguish between private, public, and package variables, nor between class (static) variables and instance variables. All of these obey the same basic underlying memory semantics.

In the next section of this paper, we will examine some of the shortcomings of the present Java Memory Model. This gives a brief picture of the motivation behind the present work. In Section 3 we outline some related work on memory models both for programming languages and at the architectural level. Section 4 will be of particular interest to Java programmers wishing to learn how to write correct multithreaded programs. It presents a high-level overview of our memory model, and algebraic rules which describe operation reordering. The rules in this section can be derived from the more detailed semantics in the sections following. In Section 5 we present the operational semantics of CRF itself. Section 6 gives a translation from high-level Java memory operations into lower-level CRF operations. We conclude by discussing some of the issues we do not address in detail in the semantics.

2. PROBLEMS WITH THE CURRENT JAVA MEMORY MODEL

The memory model given in Chapter 17 of the Java Language Specification [8] has problems which fall into three broad categories. First, it prohibits important compiler optimizations. Second, it is incomplete; no semantics are given for final fields, and the interaction of various kinds of storage is poorly specified. Finally, it is complex, and these complexities yield behavior that is difficult to capture or explain at the source level. The memory model we give in this paper is a response to these problems. Many potential memory semantics can be given for Java; in the course of developing this paper, we have captured several such models within CRF. Thus, we view CRF as a general tool which avoids the problems we describe in this section.

2.1 Enabling compiler optimization

The mapping from Java to CRF permits optimizations prohibited by the current Java Memory Model. The current model requires what Pugh refers to as coherence [13] (in all examples, memory cells are assumed to initially contain 0 or null, and no extraneous writes occur):

Thread 1	Thread 2
`v = p.f;`	`p.f = 2;`
`w = q.f;`	
`x = p.f;`	

What if `p == q` at run time? Then in practice we have the following code:

```
v = p.f;
w = p.f;
x = p.f;
```

Coherence requires that if we observe w=2 then x=2. The compiler is thus forbidden from performing fetch elimination and having `x = v`:

```
v = p.f;
w = q.f;
x = v;
```

Again, it's vital that a memory model be able to reason about reordering *all* operations—even multiple loads from the same memory location. The memory model we give in this paper permits loads to be re-ordered without recourse to alias analysis and as such always allows this optimization. This change reflects current practice (reported as Bug #4242244 in Sun's bug parade).

2.2 The semantics of final

The current memory model for Java does not provide memory semantics for final. The semantics of final fields are captured in the CRF translation by using the Freeze operation to declare that a field will not change. However, keeping the contents of final locations themselves consistent isn't enough. The String Problem provides a remarkable case in point. Consider the following pared-down implementation of java.lang.String:

```
public final class MyString {
  private final char[] theCharacters;
  public MyString(char[] value) {
    char [] internalValue = value.clone();
    theCharacters = internalValue;
  }
  ...
}
```

How do we guarantee that the call to `value.clone()` finishes all its writes before another thread looks at the contents of `theCharacters`? This code may print a partial copy of the string `a`, whereas it ought to print all of it or throw a null pointer exception:

Thread 1	Thread 2
`char[] characters =`	`print(a);`
`'T','a','n','g','0';`	
`a = new MyString(characters);`	

A partially-initialized string is a big security problem when strings are used for naming internal state. Thread 1 should make sure `internalValue` is up to date before `theCharacters` may be read remotely, and Thread 2 must use the updated copy of `theCharacters` when it is printing. In the current model, we must synchronize every single access to the contents of `theCharacters`. This leads to unacceptable overhead, especially given the pervasiveness of strings used for internal naming. We instead give semantics which guarantee that the contents of `theCharacters` will be up to date as of initialization.

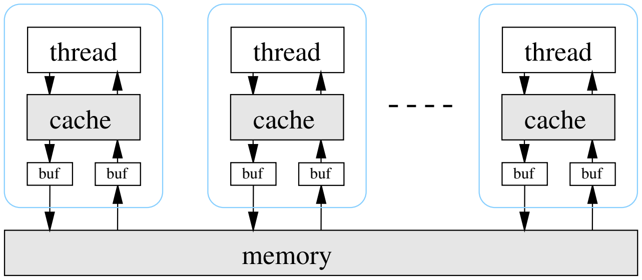

Figure 1: Memory hierarchy of the current Java Memory Model

2.3 Complexity of semantics

A more subtle and pernicious problem is that certain reorderings which might otherwise be legal cannot be expressed at the source or bytecode level in Java because they are entangled in the complexity of Java's current memory semantics. The existing Java Memory Model divides the operation of memory into three stages. The instruction stream acts on thread-local memory (cache), a main memory holds persistent state, and a buffer holds data moving between cache and main memory (see Figure 1). By placing the buffer between cache and main memory, it becomes difficult to reason about the ordering of operations at either end of the buffer. In the CRF model we instead reorder the instruction stream itself before execution. We then reason explicitly about the order in which instructions act on the cache. Every Java memory operation translates into one or more CRF instructions.

Pugh gives the following example of a seemingly innocuous addition which changes the underlying memory semantics [13]. Each left-hand side gives the original source code; the right-hand side describes a possible order in which operations are resolved in main memory. In this example, the fields `x` and `y` are distinct and therefore cannot possibly alias. In spite of this, simply adding the assignment of `u` prevents further code motion of the assignment to `v`.

```
v = q.y;          w = p.x;
w = p.x;    ⇒     p.x = 42;
p.x = 42;         v = q.y;

u = p.x;          u = p.x;
v = q.y;          w = p.x;
w = p.x;    ⇏     p.x = 42;
p.x = 42;         v = q.y;
```

The context sensitivity exhibited in this example is a surprising consequence of the axiomatic semantics (interested readers should consult Pugh's paper for the rather complex reasoning this entails). In the CRF framework decisions about instruction ordering are local (context-insensitive), and occur exclusively before instruction execution.

3. RELATED WORK

There are a large number of papers devoted to formalizing the semantics of Java and of the Java Virtual Machine; the ultimate reference on the language, however, remains the official Java Language Specification [8] and the Java Virtual Machine Specification [11]. Most efforts to formalize Java focus on the language's safety properties. There have been comparatively few attempts to formalize Java's memory model. The problems of the Java memory model are characterized succinctly by Pugh [13]. Others formalize these problems, and compare Java's properties to those of extant memory models [7].

DAG Consistency and Location Consistency are two memory models that can be applied to the semantics of multithreaded programs. DAG Consistency [3] is a memory model that is defined using the directed acyclic graph (DAG) of user-level threads that make up a parallel computation. Intuitively, each thread observes values that are consistent with some serial execution order of the DAG, but different threads can observe different serial orders. Thus, store operations performed by a thread can be observed by its successors, but threads that are incomparable in the DAG may or may not observe each other's write operations. DAG Consistency allows load operations to retrieve values that are based on different sequential executions, provided that data dependencies are respected.

Location Consistency [5] is a memory model that is defined from the compiler's point of view. It specifies a partially ordered multiset for write and synchronization operations. An ordering between two write operations is created if they are performed on the same processor, while an ordering is created between a synchronization and a write operation if the processor that performs the write operation also participates in the synchronization. For each read operation, a value set is defined as the set of legal values that can be returned.

CRF is intended as a mechanism-oriented memory model which exposes both data replication and instruction reordering at the ISA level [16]. It is intended for architects and compiler writers rather than for high-level parallel programming. The present paper is the first formal attempt to map a high-level language into the CRF model. There is a rich literature on architecture-level memory models; the interested reader is referred to our previous work [16, 15].

4. OVERVIEW OF THE MEMORY MODEL

Before we formalize the Java Memory Model using CRF, we would like to have a clear idea of how the resulting model should behave at the language level. We therefore describe a programmer-level view of the desired memory mode. We also give a simplified set of algebraic rules which can be used to reason about operation ordering in Java. This is sufficient for much of the work done by a compiler. However, the algebraic rules don't give a detailed view of what must happen at the machine level. In order to understand the actual behavior of interacting threads, and further optimize that interaction, we must resort to the lower-level constructs and more precise semantics of CRF. The algebraic rules are easily derived from the CRF semantics.

In Java, the fields of an object are referred to using the source language notation `obj.field`. Here `obj` is an object reference (a run-time value) and `field` is a field name (which is resolved to a constant during bytecode compilation). A load of a field produces a `getfield` bytecode; assignment to a field produces a `putfield` bytecode. Both of these bytecodes take the object reference and a constant field descriptor as arguments.

The Java language provides four different sorts of memory locations (note that the Java language specification refers to most memory locations as variables). Variables declared as `volatile` can be read and written in order without locking. Variables declared as `final` can be written at most once, in the object's constructor, and then read many times. Other variables fall under Java's default semantics, and we refer to these as *regular* memory locations. Finally, every Java object has an associated monitor, which can be used to enforce mutual exclusion. Java's type system guarantees that these four storage classes are completely disjoint. A field's storage class is stored along with its type in the field descriptor.

Array elements are referred to using the source language notation `arr[index]`. They are treated in the same way as regular fields.

2nd⇒	LoadR	StoreR	LoadV	StoreV	LoadF	StoreF	Enter	Exit	EndCon
1st⇓	*b*	*b*,*w*	*b*	*b*,*w*	*b*	*b*,*w*	*l*	*l*	
LoadR *a*		$a{\neq}b$		no				no	
StoreR *a*, *v*	$a{\neq}b$	$a{\neq}b$		no				no	no
LoadV *a*	no	no	no	no				no	
StoreV *a*, *v*			no	no				no	no
LoadF *a*						$a{\neq}b$			
StoreF *a*, *v*					$a{\neq}b$				no
Enter *l*	no	no	no	no			no	no	
Exit *l*							no		
EndCon		no		no					

$a{\neq}b$: Locations must never alias

Figure 2: Algebraic reordering table. Blank entries may always be reordered. Note that reordering tables are asymmetric; the first operation is found in the leftmost column, the second at the top of the table.

There are five `aload` and five `astore` bytecodes, one for references and one for each primitive type. All the array bytecodes include an array reference and an index as arguments.

In this paper we will frequently refer to the *address* of a particular field. This is *not* a language-level concept in Java! An address is a pairing of an object reference and either a field or an array index. Thus, for every address there is a clear notion of the object which contains the referenced field.

At a high level there are three kinds of loads and stores in the Java language: LoadR and StoreR for regular locations, LoadV and StoreV for volatile locations, and LoadF and StoreF for final locations. We choose this representation rather than using Java source code or byte code because we want to give rules which re-order memory operations—conceptually quite simple, but complicated to describe if those memory operations involve manipulating a stack.

Because Java uses monitor-style locking, we know that lock operations will be properly nested in the source program. Every synchronized region begins with an Enter and ends with an Exit of the corresponding monitor. We indicate monitors using an *l*. Our Enter and Exit instructions correspond directly to the `monitorenter` and `monitorexit` bytecodes in Java.

For the purposes of the memory model presented in this paper we add one additional operation, EndCon. The EndCon operation is used to indicate the end of a constructor method. As we shall see, this instruction is used to make sure the effects of object initialization are visible to every thread.

In order to share data between Java threads, it must be written to a field which is already shared by multiple threads. The class variables of all Java classes (fields notated as `static`) are always shared; in addition, new threads are created by creating an instance of a subclass of class `java.lang.Thread`, and this thread object will therefore be visible to the creating thread.

4.1 A discipline for Java programming

Any memory model for Java will dictate a programming discipline for the language. We therefore begin by outlining the programming discipline we are assuming from the Java programmer. Practices such as correct locking are already generally accepted (though widely disregarded). The other practices we propose are based on discussions on the Java Memory Model mailing list [9].

Other papers on memory semantics have attempted to give very precise requirements for programs, *e.g.* by defining a notion of *properly synchronized programs* [6, 1]. Such definitions prove remarkably slippery in practice. For example, the usual definition of properly synchronized programs refers to sequential consistency, which requires the programmer to have a thorough and deep understanding of two memory models in order to reason about programs. Furthermore, our proposed model allows programmers to write code that certainly is not properly synchronized, yet is safe: we permit certain shared fields to be accessed without synchronization, and using volatile fields without synchronization leaves correctness entirely up to the programmer.

We therefore do not attempt to describe a precise set of conditions for correct program execution. Instead, we give guidelines for coding and guarantees we can make in return. It is possible (and sometimes desirable) to write deterministic code which operates outside these restrictions, but doing so requires detailed knowledge of the memory model. Operating within these requirements will yield well-behaved code.

Regular fields

Operations on regular fields have no special ordering constraints, and thus must be used carefully.

Regular loads and stores must be correctly synchronized. Threads must synchronize on a single shared object in order to access a regular shared field safely. No regular memory operation will escape from a synchronized region. It is unsafe to access a regular field without synchronization unless that field is not shared. Our semantics will yield a range of behaviors when an improperly synchronized program is run.

Read-only objects can omit synchronization. We make a special exception to the need for synchronization for the regular fields of objects transitively reachable from a final field, and which are completely written before the reference is stored into the final field. This exception lets the programmer use information hiding to construct read-only versions of arbitrary structures which do not require locking. For example, `MyString` acts somewhat like a read-only character array (by keeping `theCharacters` private); thus it does not require further synchronization.

Constructors and final fields

Final fields allow the programmer to easily construct objects that can be read from multiple threads without locking.

An object must not escape its constructor. No constructor function may make `this` visible to another thread. A creator method should be used instead if this behavior is desired. Any writes in a constructor will occur before other threads can see the object constructed. Each constructor method is separately guarded, so if a subclass allows `this` to escape the superclass constructor will still behave in the expected way.

Final fields may be read without synchronization. It will be safe to access these locations without locking if the previous rule is obeyed. We ensure the data being read is current before the first final load from a particular address in a given thread.

Volatile fields

There are few restrictions on volatile fields, and they can be used by programmers to implement very free-form data exchange. Thus, using volatile storage is always comparatively dangerous.

Any field may be made volatile without affecting program behavior. If the program was previously correct, changing the storage class of a variable to volatile does not change the behavior of the program. Thus, volatile fields obey the constraints of both regular and final fields.

Volatile loads and stores are sequentially consistent [10]. They may not be reordered in any way with respect to one another.

Volatile stores fence previous memory operations. Memory operations which occur earlier in program order will be performed before a volatile store.

Volatile loads fence subsequent memory operations. Memory operations which occur later in program order will be performed after a volatile load.

4.2 Algebraic rules

We present the model by first giving algebraic rules for Java memory operations, and then translating these operations into an enriched version of CRF. The algebraic rules will be derivable from the translation. Our goal is to permit the greatest possible number of behaviors while preserving the programming discipline described above.

Because reordering constraints are so vital to understanding any memory model, we begin by examining the reordering constraints for the Java Memory Model given in Figure 2. We can reorder instructions according to the table without resorting to complex reasoning about the CRF operations from which they are derived (Pugh's example, given in Section 2, shows such a high-level approach is not possible in the current Java Memory Model). A lot of simple reasoning can happen at the algebraic level; in many cases we can get a picture of how threads will interact simply by reordering the instructions of threads according to the table, then seeing how those instructions interleave.

The condition $a{\neq}b$ captures the usual constraints between operations on a single memory address. Note that loads on the same address may be reordered freely. The "no" constraints for volatile memory operations capture the sequentially consistent behavior and fence semantics of volatile memory (the operations must occur in program order). The "no" constraints between memory operations and lock operations prevent memory operations from escaping a lock region. Note that final fields should only be written before the object is visible to other threads; consequently, lock regions and volatile memory operations do not affect the ordering of final memory operations. Note also that we allow arbitrary loads and stores to move *into* a lock region.

4.3 Eliminating operations

We can *eliminate* certain operations if doing so is consistent with some execution of a thread. This will reduce the possible program behaviors, but will not introduce any new behavior. Thus, the correctness of operation elimination at this high level depends on showing similar eliminations can be performed within CRF itself. We may eliminate any load operation if our reordering rules allow it to be preceded immediately by a load or store to the same address. Similarly, we may eliminate any store which can be reordered so that it is immediately followed by a second store to the same address. For final loads, we allow fetch elimination if a load or store to the same address occurs *anywhere* earlier in the program order.

4.4 Compilation using algebraic rules

A static, purely local notion of how memory operations behave *within* a thread is indispensable for generating efficient code without resorting to complex inter-thread analyses. Optimizations involving code motion require the construction of a dependency graph. Our goal is to create a dependency graph with as few edges as possible, giving the compiler the greatest possible leeway to reorder instructions.

Constructing a dependency graph using the reordering table is simple. For every pair of operations A and B, where A occurs before B in the program, we see if our reordering table permits $A;B$ to be reordered to $B;A$. If it does, no dependency exists between the instructions. If they cannot be reordered, we add a dependency edge from A to B. The resulting graph is unique modulo transitivity. In practice, of course, we can be more careful and avoid adding transitive edges to the graph (these edges increase algorithmic complexity without changing results).

While this dependency graph must be conservative, since not all information is known at compile time, our side conditions correspond to well-known compiler analyses:

Type information and alias analysis can approximate the condition $a{\neq}b$. Distinct fields and disjoint types will never share an address. Alias analysis can show that many remaining addresses are certainly different.

Points-to analyses can determine which loads and stores might potentially refer to or be referred to by a particular final field, approximating the condition $a \notin B$ which we will encounter in the CRF semantics. Note that type information can be used to give a conservative approximation to points-to analysis.

Inter-thread escape analysis [4, 19] will reveal when an object is used entirely within one thread. In this case, we can eliminate coherency operations on the object. Local loads and stores can therefore be reordered freely. Only the constraint $a{\neq}b$ holds for purely local fields.

4.5 Other constraints on reordering

We have noted already that a compiler must reason about program addresses symbolically, whereas a running JVM can work with actual addresses. In addition, however, a compiler must work with unresolved control flow—conditionals whose outcome cannot be determined at compile time. We do not consider control flow to constrain instruction reordering, except that the results of reordering must be consistent with an execution of the original program (for example, data dependency must be respected).

Consider the following example:

Thread 1	Thread 2

`if (p.next == p) {`
`  List tmp = p.next;`
`  system.out.print("y");`
`  if (tmp.next == null) {`
`    system.out.print("es");`
`  }`
`}`
	`p.next = p;`

The data dependency between `tmp` and `tmp.next` ensures that the latter read must occur after the former read. It is therefore impossible for the program to print "yes". It may, however, print "y" and throw a null pointer exception—the initialization of `tmp` can occur speculatively before we check `p.next`. Memory models based on Location Consistency allow this program to print "yes", permitting the following "optimization":

Thread 1	Thread 2
`List tmp = p.next;`	`p.next = p;`

```
List tmp = p.next;
// lifted out of conditional
if (p.next == p) {
  system.out.print("y");
  if (tmp == null) {
    system.out.print("es");
  }
}
```

We observed that since `p.next` equals `p` and `tmp` equals `p.next` we ought to be able to replace `tmp.next` by `null`. This reasoning works properly only if we are *consistent* about replacing `p.next` by `tmp`. The ability to express behaviors like this one is widely regarded as an advantage of permissive memory models. However, we have yet to encounter a situation in which such a rewriting was applied consistently and whose behavior could not be explained by our model.

4.6 The algebraic rules are not enough

Most modern architectures have very flexible memory models. This has a few consequences for the Java Memory Model. First, our algebraic rules say very little about how multiple threads interact with one another during program execution. In particular, a different instruction order may be chosen each time particular code is executed. Second, the architecture makes dynamic choices about instruction order based on the actual value of memory addresses—something we only approximate when we statically reorder program operations. Thus, we need a clear picture of how the memory model can be mapped to an actual machine with multiple processors and caches. If Java is to be efficient, this mapping must be clean and inexpensive. This is challenging, as enforcing ordering constraints among memory operations requires the use of coarse-grained architecture-level memory synchronization operations. An efficient realization of the Java Memory Model on any modern architecture will require a code generator which can coalesce memory barriers.

We give an instruction-by-instruction translation from the Java Memory Model to CRF in order to fulfill all these requirements. CRF has clear operational semantics to capture the interaction of multiple threads. The reordering table for enriched CRF allows us to easily prove that our translation matches the algebraic reordering table. CRF is easily and efficiently mapped to modern architectures. Finally, CRF provides a fine-grained model of memory synchronization; we avoid introducing spurious synchronization constraints into our programs. Thus, a machine code compiler can use CRF to merge fine-grained synchronization operations which are then mapped to a single, coarse architecture-level memory synchronization instruction.

5. THE CRF MODEL

CRF is intended for both architects and compiler writers. It is defined by giving precise semantics to memory instructions so that every CRF program has a well-defined dynamic behavior. This behavior is specified in terms of concrete instructions and addresses using operational semantics. CRF memory operations act on a local semantic cache (sache). We view our Java Virtual Machine as a collection of threads, each with its own cache. A thread's cache is initially a copy of the cache of the thread which started it. The contents of the cache can be moved directly to or from main

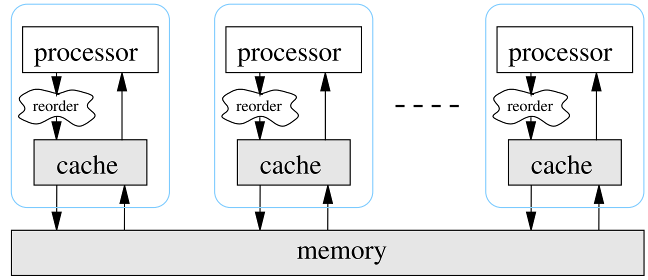

Figure 3: Structure of memory hierarchy in CRF memory model

memory. There is no buffer separating cache and memory; instead, our semantics move data atomically between them. Cache-to-cache data movement is possible only via main memory. Thus, the CRF memory architecture is much simpler than the one used in Java (Figure 3). In a real machine, we have a fixed number of processors each with a distinct cache. An implementation must provide a thread scheduler which performs the necessary operations to map the thread-centered CRF model used in this paper onto the processors of a concrete machine.

In CRF, we decompose conventional Load and Store instructions into finer grain instructions: a load becomes a Loadl (load-local) preceded by a Reconcile, and a Store becomes a Storel (store-local) followed by a Commit. The Commit and Reconcile instructions ensure that the data produced by one thread can be observed by another thread whenever necessary. This fine-grained control over memory consistency will allow us to develop a simple, yet precise, memory semantics for Java.

We formalize the CRF model by imagining the code as a sequence of pending memory operations, each labeled with a unique result tag r. We separate two instructions with a semicolon, which is associative (we can also glue groups of instructions together with a semicolon):

r_1 = Storel a, 5;
r_2 = Loadl a;

When an instruction is executed, we associate its result with the result tag in a completion map. This result is the value loaded in case of a load, and otherwise is a tick $\surd$ indicating that the instruction has completed execution. Each thread consists of a sequence of pending instructions, a set of completed instructions, and a cache which maps addresses to values tagged with a state of either Clean, Dirty, Locked, or Frozen.

5.1 Rewriting semantics

We give the basic rules for CRF in Figure 4. Note the notation: The first line describes a state where the next instruction is a Storel of value v to address a and the cache contains a mapping for a whose cache state is s (and whose value is irrelevant, since it is discarded). If this is the case we can write the result into the cache, changing the value stored to v and marking the entry as dirty. The Storel is marked as done using a tick ($\surd$). We separate completed instructions with a slash (/)—their exact arrangement doesn't matter, and we consider the simplest order we can.

Loads and stores are local, acting purely on the cache. In order to implement cache coherence, we must have *background rules* to move values to and from main memory and between threads. The background rules can be used at any time by any thread. The simplest of these is a local rule to flush clean data from a cache.

Local Rules

$$\begin{array}{lll} (r = \text{Storel } a,v; & \textit{instructions}, \quad \textit{completions}, \textit{cache}[a := -,-]) & \Rightarrow \\ (& \textit{instructions}, r = \surd \,/\, \textit{completions}, \textit{cache}[a := v, \text{Dirty}]) & \end{array}$$

$$\begin{array}{lll} (r = \text{Loadl } a; & \textit{instructions}, \quad \textit{completions}, \textit{cache}[a := v,s]) & \Rightarrow \\ (& \textit{instructions}, \ r = v \,/\, \textit{completions}, \textit{cache}[a := v,s]) & \end{array}$$

$$\begin{array}{lll} (r = \text{Commit } a; & \textit{instructions}, \quad \textit{completions}, \textit{cache}) & \\ & \quad \text{where } a \text{ is not in } \textit{cache}\text{, or } a \text{ is Clean} & \Rightarrow \\ (& \textit{instructions}, r = \surd \,/\, \textit{completions}, \textit{cache}) & \end{array}$$

$$\begin{array}{lll} (r = \text{Reconcile } a; & \textit{instructions}, \quad \textit{completions}, \textit{cache}) & \\ & \quad \text{where } a \text{ is not in } \textit{cache}\text{, or } a \text{ is Dirty or Frozen} & \Rightarrow \\ (& \textit{instructions}, r = \surd \,/\, \textit{completions}, \textit{cache}) & \end{array}$$

$$\begin{array}{lll} (r = \text{Fence } a,b; & \textit{instructions}, \quad \textit{completions}, \textit{cache}) & \Rightarrow \\ (& \textit{instructions}, r = \surd \,/\, \textit{completions}, \textit{cache}) & \end{array}$$

$$\begin{array}{lll} (r = \text{Freeze } a; & \textit{instructions}, \quad \textit{completions}, \textit{cache}[a := v, \text{Clean}]) & \Rightarrow \\ (& \textit{instructions}, r = \surd \,/\, \textit{completions}, \textit{cache}[a := v, \text{Frozen}]) & \end{array}$$

$$\begin{array}{lll} (r = \text{Freeze } a; & \textit{instructions}, \quad \textit{completions}, \textit{cache}) & \Rightarrow \\ (& \textit{instructions}, r = \surd \,/\, \textit{completions}, \textit{cache}) & \end{array}$$

Background Rules

$$\begin{array}{ll} (\textit{instructions}, \textit{completions}, \textit{cache}[a := v,s]) & \\ \quad \text{where } s \text{ is Clean or Frozen} & \Rightarrow \\ (\textit{instructions}, \textit{completions}, \textit{cache}) & \end{array}$$

$$\begin{array}{ll} (\textit{instructions}, \textit{completions}, \textit{cache}) \,/\, \textit{threads}, \textit{memory}[a := v] & \\ \quad \text{where } \textit{cache} \text{ contains no mapping for } a & \Rightarrow \\ (\textit{instructions}, \textit{completions}, \textit{cache}\,[a := v, \text{Clean}]) \,/\, \textit{threads}, \textit{memory}[a := v] & \end{array}$$

$$\begin{array}{ll} (\textit{instructions}, \textit{completions}, \textit{cache}\,[a := v, \text{Dirty}]) \,/\, \textit{threads}, \textit{memory}[a := -] & \Rightarrow \\ (\textit{instructions}, \textit{completions}, \textit{cache}\,[a := v, \text{Clean}]) \,/\, \textit{threads}, \textit{memory}[a := v] & \end{array}$$

Figure 4: Rewrite rules for basic CRF

Local Rules

$$\begin{array}{lll} (r = \text{Lock } l; & \textit{instructions}, \quad \textit{completions}, \textit{cache}[l := n, \text{Locked}]) & \Rightarrow \\ (& \textit{instructions}, r = n \,/\, \textit{completions}, \textit{cache}[l := n+1, \text{Locked}]) & \end{array}$$

$$\begin{array}{lll} (r = \text{Unlock } l; & \textit{instructions}, \quad \textit{completions}, \textit{cache}[l := n+1, \text{Locked}]) & \Rightarrow \\ (& \textit{instructions}, r = n \,/\, \textit{completions}, \textit{cache}[l := n, \text{Locked}]) & \end{array}$$

Background Rules

$$\begin{array}{ll} (\textit{instructions}, \textit{completions}, \textit{cache}) \,/\, \textit{threads}, \textit{memory}[l := 0] & \Rightarrow \\ (\textit{instructions}, \textit{completions}, \textit{cache}\,[l := 0, \text{Locked}]) \,/\, \textit{threads}, \textit{memory} & \end{array}$$

$$\begin{array}{ll} (\textit{instructions}, \textit{completions}, \textit{cache}\,[l := 0, \text{Locked}]) \,/\, \textit{threads}, \textit{memory} & \Rightarrow \\ (\textit{instructions}, \textit{completions}, \textit{cache}) \,/\, \textit{threads}, \textit{memory}[l := 0] & \end{array}$$

Figure 5: Additional rewrite rules for CRF augmented with locking

2nd ⇒ 1st ⇓	Loadl b	Storel b	Lock b	Unlock b	Freeze b	Commit b	Reconcile b	$\text{Fence}_{r\alpha}$ $B, -$	$\text{Fence}_{w\alpha}$ $B, -$
Loadl a		$a \neq b$						$a \notin B$	
Storel a	$a \neq b$	$a \neq b$				$a \neq b$			
Lock a				$a \neq b$				$a \notin B$	$a \notin B$
Unlock a								$a \notin B$	$a \notin B$
Freeze a									
Commit a					$a \neq b$				$a \notin B$
Reconcile a	$a \neq b$				$a \neq b$				
$\text{Fence}_{\alpha r}$ $-, A$			$b \notin A$	$b \notin A$			$b \notin A$		
$\text{Fence}_{\alpha w}$ $-, A$		$b \notin A$	$b \notin A$	$b \notin A$					

$a \neq b$: Addresses must not match
$a \notin B$: Address of 1st instr must not be contained in pre-address of fence
$b \notin A$: Address of 2nd instr must not be contained in post-address of fence
α: Either r (read) or w (write)

Figure 6: Reordering table for extended CRF (Blank entries may always be reordered).

The remaining rules move data between a thread and main memory, and thus act on the state of the entire system (which includes the remaining threads, though they are not affected by the operations). A thread can at any time cache an entry contained in global memory if it has not done so already, or can write dirty data back to main memory. Again, instruction execution happens purely locally within a thread—the state of main memory is required only for background operations.

Thus, we can take the pair of load and store defined earlier and execute them locally, on a single processor:

$(r_1 = \text{Storel } a, 5;\ r_2 = \text{Loadl } a;\ \ instrs, \quad c, \quad cache[a := 0, \text{Clean}])$
$\Rightarrow$
$(r_2 = \text{Loadl } a;\ \ instrs, \quad r_1 = \surd \,/\, c, \quad cache[a := 5, \text{Dirty}])$
$\Rightarrow$
$(\ instrs, \quad r_2 = 5 \,/\, r_1 = \surd \,/\, c, \quad cache[a := 5, \text{Dirty}])$

Thus, we remove instructions one at a time from the instruction stream, and keep track of their results and of the contents of the cache.

To ensure that background operations complete, we use the thread-level operations Commit and Reconcile. Commit ensures the results of a write to a particular location reach main memory. Reconcile ensures that the cache is refreshed with new data for a particular location before it is read. These behaviors are reflected in different reordering rules for Commit and Reconcile. Note that these instructions simply block and wait for the background rules to do the real work of moving data between cache and memory. Picking up from the previous example, the next instruction might be a Commit, in which case we must wait for a to be written back to memory:

$(r = \text{Commit } a;\ \ instrs, \quad c, \quad cache[a := 5, \text{Dirty}])\ /$
$threads, \quad memory[a{:=}\,0]$
$\Rightarrow$
$(r = \text{Commit } a;\ \ instrs, \quad c, \quad cache[a := 5, \text{Clean}])\ /$
$threads, \quad memory[a{:=}\,5]$
$\Rightarrow$
$(\ instrs, \quad r = \surd \,/\, c, \quad cache[a := 5, \text{Clean}])\ /$
$threads, \quad memory[a{:=}\,5]$

Real systems use much more sophisticated cache coherence protocols, and in practice these operations will not oblige a processor either to suspend or to flush entries from its cache (as we shall see in Section 5.5, Commit and Reconcile are no-ops on most modern architectures).

For final fields we will need some way to declare that a location's value should remain invariant. We thus add a Frozen cache state, which declares that a memory location can remain in the cache indefinitely. A Freeze operation optionally marks a cache line as Frozen (it may instead be ignored; this decision is non-deterministic). Frozen cache lines need not be subjected to background operations.

5.2 Augmented CRF for Java

We can imagine performing locking in CRF using some variation on Dekker's algorithm. In practice, actual architectures provide more efficient atomic memory primitives. Augmenting CRF with monitor-style synchronization makes our semantics simpler and clearer. These augmentations are described in Figure 5. Lock and Unlock are the only *atomic* memory operations, and as such require Locked access to their address (and corresponding background operations).

Note that in our semantics a Locked location is *removed* from main memory; in practice we would use lower-level atomic memory primitives to check for a distinguished value. Monitors can be locked recursively; thus locks contain a count of the number of times the owning thread has locked the lock. Note also that locking does nothing to enforce mutual exclusion; it just manipulates a lock. We rely on explicit Fence operations (described in the next section) to define synchronization regions. Finally, Java associates a unique lock location with each object in the system; because these lock locations are not regular fields, they will only be accessed using Lock and Unlock. We therefore give no semantics for reading and writing locked locations.

5.3 Reordering

Much of the power of the CRF model comes from the ability to reorder instructions. For our purposes this reordering captures a mix of both compiler optimizations and architectural speculation and reordering. In practice, the greatest constraint on re-ordering is data dependence: operation ordering must respect data dependencies. We make use of data dependence constraints in our semantics. Fine-grained Fence instructions enforce ordering between operations which are not data-dependent.

Each memory fence has a pair of arguments, a pre-address and a post-address, and imposes an ordering constraint between memory operations involving the pre- and post- addresses. For example, Fence_{rw} a, b prevents preceding reads on address a from being reordered with respect to following writes on address b. Note that a Fence_{wr} instruction imposes ordering constraints on preceding Commit instructions instead of Storel instructions; it makes little sense to ensure a Storel is completed if it is not followed by a Commit. Similarly, a Fence_{wr} instruction imposes ordering constraints on following Reconcile instructions instead of Loadl instructions.

The Fence instruction itself doesn't affect memory state when it executes; it merely constrains the order of other operations. Reordering is specified by referring to the reordering table in Figure 6. Memory access instructions can be reordered if they access different addresses or if they are both Loadl instructions. Memory rendezvous instructions (Commit and Reconcile) can always be swapped. Memory fence instructions can always be swapped. We also allow locking operations to be reordered freely, except that we can use Fence instructions on lock locations to enforce the ordering constraints required to prevent deadlock. For this purpose, we consider Lock and Unlock to each be both a read and a write. The underlying rationale is to allow maximum reordering flexibility for both compiler and architecture.

5.4 Stronger coherence

When mapping CRF to a processor it is vital to realize that a CRF program dictates the minimum required synchronization. It is always legal to insert extra Commit, Reconcile, and Fence operations into the program. Such excess operations simply enforce stronger coherence and eliminate opportunities for reordering—they never prohibit our programs from executing correctly. It is this observation which safely allows us to replace fine-grained CRF operations with coarse-grained processor-level operations.

5.5 Machine Mapping

The CRF model isolates compiler writers from the plethora of microprocessors by providing an implementation-independent representation of memory coherency which is more general than any particular implementation. For modern microprocessors, the Loadl and Storel can be translated into normal Load and Store instructions. The load and store instructions of most current machines implicitly enforce cache coherence, so all Reconcile and Commit operations can simply be eliminated. Similarly, Freeze operations exist purely for semantic reasons and can be eliminated when a program is run.

The coarse-grained memory fence provided by most modern architectures is both a blessing and a curse: a blessing because it can encompass many finer-grained operations in one, a curse because it is usually quite expensive. To run CRF programs on a sequentially consistent machine, we eliminate all fences since memory accesses are executed strictly in-order. To run CRF programs on a machine that supports Sparc's RMO model, we need to translate fences into appropriate memory barrier (Membar) instructions [18]. Any Java compiler will need to coalesce memory barriers before translation to machine code. One simple way to do this is to add dependencies between all barriers such that there is a single ordering to the barriers which respects the original dependencies. A dependency-based code generator can then do the actual coalescing. If our initial dependence graph has fewer edges, there will be greater opportunity to choose a good ordering for the barriers.

5.6 Non-atomic Memory

In the CRF model, all operations are *atomic*—that is, they happen in a single step. There is a single global memory, and all cache updates to that memory occur (conceptually) in some particular order. Some characteristics of memory semantics can be captured using either instruction reordering or non-atomic store operations. This can be illustrated by the following example:

Thread 1	Thread 2	Thread 3	Thread 4
Store a,1;	Store a,2;	v_1 = Load a;	w_1 = Load a;
		v_2 = Load a;	w_2 = Load a;

In this example, we might want to allow $v_1 = 1$, $v_2 = 2$, $w_1 = 2$, and $w_2 = 1$. Such a property allows the two load instructions (in thread 3 or 4) to be reordered statically, when it may be unclear if their addresses differ. We use explicit instruction reordering to model this; load instructions can always be reordered, even when they access the same address.

The same semantics can be modeled by requiring that memory instructions be executed in-order but requiring that stores occur non-atomically. Compiler-oriented memory models such as Location Consistency and DAG Consistency use this approach, as do many processor architecture specifications such as Alpha [17] and PowerPC [12]. The prescient stores in the original Java Memory Model effectively use this strategy, allowing a store *action* to be reordered with respect to a store *instruction*. We feel that atomic stores are easy to understand and that operation reordering is natural to programmers.

Some of the behaviors allowed by non-atomic stores cannot be captured by the instruction reordering in CRF. We already saw one such example in Section 4.5. Consider another example:

Thread 1	Thread 2
Store a,1;	Store a,2;
v = Load a;	w = Load a;

None of the memory operations can be reordered, yet memory models such as Location Consistency permit v to be 2 while w is 1.

We can capture this behavior formally by extending our memory model (for example by using Generalized CRF [15]). The translation from Java would remain unchanged; only programs which violate our memory discipline will exhibit new behavior. However, there is no clear advantage to the compiler writer or the programmer. Indeed, we know of no actual processor which exhibits the behavior described above; it may simply be an artifact of choosing to describe stores non-atomically in the memory model. Non-atomic store operations complicate the memory model to a point where it defies comprehension, yet provide no additional power to the programmer; we therefore advocate CRF as a simpler alternative.

6. TRANSLATING JAVA INTO CRF

The CRF semantics we have described so far reason about concrete operations on actual machine addresses. A compiler does not have this luxury, and must instead manipulate abstract references which may potentially overlap. Addresses in the reordering table are now variable references rather than memory addresses. As in the algebraic rules, we approximate the side conditions on reordering using conventional compiler analyses—primarily type information and alias information. It is therefore straightforward to reason about CRF operations statically.

Each of the Java memory operations has a straightforward static translation into CRF. The translation is designed to be used by a compiler, or by a Java programmer reasoning about program behavior. For clarity, we introduce versions of the Commit, Reconcile, and Fence instructions which act on multiple addresses at once. The notation $*_V$ indicates all volatile locations; $*_{VR}$ indicates all volatile and regular locations; $*_{VRL}$ indicates volatile, regular, and lock locations; $*$ indicates all memory locations. For example:

Fence_{rw} $*_V$, a;	Complete last volatile Loadl before allowing Storel to a
Fence_{ww} $*$, $*_{VR}$;	Wait for all previous writes to complete before further volatile or regular writes.

Operation	Regular	Final	Volatile
Store a,v;	Storel a,v; Commit a;	Storel a,v; Commit a; Freeze a;	Fence_{rw} $*_{\text{VR}}$, a; Fence_{ww} $*_{\text{VR}}$, a; Storel a,v; Commit a;
v = Load a;	Reconcile a; v = Loadl a;	Reconcile a; v = Loadl a; Freeze a;	Fence_{wr} $*_{\text{V}}$, a; Reconcile a; v = Loadl a; Fence_{rr} a, $*_{\text{VR}}$; Fence_{rw} a, $*_{\text{R}}$;

Figure 7: CRF translations for Java loads and stores

Operation	Translation
Enter l;	Fence_{ww} $*_{\text{L}}$, l; Lock l; Fence_{wr} l, $*_{\text{VR}}$; Fence_{ww} l, $*_{\text{VRL}}$;
Exit l;	Fence_{ww} $*_{\text{VR}}$, l; Fence_{rw} $*_{\text{VR}}$, l; Unlock l;
EndCon;	Fence_{ww} $*$, $*_{\text{VR}}$;

Figure 8: CRF translations for Java synchronization operations

A precise translation of these instructions into dynamic CRF would require either precise type information for each memory location, or a giant loop over all of memory. As noted above, however, a compiler is free to choose a less permissive translation of these operations (for example, by substituting global memory fences for finer-grained Fence operations). We expect compilers to exploit our reordering rules so that many fences may be consolidated.

6.1 Regular memory

CRF's fine-grained memory operations nicely express the semantics of regular memory locations, which ordinarily require locking for coherence (first column of Figure 7). We Fence both volatile and regular locations during locking (Figure 8) so that we can safely change a regular location into a volatile location without changing the behavior of a program.

Note that our semantics permit any load or store operation to be moved *into* a lock region. Thus, a lock region simply prevents memory operations inside the region from being moved outside. We synchronize on all lock locations before we enter a lock region; this guarantees that previous exit operations complete before entry. This condition is necessary to prevent the memory model from introducing deadlocks.

6.2 Final fields

The semantics of final fields (second column of Figure 7) require explanation. We must ensure every final field is up to date when a constructor completes. This is the purpose of the EndCon operation, which is the only operation in our semantics that affects the ordering of operations on final fields. Objects pointed to by a final field are up to date if they've been fully initialized by the end of the constructor, since the fence in EndCon ensures those initializing writes are complete as well. When we follow a reference from a final field, we cannot Reconcile until we have fetched the object reference from the final field. Consequently data dependence is sufficient to enforce the correct semantics for final references.

If we did not ensure final writes were completed in the constructor, we would need to make sure they were complete before a reference to the containing object escaped. We would then need to translate Store as follows (with a concomitant change for StoreF):

$$\text{Store } a,p; \quad \equiv \quad \begin{array}{l} \text{Fence}_{\text{ww}}\ \vec{p}\ , a; \\ \text{Storel } a,p; \\ \text{Commit } a; \end{array}$$

Here the notation $\vec{p}$ indicates all memory locations in the object referred to by pointer p. This fence would *only* affect dependent writes, but it requires a Fence before every Store.

Note the effect of Freeze a when p_1 = LoadF a is followed by p_2 = LoadF a and we move the Freeze a as far down in instruction order as possible, eliminating Reconcile a:

$$\begin{array}{l} \text{Reconcile } a; \\ p_1 = \text{Loadl } a; \\ \text{Freeze } a; \\ \text{Reconcile } a; \\ p_2 = \text{Loadl } a; \\ \text{Freeze } a; \end{array} \equiv \begin{array}{l} \text{Reconcile } a; \\ p_1 = \text{Loadl } a; \\ p_2 = \text{Loadl } a; \\ \text{Freeze } a; \\ \text{Freeze } a; \end{array} \equiv \begin{array}{l} \text{Reconcile } a; \\ p_1 = \text{Loadl } a; \\ \text{Freeze } a; \\ p_2 = p_1; \end{array}$$

Replacing p_2 = Loadl a by $p_2 = p_1$ in the above yields a legal behavior, and thus LoadF operations can be combined. The remaining redundant operations then combine trivially. Similarly, when we follow StoreF a, p_1 by p_2 = LoadF a:

$$\begin{array}{l} \text{Storel } a,p_1; \\ \text{Commit } a; \\ \text{Freeze } a; \\ \text{Reconcile } a; \\ p_2 = \text{Loadl } a; \\ \text{Freeze } a; \end{array} \equiv \begin{array}{l} \text{Storel } a,p_1; \\ p_2 = \text{Loadl } a; \\ \text{Commit } a; \\ \text{Freeze } a; \\ \text{Freeze } a; \end{array} \equiv \begin{array}{l} \text{Storel } a,p_1; \\ \text{Commit } a; \\ \text{Freeze } a; \\ p_2 = p_1; \end{array}$$

Again, we know that we can fetch-eliminate p_2 = Loadl a. Once again we have fetch-eliminated the LoadF.

6.3 Volatile memory

Operations on volatile fields (third column of Figure 7) cannot be reordered at all. This is quite easy to capture by adding fences before every volatile memory operation separating it from other volatile operations. Every volatile read must be followed by two fences to enforce acquire semantics; fences before every volatile write enforce release semantics. Note that as with locking these fences do not affect final locations. The guarantees given by EndCon are sufficient for this purpose, as long as we make sure an object reference does not escape from the constructor. Because EndCon fences volatile writes as well as final writes, we can correctly replace final fields with volatile fields.

6.4 Synchronization

The translations for Enter and Exit (Figure 8) must provide the necessary synchronization to enforce the ordering constraints required by monitors. A pair of fence operations after the Lock in Enter and before the Unlock in Exit prevent loads and stores from escaping the synchronized region. However, we must still ensure that Exit operations can be reordered, but that they are not reordered with respect to Enter operations. As a result, we do not fence lock locations in the translation for Exit, but we fence them both before and after every Enter operation (thus preventing Exit from being reordered with respect to Enter in either direction). As a matter of convention, we use only a write fence to constrain the ordering of lock locations in the translation.

The translation of EndCon is quite straightforward—it simply constrains preceding writes to occur before succeeding non-final writes. This results in a single Fence instruction as shown in Figure 8.

6.5 Allowing lock elimination

The translation in the previous section was simplified to make the presentation clearer. It does not allow for the elimination of synchronization when the data involved is local to one processor—even if we eliminate the Lock and Unlock operations, we must simulate the effect of the Fence instructions which accompany them. Experiments show that static elimination of useless locking can reduce synchronization by approximately 50% [4].

We *can* eliminate useless lock operations when they are nested with more useful ones. This is because we may *enlarge* a lock region as long as we don't change the nesting of locks. Consider this example:

```
...
synchronized(o) {
  o.m1();
  synchronized(p) {
    p.m2(o);
  }
  o.m3();
}
...
```

If neither `m1` nor `m3` performs synchronization, we can actually enlarge the region *p* to overlap with region *o*:

```
...
synchronized(o) {
  synchronized(p) {
    o.m1();
    p.m2(o);
    o.m3();
  }
}
...
```

In practice we usually want to make synchronization regions as small as possible, but consider the effect of this transformation if `p` is not shared: we perform the same Fence operations in quick succession on entry and exit, and we will not be able to tell if `p` is actually locked or not. Thus, we may erase `synchronized(p)` with no effect.

We are therefore really worried about eliminating useless synchronization that is *not* nicely nested. Imagine none of the method calls in the following code contains synchronization:

```
...
synchronized(o) {
  o.m1();
}
synchronized(p) {
  p.m2();
}
p.m3();
...
```

In this case, if `p` is not shared we must still fence the synchronization region for `p`. This means the call to `m2` may not be moved.

It is not hard to modify our translation to permit further elimination of redundant locks. There is no need for a Fence unless new information is being obtained—that is, unless some other thread has synchronized on the object and modified it in some way. We imagine that the Lock operation returns the identity of the previous thread to hold the lock. We fence only when some other thread has used the lock. The operation Enter *l* is therefore translated as shown in Figure 9.

Operation	Translation

Enter *l*;	$\text{Fence}_{\text{ww}}\ *_{\text{L}}, l$;
	r = Lock l;
	if (r != *currentThread*) {
	  $\text{Fence}_{\text{wr}}\ l, *_{\text{VR}}$;
	  $\text{Fence}_{\text{ww}}\ l, *_{\text{VRL}}$;
	}

Figure 9: Improved CRF translation for Enter

Now we eliminate lock operations by "drifting down". Operations escape from the *top* of the synchronization region, where the Fence operations are skipped:

```
...
synchronized(o) {
  o.m1();
}
p.m2();
p.m3();
synchronized(p) {
}
...
```

There is one asymmetry here: the eliminated synchronization regions will get stuck when they reach a synchronization region that actually exists:

```
...
p.m2();
p.m3();
synchronized(p) {
}
synchronized(o) {
  o.m1();
}
...
```

This prevents prior code (in this case the invocations of `m2` and `m3`) from being moved into the synchronization region for no particularly good reason. This restriction is necessary because our new translation gives equal treatment to locks which are *never* shared and locks which simply are not shared *yet*. If `p` later becomes shared, such code motion would lead to incorrectness or deadlock. This is a shortcoming of the dynamic nature of our semantics.

7. CONCLUSION

We have simplified our presentation of the Java Memory Model. Other translations yielding the same model as described here are possible—for example, by doing pairwise instruction translation we can use single-address Fence operations in our translation of LoadV and StoreV. Moreover, the memory model we present is not the only possible memory model for Java. We have used the CRF framework to explore a series of different proposals for a Java Memory Model in order to assess their effects on compilation. More permissive memory models exist which use data-dependent synchronization; they are, however, arguably more difficult for the programmer to understand.

Java's strongly-ordered exception model has a major influence on compiled code; in many cases null pointer checks will require otherwise unconstrained memory operations to be ordered. Our choice of memory model does not affect this decision. We must simply respect these additional constraints on operation order.

Object finalization is a sticky memory model issue. A finalizer method is run on an object when that object is no longer reachable from any thread. There is no guarantee that any particular thread will run the finalizer; as a result, memory may look quite different to the finalizing thread than it did to any thread which manipulated the object while it was alive. We propose that an object referred to by p be considered unreachable only when all stores to that object have been committed (Commit $\vec{p}$ in every thread). This constraint is necessary for a garbage collector to safely recycle the object's storage. The finalizer must begin with Reconcile $\vec{p}$; any other object references from the finalizer must synchronize in the usual way, with the assumption that each finalizer might run in a distinct thread.

Using CRF as a formalism for reasoning about Java is proving useful. Its descriptive flavor makes it possible to reason clearly about program behavior. Using the semantics in Section 5 we can reason about language-level constructs (such as write-once memory) which have no architectural equivalent. In Section 6 we gave a straightforward translation of language-level memory operations into CRF code; the resulting translation captures fine-grained consistency constraints, allowing a simple solution to the String Problem. Moreover, our translation captures high-level memory semantics which allow local reasoning about thread behavior and which permit bytecode-level optimizations which are currently disallowed. From this we derive the programmer-level view of memory synchronization given in Section 4. Our approach is simplified by making instruction reordering is a clear and fundamental part of the semantics. Finally, the memory model we describe can be realized on current processors using familiar compilation techniques.

Acknowledgments

Jan-Willem Maessen would like to thank Vivek Sarkar and the rest of the Jalapeño group at IBM, who first got him interested in the problems with Java's memory model and in the connections between memory model and compilation. The model presented in this paper has been heavily revised based on several meetings including the authors, Bill Pugh, Jeremy Manson, Guy Steele, and David Detlefs. Because we developed the model independently, however, it is likely to differ in some ways from any models they may propose. Those meetings followed extensive (and invaluable) feedback from all of the above on older drafts of the paper. Marc Snir and various anonymous referees all provided useful comments on the paper.

8. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, Dec. 1996.

[2] Arvind, A. Caro, J.-W. Maessen, and S. Aditya. A multithreaded substrate and compilation model for the implicitly parallel language pH. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[3] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. DAG-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[4] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *OOPSLA 1999 Conference Proceedings*, October 1999.

[5] G. R. Gao and V. Sarkar. Location Consistency – A New Memory Model and Cache Coherence Protocol. Technical Memo 16, CAPSL Laboratory, Department of Electrical and Computer Engineering, University of Delaware, Feb. 1998.

[6] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. Phd. thesis, Stanford University, 1995.

[7] A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations for java memory behavior. Technion/CS Technical Report CS0922, Computer Science Department, Technion, Nov. 1997.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Menlo Park, CA, 1996.

[9] Java memory model mailing list.

[10] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, CA, 1997.

[12] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. Morgan Kaufmann, 1994.

[13] W. Pugh. Fixing the java memory model. In *Proceedings of the ACM Java Grande Conference*, June 1999.

[14] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[15] X. Shen. Design and Verification of Adaptive Cache Coherence Protocols. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Oct. 1999.

[16] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium On Computer Architecture, Atlanta, Georgia*, May 1999.

[17] R. L. Sites and R. T. Witek, editors. *Alpha AXP Architecture Reference Manual (Second Edition)*. Butterworth-Heinemann, 1995.

[18] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual (Version 9*. Prentice-Hall, 1994.

[19] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA 1999 Conference Proceedings*, October 1999.