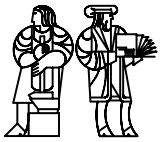
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Improving the Java Memory Model Using CRF

Computation Structures Group Memo 428
November 15, 1999

Jan-Willem Maessen, Arvind, and Xiaowei Shen

Submitted to PLDI 2000, Vancouver, BC. This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Improving the Java Memory Model Using CRF

Jan-Willem Maessen, Arvind, and Xiaowei Shen*

Abstract

We describe an alternative memory semantics for Java programs using an enriched version of the Commit/Reconcile/Fence (CRF) memory model [SAR99]. We need to enrich CRF with semantics for Java's monitor-style locking, and with an instruction to mark cached data which will never change so that we can give semantics for final slots. With these enrichments, we give an instruction-by-instruction translation of Java memory operations into CRF operations. The resulting Java memory semantics allow a number of optimizations such as load reordering that are currently prohibited. Using the translation, we develop a simple thread-local algebraic semantics for Java so that optimizations can be expressed at the source or bytecode level. Finally, we show how the given semantics can be applied to give a simple dependency analysis algorithm for Java.

1 Introduction

Java is the first widely-accepted computer language to contain language-level support for multi-threaded programs running on multiple processors. Java objects are shared between all threads; thus, 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 a new semantics for Java memory operations in terms of an enriched version of the Commit/Reconcile/Fence (CRF) memory model. This semantics is 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. Reordering of operations is captured compactly in a reordering table. We enrich CRF by adding monitor-like locks and allowing memory to be frozen to capture the write-once semantics of final fields. From the enriched CRF semantics we obtain a high-level *algebraic semantics* which captures the legal reorderings of instructions within a single thread. Java can be compiled efficiently by referring to the algebraic semantics or by representing CRF operations. Side conditions on reordering correspond to compiler analyses such as escape analysis, alias analysis, and pointer analysis. This is the first detailed presentation of compilation using CRF.

The problems with the existing Java memory model fall into three broad categories. First, the memory model given in Chapter 17 of the Java Language Specification [GJS96] is difficult to understand. It views the operation of memory in 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. 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

*[jmaessen,arvind,xwshen]@lcs.mit.edu. This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150. *Submitted to PLDI 2000, Vancouver, BC*

CRF model we instead buffer the instruction stream. We then reason explicitly about the order in which buffered instructions act on the cache. We translate every Java memory operation into one or more CRF instructions.

The new mapping from Java to CRF permits optimizations prohibited by the current Java memory model. The current model requires what Pugh [Pug99] refers to as coherence (in all examples, memory cells are assumed to initially contain zero or null, and no extraneous writes occur):

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

Java requires that if we observe w=2 then x=2. Consider this source code:

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

If `p = q` at run time, the compiler is forbidden from performing fetch elimination and having `x = v`, as the above example shows. The memory model we give in this paper permits loads to be re-ordered without recourse to alias analysis and as such allows this optimization. This change reflects current practice (reported as Bug #4242244 in Sun's bug parade).

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. Pugh [Pug99] gives the following example of a seemingly innocuous reordering which changes the underlying memory semantics (the right-hand side cannot in this case actually be expressed legally in Java, but describes the order of operations on main memory):

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

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

Simply exchanging the unrelated assignments `u` and `v` prevents further code motion of assignment `u`. In the CRF model, because operation ordering is resolved at the instruction level we can always express legal instruction reorderings directly in code. The algebraic semantics we develop permits such reasoning to be extended to the source language and bytecode levels.

The current memory model for Java does not provide memory coherence semantics for final locations. 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.

The semantics of final fields cannot be captured in the CRF translation without introducing a new operation to declare that a field will not change. Moreover, keeping the contents of final locations themselves coherent 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 theCharacters;
   public String(char [] value) {
     char [] internalValue = value.clone();
     theCharacters = internalValue;
   }
  etc.
}
```

How do we guarantee that the call to `value.clone()` finishes all its writes before another thread looks at the contents of `theCharacters`? Writers must synchronize before a field may be read remotely, and readers must synchronize before performing the first read of a final field. If the copy has not completed, a partially-initialized string could be seen—a big security problem when strings are used for naming internal state. 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 a semantics which guarantees that the contents of `theCharacters` will be up to date as of initialization.

In an effort to keep the memory model as permissive as possible we make use of data-dependent memory synchronization to solve the string problem. This permissiveness is essential to obtaining an implementation where only the object involved in the final store requires synchronization. This result is 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 [Rep99].

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. We will be concerned about the state of the object pointed to by an object reference whenever that reference is loaded or stored. We therefore give simplified translations for memory operations on scalar types.

2 Overview of the Memory Model

A brief overview of the proposed Java memory model will make the semantics in the paper clear. At a high level there are three kinds of loads and stores in the Java language: Load and Store for regular locations, LoadV and StoreV for volatile locations, and LoadF and StoreF for final locations. We indicate scalar operations with a subscript S. The correspondence between Java constructs and these abstract operations is given in Figure 2. Note the notation: We use o and p to stand for object references, a and b to stand for addresses, m and n to stand for scalars, v and w to stand for arbitrary values (scalars or object references), i to stand for an array index, and r to stand for the result of an instruction. We formalize the model by translating Java memory operations into an enriched version of CRF. In doing so, we try to be as permissive as possible, giving the implementation more flexibility while providing these guarantees to the programmer:

Storage Class	Source	Bytecode	Operation
regular	`int x = o.n;`	[i,l,f,d]load	Load_S
regular	`Object p = o.q;`	aload	Load
regular	`o.n = 5;`	[i,l,f,d]store	Store_S
regular	`o.q = (Object) p;`	astore	Store
`volatile`	`int x = o.n;`	[i,l,f,d]load	LoadV_S
`volatile`	`Object p = o.q;`	aload	LoadV
`volatile`	`o.n = 5;`	[i,l,f,d]store	StoreV_S
`volatile`	`o.q = (Object) p;`	astore	StoreV
`final`	`int x = o.n;`	[i,l,f,d]load	LoadF_S
`final`	`Object p = o.q;`	aload	LoadF
`final`	`o.n = 5;`	[i,l,f,d]store	StoreF_S
`final`	`o.q = (Object) p;`	astore	StoreF
	`int x = a[i];`	[i,l,f,d]aload	Load_S
	`Object p = a[i];`	aaload	Load
	`a[i] = 5;`	[i,l,f,d]astore	Store_S
	`a[i] = (Object) p;`	aastore	Store

Figure 1: Mapping from Java byte code operations to memory operations

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

It is always safe to change any field's storage class to volatile. To ensure that a regular field may be made volatile without affecting existing code, volatile fields obey the same locking semantics as regular fields.

Regular loads and stores must be properly synchronized. No operation after a lock may be moved before a lock, and no operation before an unlock may be moved after an unlock. *This means that Java code must synchronize on a shared monitor in order to access shared fields safely.*

Final loads are considered constant. This will be true anyhow if final fields are properly initialized before they escape a thread. This assumption permits aggressive re-ordering and fetch elimination of final loads.

Final and Volatile fields provide initialization safety. We ensure the data being read is current before every volatile load and before the first final load from a particular address in a given thread.

Final and Volatile fields snapshot their contents. If such a field holds a reference, then the object being referenced must be up to date as of the time the field is written. This is the only exception to the synchronization requirement for regular fields. This allows us to capture ordinary mutable data in a consistent fashion, allowing us to implement `MyString` as described above. However, we only guarantee this for the object pointed at, not for any objects it may point to—the stronger guarantee is hard to reason about and has dubious benefit.

No need for prescient stores. Prescient stores are used in Java primarily to capture reordering of writes in compiler or processor. Speculation is captured using reordering rules. Arbitrary processor-level control speculation is permitted by assuming an oracle which correctly predicts the outcome of every branch; any reordering permitted under this assumption is allowed by our semantics. We do not address the effects of value speculation in this paper.

3 The CRF model

CRF is intended for architects and compiler writers. It is defined by giving precise semantics to the memory instructions so that every CRF program has a well-defined operational behavior. We use these semantics to derive source-level compiler optimizations. CRF memory operations act on a cache; each Java thread has its own cache, which is initially a copy of the cache of the thread which started it. We decompose conventional Load and Store instructions into some finer grain instructions. In CRF, 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 processor can be observed by another processor 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 instructions with semicolon, which is associative (we can also glue groups of instructions together with semicolon):

$$r_1 = \text{Storel } a,\ 5;\ r_2 = \text{Loadl } a$$

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 very complicated to describe if those memory operations involve manipulating a stack.

3.1 Rewriting semantics

Loads and stores are local, acting purely on the cache:

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

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

Note the notation: Each thread consists of a sequence of pending instructions, a group of completed instructions, and a cache which maps addresses to values tagged with a state of either Clean, Dirty, Locked, or Frozen. 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 and entries in the cache with a slash (/)—their exact arrangement doesn't matter, and we consider the simplest order we can.

In order to implement cache coherence, we must have some way of moving values to and from main memory and between threads.

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

Here we see an entire system, which contains multiple threads and a shared global memory. A thread can at any time cache an entry contained in global memory if it has not done so already. The state of main memory is needed only for these coherence operations—instruction execution happens purely locally within a thread.

$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache}\ /[a := v, \text{Dirty}]\ )\ /\ \mathit{threads},\ \mathit{memory}/[a := -] \quad \Rightarrow$$
$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache}\ /[a := v, \text{Clean}]\ )\ /\ \mathit{threads},\ \mathit{memory}/[a := v]$$

$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache}\ )\ /\ \mathit{threads},\ \mathit{memory}/[a := 0]$$
$$\text{where } \mathit{cache} \text{ contains no mapping for } a \quad \Rightarrow$$
$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache}\ /[a := 0, \text{Locked}])\ /\ \mathit{threads},\ \mathit{memory}$$

$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache}\ /[a := 0, \text{Locked}])\ /\ \mathit{threads},\ \mathit{memory}$$
$$\text{where } \mathit{cache} \text{ contains no mapping for } a \quad \Rightarrow$$
$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache}\ )\ /\ \mathit{threads},\ \mathit{memory}/[a := 0]$$

$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache}/[a := v, s])$$
$$\text{where } s \text{ is Clean or Frozen} \quad \Rightarrow$$
$$(\mathit{instructions},\ \mathit{completions},\ \mathit{cache})$$

These rules are referred to as *background operations* because they can happen at any time irrespective of the instructions in any thread. To ensure that coherence operations complete, we use the thread-level operations Commit and Reconcile:

$$(r = \text{Commit } a;\ \mathit{instructions},\ \mathit{completions},\ \mathit{cache})$$
$$\text{where } a \text{ is not in } \mathit{cache}, \text{ or } a \text{ is Clean} \quad \Rightarrow$$
$$(\mathit{instructions},\ r = \surd\ /\ \mathit{completions},\ \mathit{cache})$$

$$(r = \text{Reconcile } a; \mathit{instructions},\ \mathit{completions},\ \mathit{cache})$$
$$\text{where } a \text{ is not in } \mathit{cache} \text{ or } a \text{ is Dirty or Frozen} \quad \Rightarrow$$
$$(\mathit{instructions},\ r = \surd\ /\ \mathit{completions},\ \mathit{cache})$$

Commit ensures the results of a write reach main memory. Reconcile ensures that the cache is refreshed with new data before a 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. Note also that 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.

We augment CRF with locking operations designed to implement Java's monitor-style synchronization. Lock and Unlock are the only *atomic* memory operations, and as such require Locked access to their address (and the corresponding coherence operations above). Locks contain a count of the number of times the owning thread has locked the lock.

$$(r = \text{Lock } l;\ \mathit{instructions},\ \mathit{completions},\ \mathit{cache}/[l := n, \text{Locked}]) \quad \Rightarrow$$
$$(\mathit{instructions},\ r = n\ /\ \mathit{completions},\ \mathit{cache}/[l := n+1, \text{Locked}])$$

$$(r = \text{Unlock } l; \mathit{instructions},\ \mathit{completions},\ \mathit{cache}/[l := n+1, \text{Locked}]) \quad \Rightarrow$$
$$(\mathit{instructions},\ r = n\ /\ \mathit{completions},\ \mathit{cache}/[l := n, \text{Locked}])$$

Note that we rely on Java's type safety to ensure that a location is the target of a Lock instruction if and only if it is actually a monitor. There are therefore no concerns about reading and writing locked locations.

We 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 should not change and can therefore remain in the cache indefinitely. A Freeze operation marks a cache line as Frozen:

2nd ⇒ 1st ⇓	Loadl	Storel	Lock	Unlock	Freeze	Commit	Reconcile	$\text{Fence}_{\text{r}\alpha}$	$\text{Fence}_{\text{w}\alpha}$
Loadl		a	a	a	a			b	
Storel	a	a				a			
Lock	a			a				b	b
Unlock	a		a					b	b
Freeze									
Commit					a				b
Reconcile	a								
$\text{Fence}_{\alpha\text{r}}$			c	c			c		
$\text{Fence}_{\alpha\text{w}}$		c	c	c					

a: Addresses must not match
b: Address of 1st instr must not match pre-address of fence
c: Address of 2nd instr must not match post-address of 1st instr

Figure 2: Reordering table for extended CRF. Blank entries may always be reordered.

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

3.2 Reordering

Much of the power of the CRF model comes from the ability to reorder instructions (provided we respect data dependencies). For our purposes this reordering captures a mix of both compiler optimizations and architectural speculation and reordering. Fine-grained Fence instructions enforce ordering if needed. The Fence instruction itself doesn't affect memory state when it executes:

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

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 fences preceding reads on address a and succeeding writes on address b. Note that a $\text{Fence}_{\text{w}\alpha}$ 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 $\text{Fence}_{\alpha\text{r}}$ instruction imposes ordering constraints on following Reconcile instructions instead of Loadl instructions.

Reordering is specified by referring to the reordering table in Figure 2. 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. The underlying rationale is to allow maximum reordering flexibility for both compiler and architecture.

3.3 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. Extra Commit, Reconcile, and Fence operations in the final program will constrain its execution, but they will not render it illegal. 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 far more expensive than most memory operations.

For modern microprocessors, the Loadl and Storel can be translated into normal Load and Store instructions. The load and store instructions of all current machines implicitly Reconcile and Commit, so explicit Reconcile and Commit operations can simply be eliminated. 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.

Some processor architecture specifications, such as Alpha and PowerPC, allow non-atomic store operations. This means two store operations can be observed in different orders by different threads, even if all reads are separated by fences. For simplicity, the CRF model does not permit this. We have defined G-CRF [She99], a generalized version of CRF that allows the semantic effect of store operations to the same address to be observed in different orders by different processors or threads even though the load operations used in the observation are performed in-order. The mapping from G-CRF to architectures with non-atomic stores remains straightforward.

We can use G-CRF as the underlying memory model to define Java's memory model; the translation from Java remains unchanged. Indeed, this change will have *no effect* on the optimizations described in this paper. Thus, although G-CRF relaxes the Java memory model, there is no clear advantage to the compiler writer. We therefore restrict our attention to CRF in this paper.

4 Translating Java into CRF

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. A complete listing of the CRF instructions used in the translation and their purposes can be found in Figure 3. We use these complex coherence operations for two reasons: First, they yield the most permissive possible semantics (by avoiding global synchronization of all of memory); we are particularly keen to eliminate accidental synchronization between unrelated objects. Second, it is possible for a compiler to reason clearly and simply about the addresses involved in these derived operations, allowing us to choose a good instruction ordering statically. While we could represent these barriers precisely at run time, doing so would be expensive; moreover, current architectures do not support fine-grained synchronization. Generous reordering permits many fine-grained barriers to be consolidated into a few global ones.

4.1 Regular memory

CRF's fine-grained memory operations nicely express the semantics of regular memory locations, which ordinarily require locking for coherence. We synchronize on both volatile and regular locations during locking so that we can safely change a regular location into a volatile location without changing the behavior of a program. Because regular locations are release consistent, loads and stores are purely local operations; consistency operations need occur only when we actually lock and unlock an object:

$\vec{p}$	The non-lock fields in the object pointed to by p
$*_{\mathrm{V}}$	All volatile locations
$*_{\mathrm{VR}}$	All volatile and regular locations
$*_{\mathrm{VRL}}$	All non-final locations (volatile, regular, and lock)
v = Loadl a;	Load v from a locally
Storel a,v;	Store v into a locally
Commit a;	Commit StoreF to a
Reconcile a;	Reconcile before Loadl a
Freeze a;	Freeze contents of a in cache
Commit $\vec{p}$;	Commit writes, if any, to fields in p
$\mathrm{Fence}_{\mathrm{ww}}$ $\vec{p}$, a;	Complete Commit $\vec{p}$ before allowing write to a
$\mathrm{Fence}_{\mathrm{rw}}$ $*_{\mathrm{V}}$,a;	Complete last volatile Loadl before allowing Storel to a
$\mathrm{Fence}_{\mathrm{ww}}$ $*_{\mathrm{V}}$,a;	Complete last volatile Commit before allowing Storel to a
$\mathrm{Fence}_{\mathrm{rr}}$ $*_{\mathrm{V}}$,a;	Complete last volatile Loadl before allowing Reconcile a
$\mathrm{Fence}_{\mathrm{wr}}$ $*_{\mathrm{V}}$,a;	Complete last volatile Commit before allowing Reconcile a
Lock l;	Monitor-style lock
Unlock l;	Monitor-style unlock
Reconcile $*_{\mathrm{VR}}$;	Reconcile all non-final storage
Commit $*_{\mathrm{VR}}$;	Commit all non-final storage
$\mathrm{Fence}_{\mathrm{wr}}$ l, $*_{\mathrm{VRL}}$;	Wait for Lock l before allowing locks, unlocks or Reconcile $*_{\mathrm{VR}}$
$\mathrm{Fence}_{\mathrm{ww}}$ l, $*_{\mathrm{VRL}}$;	Wait for Lock l before allowing locks, unlocks or writes to non-finals
$\mathrm{Fence}_{\mathrm{ww}}$ $*_{\mathrm{VRL}}$, l;	Wait for locking and Commit $*_{\mathrm{VR}}$ before allowing Unlock l
$\mathrm{Fence}_{\mathrm{rw}}$ $*_{\mathrm{VRL}}$, l;	Wait for locking and non-final reads before allowing Unlock l

Figure 3: Address groups and CRF instructions used in translation of Java operations.

Store a,p	$\equiv$	$\mathrm{Fence}_{\mathrm{ww}}$ $\vec{p}$, a; Storel a,p
p = Load a	$\equiv$	p = Loadl a
Store_S a,n	$\equiv$	Storel a,n
n = Load_S a	$\equiv$	n = Loadl a
Enter l	$\equiv$	Lock l; $\mathrm{Fence}_{\mathrm{wr}}$ l, $*_{\mathrm{VRL}}$; $\mathrm{Fence}_{\mathrm{ww}}$ l, $*_{\mathrm{VRL}}$; Reconcile $*_{\mathrm{VR}}$
Exit l	$\equiv$	Commit $*_{\mathrm{VR}}$; $\mathrm{Fence}_{\mathrm{ww}}$ $*_{\mathrm{VRL}}$, l; $\mathrm{Fence}_{\mathrm{rw}}$ $*_{\mathrm{VRL}}$, l; Unlock l

The $\mathrm{Fence}_{\mathrm{ww}}$ in the translation of Store guarantees initialization safety—writes to final and volatile fields pointed to by p will complete before the field is written, so that if a is later read in another thread those fields will be up to date. The fence does not affect regular fields because Load does not actually reconcile its result.

4.2 Final fields

The semantics of final fields require explanation. We must ensure that an object reached through a final field is up to date as of the time the field is written (for example, in `MyString` the contents of the array referred to by `theCharacters`). Note that we need to perform mirror-image synchronization at the reader and the writer; if either synchronization is omitted the memory order is weakened and we lose this guarantee. We do this using operations on $\vec{p}$:

StoreF a,p	$\equiv$	Commit $\vec{p}$; Fence$_{\text{ww}}$ $\vec{p}$, a; Storel a,p; Commit a; Freeze a; Reconcile $\vec{p}$
p = LoadF a	$\equiv$	Reconcile a; p = Loadl a; Freeze a; Reconcile $\vec{p}$
StoreF$_S$ a,n	$\equiv$	Storel a,n; Commit a; Freeze a
n = LoadF$_S$ a	$\equiv$	Reconcile a; n = Loadl a; Freeze a

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:

Reconcile a; p_1 = Loadl a; Freeze a; Reconcile $\vec{p}_1$; Reconcile a; $p2$ = Loadl a; Freeze a; Reconcile $\vec{p}_2$	$\equiv$	Reconcile a; p_1 = Loadl a; p_2 = Loadl a; Reconcile $\vec{p}_1$; Reconcile $\vec{p}_2$; Freeze a; Freeze a	$\equiv$	Reconcile a; p_1 = Loadl a; Reconcile $\vec{p}_1$; Freeze a; $p_2 = p_1$

Replacing p_2 = Loadl a by $p_1 = p_2$ 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:

Commit $\vec{p}_1$; Fence$_{\text{ww}}$ $\vec{p}_1$, a; Storel a,p_1; Commit a; Freeze a; Reconcile $\vec{p}_1$; Reconcile a; p_2 = Loadl a; Freeze a; Reconcile $\vec{p}_2$	$\equiv$	Commit $\vec{p}_1$; Fence$_{\text{ww}}$ $\vec{p}_1$, a; Storel a,p_1; p_2 = Loadl a; Commit a; Reconcile $\vec{p}_1$; Reconcile $\vec{p}_2$; Freeze a; Freeze a	$\equiv$	Commit $\vec{p}_1$; Fence$_{\text{ww}}$ $\vec{p}_1$, a; Storel a,p_1; Commit a; Reconcile $\vec{p}_1$; Freeze a; $p_2 = p_1$

Again, we know that we can fetch-eliminate $p_2 = \text{Loadl } a$. The redundant synchronization then disappears, and once again we have fetch-eliminated the LoadF. Note that we must Reconcile $\vec{p}$ during a StoreF to allow the fetch elimination to work! Because there is no matching Commit in LoadF, this will have no direct effect on the interaction between LoadF and StoreF. It does leave a small but undesirable wart on our semantics: writing a shared object to an unshared final field will make the object coherent with respect to main memory. However, the reconcile can be reordered much earlier in the instruction stream, so this can't meaningfully be exploited to synchronize shared objects.

4.3 Volatile memory

Operations on volatile fields 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. The operations on $\vec{p}$ ensure that we can safely replace a final field with a volatile field without affecting pre-existing program behavior.

$$
\begin{array}{lcl}
\text{StoreV } a,p & \equiv & \text{Commit } \vec{p};\\
 & & \text{Fence}_{\text{ww}}\ \vec{p},\ a;\\
 & & \text{Fence}_{\text{rw}}\ *_{\text{V}},a;\\
 & & \text{Fence}_{\text{rw}}\ *_{\text{V}},a;\\
 & & \text{Storel } a,p;\\
 & & \text{Commit } a;\\
 & & \text{Reconcile } \vec{p}\\
p = \text{LoadV } a & \equiv & \text{Fence}_{\text{rr}}\ *_{\text{V}},a;\\
 & & \text{Fence}_{\text{wr}}\ *_{\text{V}},a;\\
 & & \text{Reconcile } a;\\
 & & p = \text{Loadl } a;\\
 & & \text{Reconcile } \vec{p}\\
\text{StoreV}_S\ a,n & \equiv & \text{Fence}_{\text{rw}}\ *_{\text{V}},a;\\
 & & \text{Fence}_{\text{ww}}\ *_{\text{V}},a;\\
 & & \text{Storel } a,n;\\
 & & \text{Commit } a\\
n = \text{LoadV}_S\ a & \equiv & \text{Fence}_{\text{rr}}\ *_{\text{V}},a;\\
 & & \text{Fence}_{\text{wr}}\ *_{\text{V}},a;\\
 & & \text{Reconcile } a;\\
 & & n = \text{Loadl } a
\end{array}
$$

5 Compilation

For the purposes of compilation, it is useful to have a high-level, static, purely local notion of how memory operations behave *within* a thread. We can use the reordering table for enriched CRF to prove reordering properties for our translations, leading to just such a semantics. For example, consider reordering StoreF a,p and $q = \text{LoadV } b$. The only nontrivial reorderings in the translation involve a and b; we are saved by Java's type system, which tells us that volatile and final storage are disjoint and thus $a \neq b$. A similar argument for other pairs of operations yields the reordering table in Figure 4. A Java-to-bytecode compiler can reorder instructions according to the table without resorting to complex reasoning about the program's memory behavior (Pugh's example, given in the Introduction, shows this is not possible in the current Java memory model).

2nd⇒ 1st⇓	Load	Store	LoadV	StoreV	LoadF	StoreF	Enter	Exit
Load		a						no
Store	a	a		b		b		no
LoadV			no	no	c			no
StoreV		b	no	no		b		no
LoadF	c		c		c	a		
StoreF		b		b	a	$a \wedge b$		
Enter	no	no	no	no			no	no
Exit							no	no

a: Addresses must not match

b: Unless the following store is a reference to the object which is stored to by the first store, e.g. Store $p.f,v$; StoreF a,v is disallowed.

c: When a preceding LoadF or StoreF to the same address exists.

Figure 4: Reordering table for algebraic semantics. Blank entries may always be reordered.

Condition b guarantees that objects are completely initialized before references to those objects are stored in fields visible to other processors. It corresponds to the $\vec{p}$ operations on stores. Condition c corresponds to the motion of Freeze operations through the program code, and can be formalized like so:

LoadF/StoreF a;		LoadF/StoreF a;
instrs;	$\equiv$	*instrs*;
$o =$ LoadF a;		LoadF $p.w$;
LoadF $p.w$		$o =$ LoadF a

5.1 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. For example, we can eliminate a load operation which is preceded by a load or store to the same address, and replace it with the value read or written—it is always legal to execute both instructions in quick succession, possibly updating main memory as we go. The discussion of final fields gave the proof of this property for LoadF and StoreF.

5.2 Dependency Analysis

Most 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 for either the algebraic semantics or for low-level CRF is very simple. For every pair of instructions A and B, where A initially occurs before B, 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 after all the edges that can be derived by transitivity are removed. Figure 5 shows an example of applying this procedure. In practice, of course, we can be more careful and avoid adding extra transitive edges to the graph.

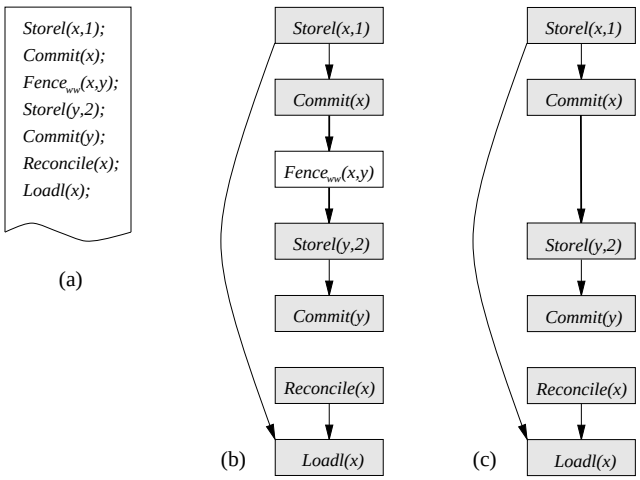


Figure 5: Partial Order of CRF Programs

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 condition a. Distinct fields 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 condition b. The same information allows us to manipulate coherence operations on $\vec{p}$ in the underlying CRF.

Value propagation algorithms can be used to detect when a final field has already been read by a particular thread, allowing us to enforce condition c. This also allows us to reason about Freeze operations in the CRF encoding.

Inter-thread escape analysis [CGS+99, WR99] will reveal when an object is used entirely within one thread. In this case, we can eliminate coherency operations on the object. Thus, local fields can ignore locking, and operations to local volatile fields can be reordered.

An efficient realization of the Java Memory Model on any modern architecture will require a code generator which can coalesce memory barriers. Instruction selection based on a BURS-like schema chooses instructions for groups of operations which are connected in the dependency graph. A separate pre-pass can be used to allow barrier instructions to be coalesced. 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.

6 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 [GJS96] and the Java Virtual Machine Specification [LY97]. 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 very

succinctly by Pugh [Pug99]. Others formalize these problems, and compare Java's properties to those of extant memory models [GS97].

CRF is intended as a mechanism-oriented memory model which exposes both data replication and instruction reordering at the ISA level [SAR99]. 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. In doing so, we extend CRF with Lock, Unlock, and Freeze. The Generalized CRF model, which eliminates causality, is presented in detail elsewhere [She99].

Sequential consistency (SC) [Lam79] has been the dominant memory model in parallel computing for decades due to its simplicity. The desire to achieve higher performance has led to various weaker memory models [AH90, AH93, GLL+90, GAG+92, GGH93, KCZ92, BZS93, GS93, GS98, BFJ+96, FL98]. Release consistency [GLL+90, GGH93] is a good representative of programmer-oriented memory models. It assumes that memory accesses to shared variables are guarded by acquire and release operations and allows memory accesses between synchronizations to be performed out-of-order and interleaved each other. The essence of release consistency is that memory accesses before a release must be globally performed before the synchronization lock can be released.

Location consistency (LC) [GS93, GS98] 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.

Some characteristics of memory semantics can be captured by 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 not requiring that stores occur atomically. Many compiler-oriented memory models such as Location Consistency and DAG Consistency [BFJ+96]. 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*. While both methods can describe the same semantics, we feel that our method is simpler because instruction reordering is natural to compiler writers.

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, if every Storel is followed by a Commit then we can do away with the Commit $\vec{p}$ instructions in StoreF and StoreV, and with the global commit in the Exit code, without materially affecting our semantics. We chose this translation to emphasize the symmetry of Commit and Reconcile operations and the locality of operations on regular fields.

As written, synchronization performs a global memory barrier even when the object being locked is not actually shared and even when the lock is already held locally. We would like to be able to totally eliminate unnecessary locking. Experiments show that this can reduce synchronization by approximately 50% [CGS+99]. We can do this in our semantics by recording the owner of a lock when it is locked (initially the creating thread). A memory barrier is only required if the ownership of the lock changes. We omit the details of this translation for brevity.

We have not addressed exceptions. Java's strongly-ordered exception model has 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 suggest that the Java compiler ignore memory ordering constraints arising due to exceptional control flow. Instead, a coarse-grain memory barrier can be used when an exception occurs.

The prescient stores in the original Java memory model render stores non-atomic. We instead choose to represent instruction reordering explicitly. While both methods can describe the same semantics, we feel that our method is simpler because instruction reordering is natural to compiler writers. If there is a genuine need for non-atomic stores in the Java memory semantics, G-CRF can be used. This will not change any of the optimizations discussed in this paper.

Using CRF as a formalism for reasoning about Java is proving useful. We now understand how to enhance CRF so that we can reason about language-level constructs (such as write-once memory) which have no architectural equivalent. Language-level memory operations translate straightforwardly into CRF code. The resulting translation captures fine-grained consistency constraints, allowing a simple solution to the String Problem. Moreover, we have used the translation to develop a high-level memory semantics which allows local reasoning about thread behavior and which permits bytecode-level optimizations which are currently disallowed. This process is simplified by a semantics in which instruction reordering is a clear and fundamental part of the model. Finally, we have shown that the memory model 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. Bill Pugh's talk at MIT and the discussions on the JavaMemoryModel mailing list suggested much of the actual memory model given in this paper. Marc Snir provided useful comments on a draft of the paper.

References

[AH90] Sarita V. Adve and Mark D. Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th International Symposium On Computer Architecture*, pages 2–14, June 1990.

[AH93] Sarita V. Adve and Mark D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, June 1993.

[BFJ+96] Robert Blumofe, Matteo Frigo, Christopher Joerg, Charles Lsiserson, and Keith Randall. DAG-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[BZS93] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON*, 1993.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA 1999 Conference Proceedings*, October 1999.

[FL98] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June/July 1998.

[GAG+92] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for Different Memory Consistency Models. In *Journal of Parallel and Distributed Computing*, pages 399–407, August 1992.

[GGH93] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors". Technical Report CSL-TR-93-568, Computer Systems Laboratory, Stanford University, April 1993.

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

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GS93] Guang R. Gao and Vivek Sarkar. Location Consistency – Stepping Beyond the Barriers of Memory Coherence and Serializability. Technical Memo 78, ACAPS Laboratory, School of Computer Science, McGill University, December 1993.

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

[GS98] Guang R. Gao and Vivek 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, February 1998.

[KCZ92] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium On Computer Architecture*, pages 13–21, May 1992.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, CA, 1997.

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

[Rep99] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[SAR99] Xiaowei Shen, Arvind, and Larry 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.

[She99] Xiaowei Shen. Design and Verification of Adaptive Cache Coherence Protocols. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, October 1999.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA 1999 Conference Proceedings*, October 1999.