Formalizing the Java Memory Model for Multithreaded Program Correctness and Optimization

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom

UUCS-02-011

School of Computing
University of Utah
Salt Lake City, UT 84112, USA

April 2, 2002

Abstract

Standardized language level support for threads is one of the most important features of Java. However, defining and understanding the Java Memory Model (JMM) has turned out to be a big challenge. Several models produced to date are not as easily comparable as first thought. Given the growing interest in multithreaded Java programming, it is essential to have a sound framework that would allow formal specification and reasoning about the JMM.

This paper presents the Uniform Memory Model (UMM), a formal memory model specification framework. With a flexible architecture, it can be easily configured to capture different shared memory semantics including both architectural and language level memory models. Based on guarded commands, UMM is integrated with a model checking utility, providing strong built-in support for formal verification and program analysis. A formal specification of the JMM following the semantics proposed by Manson and Pugh is presented in UMM. Systematic analysis has revealed interesting properties of the proposed semantics. In addition, several mistakes from the original specification have been uncovered.

Formalizing the Java Memory Model for Multithreaded Program Correctness and Optimization

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom
School of Computing, University of Utah
{yyang | ganesh | gary}@cs.utah.edu

Abstract

Standardized language level support for threads is one of the most important features of Java. However, defining and understanding the Java Memory Model (JMM) has turned out to be a big challenge. Several models produced to date are not as easily comparable as first thought. Given the growing interest in multithreaded Java programming, it is essential to have a sound framework that would allow formal specification and reasoning about the JMM.

This paper presents the Uniform Memory Model (UMM), a formal memory model specification framework. With a flexible architecture, it can be easily configured to capture different shared memory semantics including both architectural and language level memory models. Based on guarded commands, UMM is integrated with a model checking utility, providing strong built-in support for formal verification and program analysis. A formal specification of the JMM following the semantics proposed by Manson and Pugh is presented in UMM. Systematic analysis has revealed interesting properties of the proposed semantics. In addition, several mistakes from the original specification have been uncovered.

1 Introduction

Java programmers routinely rely on threads for structuring their programming activities, sometimes even without explicit awareness. As future hardware architectures become more aggressively parallel, multithreaded Java also provides an appealing platform for high performance application development, especially for server applications. The Java Memory Model (JMM), which specifies how threads interact with each other in a concurrent system, is a critical component in the Java threading system. It imposes significant implications to a broad range of engineering activities such as programming pattern developments, compiler optimizations, Java virtual machine (JVM) implementations, and architectural designs.

Unfortunately, developing a rigorous and intuitive JMM has turned out to be a big challenge. The existing JMM is given in Chapter 17 of the Java Language Specification [1]. As summarized by Pugh [2], it is flawed and very hard to understand. On the one hand, it is too strong and prohibits many common optimization techniques. On the other hand, it is too weak and compromises safety guarantees.

The need for improvements in JMM has stimulated broad research interests. Two new semantics have been proposed for Java threads, one by Manson and Pugh [3], the other by Maessen, Arvind, and Shen [4]. We refer these two proposals as $\mathrm{JMM}_{\mathrm{MP}}$ and $\mathrm{JMM}_{\mathrm{CRF}}$ respectively in this paper. The JMM is currently under an official revisionary process [5] and will be replaced in the future. There is also an ongoing discussion in the JMM mailing list [6].

Although [3] and [4] have initiated promising improvements on Java thread semantics, the specification framework can be enhanced in several ways. One area of improvement is towards the support of formal verification. Being able to provide a concise semantics is only part of the goal. People also need to reason their programs against the JMM for compliance. Multithreaded programming is notoriously difficult. Developing efficient and reliable compilation techniques for multithreading is also hard. The difficulty of being able to understand and reason about the JMM has become a major obstacle for allowing Java threading to reach its full potential. Although finding an ultimate solution is not an easy task, integrating formal verification techniques does provide an encouraging first step towards this goal.

Another problem is that both proposals are somewhat limited to the data structures chosen for their specific semantics. Since they use totally different notations, it is hard to formally compare the two models. In addition, none of the proposals can be easily re-configured to support different desired memory model requirements. JMM_{CRF} inherits the architecture from its predecessor hardware model [7]. Java memory operations have to be divided into fine grained Commit/Reconcile/Fence (CRF) instructions to capture the precise thread semantics. This translation process adds unnecessary complexities for describing memory properties. On the other hand, the dependency on cache based architecture prohibits it from describing more relaxed models. JMM_{MP} uses *multiset* structures to record the history of memory operations. In stead of explicitly specifying the intrinsic memory model properties, e.g., the ordering rules, it resorts to nonintuitive mechanisms such as splitting a write instruction and using assertions to enforce certain conditions. While this is sufficient to express the proposed synchronization mechanism, adjusting it to specify different properties is not trivial.

Similar to any software engineering activities, designing a memory model involves a repeated process of fine-tuning and testing. Therefore, a generic specification framework is needed to provide such flexibilities. In addition, a uniform notation is desired to help people understand the differences among different models.

In this paper, we present the Uniform Memory Model (UMM), a formal framework for memory model specification. It explicitly specifies the intrinsic memory model properties and allows one to configure them at ease. It is integrated with a Model Checking tool using Murφ, facilitating formal analysis of corner cases. To aid program analysis, it extends the scope of traditional memory models by including the state information of thread local variables. This enables source level reasoning about program behaviors. The JMM based on the semantics from JMM_{MP} is formally specified and studied using UMM. Subtle design flaws from the proposed semantics are revealed by our systematic analysis using idiom-driven test programs.

We review the related work in the next section. Then we discuss the problems of the current JMM specification. It is followed by an introduction of JMM_{MP}. Our formal specification of the JMM in UMM, primarily based on the semantics proposed in JMM_{MP}, is described in Section 5. In Section 6, we discuss interesting results and compare JMM_{MP} with JMM_{CRF}. We conclude and explore future research opportunities in Section 7 . An equivalence proof between our model and JMM_{MP} is outlined in the Appendix.

2 Related Work

A *memory model* describes how a memory system behaves on memory operations such as reads and writes. Much previous research has concentrated on the processor level memory models. One of the strongest memory models for multiprocessor memory systems is *Sequential Consistency* [8]. Many weaker memory models [9] have been proposed to enable optimizations. One of them is *Lazy Release Consistency* [10], where synchronization is performed by releasing and acquiring a lock. When a lock is released, all previous operations need to be made visible to other processors. When the lock is subsequently acquired by another processor, that processor needs to reconcile with the shared memory to get the updated data. *Lazy Release Consistency* requires an ordering property called *Coherence*. Using the definition given by [11], *Coherence* requires a total order among all *write* instructions at each individual address. Furthermore, this total order respects the program order from each processor. This requirement is further relaxed by *Location Consistency* [12]. The write operations in *Location Consistency* are only "partially" ordered if they are issued by the same processor or if they are synchronized through locks. With the verification capability in UMM, we can formally compare the JMM with some of these conventional models.

To categorize different memory models, Collier [13] specified them based on a formal theory of memory ordering rules. Architectural testing programs can be executed on a target system to test these orderings. Using methods similar to Collier's, Gharachorloo et al. [11] [14] developed a generic framework for specifying the implementation conditions for different memory consistency models. The shortcoming of their approach is that it is nontrivial for people to infer program behaviors from a compound of ordering constraints.

Park and Dill [15] developed an executable specification framework with formal verification capabilities for the *Relaxed Memory Order* (RMO [16]) [17]. We extended this method to the domain of the JMM in our previous work on the analysis of JMM_{CRF} [18]. After adapting JMM_{CRF} to an executable specification, we exercised the model with a suite of test programs to reveal pivotal properties and verify common programming idioms. Roychoudhury and Mitra [19] also applied the same technique to verify the existing JMM,

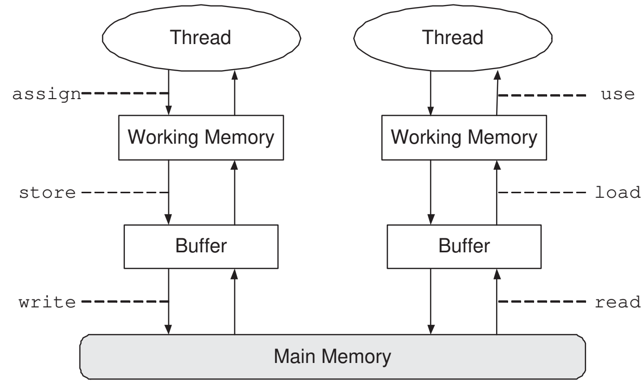

Figure 1: Architecture of the existing Java Memory Model

achieving similar success. However, these previous executable specifications are all restricted to the specific architectures of their target memory models. UMM provides a generic abstraction mechanism for capturing different memory consistency requirements into a formal executable specification.

3 Problems of the Existing JMM

The existing JMM uses a memory hierarchy illustrated in Figure 1. In this framework, every variable has a *working copy* stored in the *working memory*. Eight *actions* are defined. As a thread executes a program, it operates on the working copies of variables via *use*, *assign*, *lock*, and *unlock* actions as dictated by the semantics of the program it is executing. Data transfers performed by JVM between the main memory and the working memory are not atomic. A *read* action initiates the activity of fetching a variable from main memory and is completed by a corresponding *load* action. Similarly, a *store* action initiates the activity of writing a variable to main memory and is committed by a corresponding *write* action. The *lock* and *unlock* actions enforce a synchronization mechanism similar to *Lazy Release Consistency*. The current JMM informally describes sets of rules to impose constraints to the actions. There are many non-obvious implications that can be deduced by combining different rules. As a result, this framework is hard to understand and the lack of rigor in specification has led to some flaws as listed below.

- *Strong ordering restrictions prohibit standard compiler optimizations.*

  The existing JMM requires a total order for operations on each individual variable [20]. Because of this requirement, important compiler optimization techniques such as *fetch elimination* are prohibited. Consider figure 2, where p.x and q.x may become the same variable due to aliasing during execution. The statement k = p.x can not be replaced by k = i by the compiler because a total order among operations on the same variable is required. As a result, adding a seemingly innocuous read instruction j = q.x introduces additional constraints. This is an annoying side effect in a threading system because people need to be able to add debugging read instructions without changing program behaviors. This ordering restriction is actually ignored by some commercial JVM implementations.

Initially, p.x == 0

Thread 1	Thread 2
i = p.x;	p.x = 1;
	q = p;
j = q.x;	
k = p.x;	

Problem: k = p.x can not be replaced by k = i

Figure 2: Current JMM prohibits fetch elimination

Initially, p == null

Thread 1	Thread 2

synchronized(this) {
  p = new Point(1,2);
}
	if(p != null) {
	  r = p.x;
	}

Finally,
can result in r == 0

Figure 3: Current JMM allows premature release of object reference

- *The existing JMM prohibits the removal of "redundant" synchronizations.*

  The present JMM requires a thread to flush all variables to main memory before releasing a lock. Because of this strong requirement on visibility effect, a synchronization block can not be optimized away even if the lock it owns is thread local.

- *Java safety might be compromised.*

  The existing JMM does not guarantee an object to be fully initialized by its constructor before the returned object reference is visible to other threads if there exists a race condition, which might only occur under some weak memory architectures such as Alpha. Take Figure 3 as an example, when thread 2 fetches the object field without locking, it might obtain uninitialized data in the statement `r = p.x` even if `p` is not null. Although this loophole is an extremely rare corner case, it does have serious consequences. Java safety is compromised since it opens the security hole to malicious attacks via race conditions. In particular, many Java objects, such as a String object, are designed to be immutable. If default values before initialization can be observed, the object becomes mutable. Furthermore, popular programming patterns, such as the *double-checked locking* [21] algorithm, are broken under the existing JMM due to the same problem.

- *Semantics for final variables is omitted.*

  Being able to declare a variable as a constant is a useful feature in multithreading systems because it offers more compilation flexibility. Unfortunately, the existing JMM does not mention final variables. In fact, final variables have to be reloaded every time at a synchronization point.

- *Volatile variables are not useful enough.*

  The existing JMM requires operations on volatile variables to be *Sequentially Consistent*. But volatile variable operations do not affect visibility on normal variable operations. Therefore, volatile and non-volatile operations can be reordered. In traditional languages such as C, volatiles are used in device drivers for accessing memory mapped device registers. A `volatile` modifier tells the compiler that the variable should be reloaded for each access. In Java, low level device access is no longer a priority. Volatile variables are mostly used as synchronization flags. Because the existing volatile semantics does not offer sufficient synchronization constraints on normal variables, it is not intuitive to use in practice. Consequently, many JVM implementations do not comply with the present specification.

4 Semantics Proposed by Manson and Pugh

In order to fix the problems listed in Section 3, JMM_{MP} is proposed as a replacement semantics for Java threads. After extensive discussions and debates through the JMM mailing list, some of the thread properties have emerged as leading candidates to appear in the new JMM.

4.1 Desired Properties

- *It should enable the removal of "redundant" synchronizations.*

Similar to the existing JMM, JMM_{MP} uses a release/acquire process for synchronization. However, the visibility restrictions are much relaxed. Instead of permanently flushing all variables when a lock is released, visibility states are only synchronized through the *same* lock. Consequently, if the lock is not used by other threads, the synchronization can be removed since it would never cause any visibility effects.

- *It should relax the total order requirement for operations on the same variable.*

  JMM_{MP} essentially follows *Location Consistency*, which only requires a "partial" order among write instructions on the same variable established through the same thread or synchronization. Most standard compiler optimizations such as fetch elimination are enabled.

- *It should maintain safety guarantees even under race conditions.*

  JMM_{MP} guarantees that all final fields can be initialized properly. To design an immutable object, it is sufficient to declare all its fields as final fields. Variables other than final fields are allowed to be observed prematurely.

- *It should specify reasonable semantics for final variables.*

  A final field v is only initialized once in the constructor of its containing object. At the end of the constructor, v is *frozen* before the reference of the object is returned. If the final variable is improperly exposed to other threads before it is frozen, v is said to be a *pseudo-final* field. Another thread would always observe the initialized value of v unless it is pseudo-final, in which case it can also obtain the default value.

- *It should make volatile variables more useful.*

  JMM_{MP} proposes two changes to the volatile variable semantics. One is weaker and the other is stronger comparing to the original JMM. First, the ordering requirement for volatile operations is relaxed to allow non-atomic volatile writes. Second, the release/acquire semantics is added to volatile variable operations to achieve synchronization effects for normal variables. A write to a volatile field acts as a release and a read of a volatile field acts as an acquire.

4.2 JMM_{MP} Notations

JMM_{MP} is based on an abstract global system that executes one operation from one thread in each step. An operation corresponds to a JVM opcode. Actions occur in a total order which respects the program orders in each thread. The only ordering relaxation explicitly allowed is for *prescient writes* under certain conditions.

4.2.1 Data Structures

A *write* is defined as a unique tuple of $\langle variable, value, GUID \rangle$. JMM_{MP} uses the *multiset* data structure to store history information of memory activities. In particular, the $allWrites$ set is a global set that records every write events that have occurred. Every thread, monitor, or volatile variable k also maintains two local sets, $overwritten_k$ and $previous_k$. The former stores the obsolete writes that are known to k. The latter keeps all previous writes that are known to k. When a variable v is created, a write w with the default value of v is added to the $allWrites$ set and the *previous* set of each thread. Every time a new write is issued, writes in the thread local *previous* set become obsolete to that thread and the new write is added to the *previous* set and the $allWrites$ set. When a read action occurs, the return value is chosen from the $allWrites$ set. But the writes stored in the *overwritten* set of the reading thread are not eligible results.

4.2.2 Prescient Write

A write w may be performed presciently, i.e., executed early, if (a) w is guaranteed to happen, (b) w can not be read from the same thread before where w would normally occur, and (c) any premature reads of w from other threads must not be observable by the thread that issues w via synchronization before where w would normally occur. To capture the prescient write semantics, a write action is splitted into initWrite and performWrite. Special assertion is used in performWrite to ensure that the prescient write conditions are met.

Prescient reads do not need to be explicitly specified. Eligible reordering of read instructions can be deduced as long as it does not result in an illegal execution.

4.2.3 Synchronization Mechanism

The thread local *overwritten* and *previous* sets are synchronized between threads through the release/acquire process. A release operation passes the local sets from a thread to a monitor. An acquire operation passes the local sets from a monitor to a thread. Any non-synchronized write instruction on the same variable from another thread is an eligible write for a read request.

4.2.4 Non-atomic Volatile Writes

Non-atomic volatile writes enable writes on different variables to arrive at different threads in different orders. To capture the semantics, a volatile write is splitted into two consecutive instructions, initVolatileWrite and performVolatileWrite. If thread $t1$ has issued initVolatileWrite but has not completed performVolatileWrite, no other thread can issue initVolatileWrite on the same variable. During this interval, another thread $t2$ can observe either the new value or the previous value of the volatile variable. As soon as $t2$ sees the new value, however, $t2$ can no longer observe the previous value. When performVolatileWrite is completed, no thread can see the previous value.

4.2.5 Final Field Semantics

A very tricky issue in final field semantics arises from the fact that Java does not allow array elements to be declared as final. For example, the implementation of a String class may use a final field r to point to its internal character array. Because the elements pointed by r can not be declared as final, another thread might be able to observe their default values even if they have been initialized before r is frozen.

JMM_{MP} proposes to add a special guarantee to these elements that are referenced by a final field. The visible state of such an element must be captured when the final field is frozen and later synchronized to another thread when these elements are accessed through the final field. Therefore, every final variable v is treated as a special lock. A special release is performed when v is frozen. Subsequently, an acquire is performed when v is read to access its sub-fields. With this mechanism, an immutable object can be implemented by declaring all its fields as final. If any field is a reference to an array or object, it is sufficient to just declare this reference as final.

Adding this special final field requirement substantially complicates JMM_{MP} because synchronization information needs to be passed between the constructing thread and every object pointed by a final field. The variable structure is extended to a *local*, which is a tuple of $\langle a, oF, kF \rangle$ where a is the reference to an object or a primitive value, oF is the overwritten set caused by freezing the final fields, and kF is a set recording what variables are known to have been frozen. Whenever a final object is read, its $knownFrozen$ set associated with its initializing thread is synchronized to the reference of the final object. This allows any subsequent access to its sub-field to know if the sub-field has been initialized.

5 Specifying JMM_{MP} Using UMM

In this section we present a formal specification of the Java Memory Model using the UMM framework. The JMM semantics, except for rules of final fields and control dependency, is based on JMM_{MP}. JMM_{MP} has two versions. [3] is an evolving specification that describes the full semantics of Java threads and [22] is a core subset of it. The one we use in UMM is based on the latest revision of [3] dated as January 11, 2002. Although we follow the specific rules outlined by JMM_{MP}, the exact semantics can be easily adjusted to meet different memory model requirements. The equivalence proof of the semantics, except for volatile and final fields, is given in the Appendix.

5.1 Overview

The UMM uses an *abstract machine* to define thread interactions in a shared memory environment. Memory instructions are categorized as *events*, which may be completed by carrying out certain *actions* if and only if

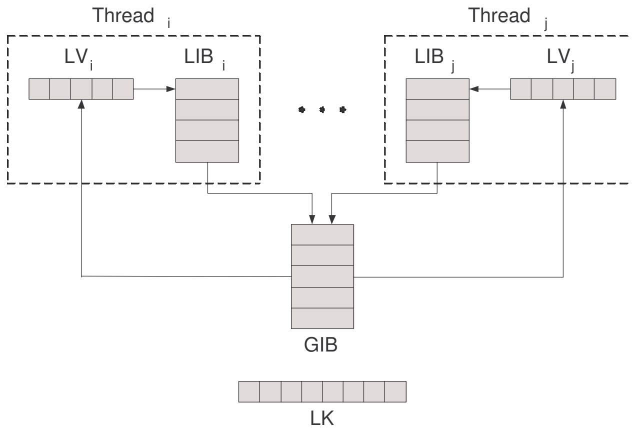

Figure 4: The UMM architecture

specific *conditions* are satisfied. A *transition table* defines all possible events along with their corresponding conditions and actions for the abstract machine.

At any given step, any legal event may be *nondeterministically* chosen and *atomically* completed by the abstract machine. The sequence of permissible actions from various threads constitutes an *execution*. A memory model $\mathcal{M}$ is defined by all possible executions allowed by the abstract machine. An actual implementation, $\mathcal{I}_{\mathcal{M}}$, may choose different architectures and optimization techniques as long as the executions allowed by $\mathcal{I}_{\mathcal{M}}$ are also permitted by $\mathcal{M}$.

5.2 The Architecture

As shown in Figure 4, each thread k has a *local instruction buffer* LIB_k that stores its pending instructions in program order. It also maintains a set of local variables in a *local variable array* LV_k. Each element $\mathsf{LV}_k[v]$ contains the data value of the local variable v. LIB_k and LV_k are not directly exposed to other threads. Thread interactions are communicated through a *global instruction buffer* GIB, which is visible to all threads. GIB stores all previously completed write and synchronization instructions. In general, a read instruction completes when the return value is bound to its target local variable. A write or synchronization instruction completes when it is added to the global instruction buffer. A multithreaded program terminates when all instructions from all threads complete.

The usage of LIB and GIB is motivated by the observation that *local ordering rules* and *global observability rules* are two pivotal properties for understanding thread behaviors. The former dictates *when* an instruction can be issued by a thread and the latter determines *what* value can be read back. In UMM, these properties are explicitly specified as conditions in the transition table.

The local instruction buffers can be used to represent effects caused by both instruction scheduling and data replication. Therefore, there is no need for intermediate layers such as cache.

Although we can store all necessary bookkeeping information in LV, LIB, and GIB to describe any important thread properties, a dedicated global lock array LK is also used for clarity. Each element $\mathsf{LK}[l]$ is a tuple $\langle count, owner \rangle$, where *count* is the number of recursive lock acquisitions and *owner* records the thread that owns the lock l.

5.3 Definitions

Definition 1 *Variable*
*A **global variable** in UMM refers to a static field of a loaded class, an instance field of an allocated object, or an element of an allocated array in Java. It can be further categorized as a normal, volatile, or final variable. A **local variable** in UMM corresponds to a Java local variable or an operand stack location.*

Definition 2 *Instruction*
An ***instruction*** *i is represented by a tuple* $\langle t, pc, op, var, data, local, useLocal, useNew, lock, time\rangle$ *where*

t(i) = t:	*thread that issues the instruction*
pc(i) = pc:	*program counter of the instruction*
op(i) = op:	*instruction operation type*
var(i) = var:	*variable operated by the instruction*
data(i) = data:	*data value in a write instruction*
local(i) = local:	*local variable used to store the return value in a read instruction or local variable to provide the value in a write instruction*
useLocal(i) = useLocal:	*tag in a write instruction i indicating whether the write value* $data(i)$ *needs to be obtained from the local variable local(i)*
useNew(i) = useNew:	*tag in a read volatile instruction to support non-atomic volatile writes*
lock(i) = lock:	*lock in a lock or an unlock instruction*
time(i) = time:	*global counter incremented each time when a local instruction is added to GIB*

5.4 Need for Local Variable Information

Because traditional memory models are designed for processor level architectures, aiding software program analysis is not a common priority in those specifications. They only need to describe how data can be shared between different processors through the main memory. Consequently, a read instruction is usually retired immediately when the return value is obtained. Following the same style, neither JMM_{MP} nor JMM_{CRF} keeps the returned values from read operations. However, Java poses a new challenge to memory model specification with an integrated threading system as part of the programming language. In Java, most programming activities such as computation, flow control, and method invocation, are carried out using local variables. Programmers have a clear need for understanding memory model implications caused by the nondeterministically returned values in local variables. Therefore, it is desired to extend the scope of the memory model by recording the values obtained from read instructions as part of the global state of the transition system.

Based on this observation, we use local variable arrays to keep track thread local variable information. Not only does this reduce the gap between program semantics and memory model semantics, it also provides a clear delimitation between them. This allows us to define the JMM at the Java byte code level as well as the source program level, giving Java programmers an end-to-end view of the memory consistency requirement.

5.5 Memory Operations

A global variable in Java is represented by object.field, where object is the object reference and field is the field name. In this paper, the object.field entity is abstracted to a single variable v. We also follow a convention that uses a, b, c to represent global variables, $r1$, $r2$, $r3$ to represent local variables, and 1, 2, 3 to represent primitive values.

A read operation on a global variable corresponds to a Java program instruction $r1 = a$. It always has a target local variable to store the returned data. A write operation on a global variable can have two formats, $a = r1$ or $a = 1$, depending on whether the *useLocal* tag is set or not. The data value of the write instruction is obtained from a local variable in the former case and is provided by the instruction directly in the latter case. The format $a = r1$ allows one to examine the data flow implications caused by the non-determinism from memory behaviors. If all write instructions have *useLocal* $= false$ and all read instructions use different local variables, the UMM degenerates to the traditional models that do not keep local variable information.

The local variables are not initialized. Java requires them to be assigned before being used. This is implicitly enforced in UMM by data dependency on local variables.

Since we are defining the memory model, only memory operations are identified in our transition system. Instructions such as $r1 = 1$ and $r1 = r2 + r3$ are not included. However, the UMM framework can be easily upgraded to a full blown program analysis system by adding semantics for computational instructions.

Lock and unlock instructions are injected as dictated by Java *synchronized* keyword. They are used to model the mutual exclusion effect as well as the visibility effect.

Event	**Condition**	**Action**
readNormal	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{ReadNormal} \wedge$ $(\exists w \in \text{GIB} : legalNormalWrite(i, w))$	$\text{LV}_{t(i)}[local(i)] := data(w)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
writeNormal	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{WriteNormal}$	**if** $(useLocal(i))$ $i.data := \text{LV}_{t(i)}[local(i)]$; **end**; $\text{GIB} := append(\text{GIB}, i)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
lock	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{Lock} \wedge$ $(\text{LK}[lock(i)].count = 0 \vee$ $\text{LK}[lock(i)].owner = t(i))$	$\text{LK}[lock(i)].count := \text{LK}[lock(i)].count + 1$; $\text{LK}[lock(i)].owner := t(i)$; $\text{GIB} := append(\text{GIB}, i)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
unlock	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{Unlock} \wedge$ $(\text{LK}[lock(i)].count > 0 \wedge$ $\text{LK}[lock(i)].owner = t(i))$	$\text{LK}[lock(i)].count := \text{LK}[lock(i)].count - 1$; $\text{GIB} := append(\text{GIB}, w)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
readVolatile	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{ReadVolatile} \wedge$ $(\exists w \in \text{GIB} :$ $(legalOldWrite(i, w) \vee legalNewWrite(i, w)))$	$\text{LV}_{t(i)}[local(i)] := data(w)$; **if** $(legalNewWrite(i, w))$ $i.useNew := true$; **end**; $\text{GIB} := append(\text{GIB}, i)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
writeVolatile	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{WriteVolatile}$	**if** $(useLocal(i))$ $i.data := \text{LV}_{t(i)}[local(i)]$; **end**; $\text{GIB} := append(\text{GIB}, i)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
readFinal	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{ReadFinal} \wedge$ $(\exists w \in \text{GIB} : legalFinalWrite(i, w))$	$\text{LV}_{t(i)}[local(i)] := data(w)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
writeFinal	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{WriteFinal}$	**if** $(useLocal(i))$ $i.data := \text{LV}_{t(i)}[local(i)]$; **end**; $\text{GIB} := append(\text{GIB}, i)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;
freeze	$\exists i \in \text{LIB}_{t(i)} : ready(i) \wedge op(i) = \text{Freeze}$	$\text{GIB} := append(\text{GIB}, i)$; $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i)$;

Table 1: Transition Table

2nd $\Rightarrow$ 1st $\Downarrow$	Read Normal	Write Normal	Lock	Unlock	Read Volatile	Write Volatile	Read Final	Write Final	Freeze
Read Normal	no	diffVar	no	no	no	no	no	no	no
Write Normal	no	yes	no	no	no	no	no	no	no
Lock	no	no	no	no	no	no	no	no	no
Unlock	no	yes	no	no	no	no	no	no	no
Read Volatile	no	no	no	no	no	no	no	no	no
Write Volatile	no	yes	no	no	no	no	no	no	no
Read Final	no	yes	no	no	no	no	no	no	no
Write Final	no	yes	no	no	no	no	no	no	no
Freeze	no	no	no	no	no	no	no	no	no

Table 2: The Bypassing Table (Table BYPASS)

Finally, a special Freeze instruction for every final field v is added at the end of the constructor that initializes v to indicate v has been frozen.

5.6 Initial Conditions

Initially, instructions from each thread are added to the local instruction buffers according to their original program order. The $useNew$ fields are set to $false$. GIB is initially cleared. Then for every variable v, a write instruction w_{init} with the default value of v is added to GIB. A special thread ID t_{init} is assigned in w_{init}. Finally, the *count* fields in LK are set to 0.

After the abstract machine is set up, it operates according to the transition table specified in Table 1. The conditions and actions corresponding to memory instructions are defined as events in the transition table. Our notation based on guarded commands has been widely used in architectural models [23], making it familiar to many hardware designers.

5.7 Ordering Rules

The execution of an instruction i is only allowed when either all the previous instructions in the same thread have been completed or i is permitted to bypass previous pending instructions according to the memory model and local data dependency. This is enforced by condition *ready*, which is required by every event in the transition table.

Condition *ready* consults the bypassing table BYPASS and guarantees that the execution of an instruction would not violate the ordering requirements from the memory model. The BYPASS table as shown in Table 2 specifies the ordering policy between every pair of instructions. An entry BYPASS[op1][op2] indicates whether an instruction with type *op*2 can bypass a previous instruction with type *op*1, where the value *yes* permits the bypassing, the value *no* prohibits it, and the value *diffVar* allows the bypassing only if the the variables operated by the two instructions are different and not aliased. JMM_{MP} specifies that within each thread operations are usually done in their original order. The exception is that writes may be done *presciently*. The straightforward implementation of UMM follows the same guideline by only allowing normal write instructions to bypass certain previous instructions as shown in Table 2. The equivalence proof in the Appendix is based on Table 2. A more relaxed bypassing policy can also be deduced, which is discussed in Section 5.12.

In addition to the ordering properties set by the memory model, the data dependency imposed by the usage of local variables also need to be obeyed. This is expressed in condition $localDependent$. The helper function $isWrite(i)$ returns *true* if the operation of i is WriteNormal, WriteVolatile, or WriteFinal. Similarly, $isRead(i)$ returns *true* if the operation of i is ReadNormal, ReadVolatile, or ReadFinal. These operation types are defined with respect to the global variables in the instructions. A *read* operation on a global variable actually corresponds to a *write* operation on a local variable.

Condition 1 *ready(i)* $\Longleftrightarrow$

$\neg\exists j \in \text{LIB}_{t(i)} : pc(j) < pc(i) \wedge (localDependent(i, j) \vee$
$\text{BYPASS}[op(j)][op(i)] = no \vee \text{BYPASS}[op(j)][op(i)] = \mathit{diffVar} \wedge var(j) = var(i))$

Condition 2 *localDependent(i, j)* $\Longleftrightarrow$

$t(j) = t(i) \wedge local(j) = local(i) \wedge$
$(isWrite(i) \wedge useLocal(i) \wedge isRead(j) \vee$
$isWrite(j) \wedge useLocal(j) \wedge isRead(i) \vee$
$isRead(i) \wedge isRead(j))$

5.8 Observability Rules

A write or a synchronization instruction carries out actions to update the global state of the abstract machine. The state is observed by a read instruction that returns the value previously set by an eligible write instruction. Besides the ordering rules, the criteria of choosing legal return values is another critical aspect of a memory model.

The synchronization mechanism used by JMM_{MP} plays an important role in selecting legal return values. This is formally captured in condition *synchronized*. Instruction $i1$ can be synchronized with a previous instruction $i2$ via a release/acquire process, where a lock is first released by $t(i2)$ after $i2$ is issued and later acquired by $t(i1)$ before $i1$ is issued. Release can be triggered by an Unlock or a WriteVolatile instruction. Acquire can be triggered by a Lock or a ReadVolatile instruction.

Condition 3 *synchronized(i1, i2)* $\iff$

$$\begin{aligned}\exists l, u \in \text{GIB} : (&op(l) = \text{Lock} \wedge op(u) = \text{Unlock} \wedge lock(l) = lock(u) \vee \\ &op(l) = \text{ReadVolatile} \wedge op(u) = \text{WriteVolatile} \wedge var(l) = var(u)) \wedge \\ &t(l) = t(i1) \wedge (t(u) = t(i2) \vee t(i2) = t_{init}) \wedge \\ &time(i2) < time(u) \wedge time(u) < time(l) \wedge time(l) < time(i1)\end{aligned}$$

The synchronization mechanism follows *Location Consistency*. It requires an ordering relationship as captured in condition *LCOrder*, which can be established if two instructions are from the same thread or if they are synchronized. This ordering relationship is transitive, i.e., $i1$ and $i2$ can be synchronized by a sequence of release/acquire operations across different threads. Therefore, *LCOrder* is recursively defined.

Condition 4 *LCOrder(i1, i2)* $\iff$

$$\begin{aligned}&((t(i1) = t(i2) \vee t(i2) = t_{init}) \wedge pc(i1) > pc(i2) \wedge var(i1) = var(i2)) \vee \\ &synchronized(i1, i2) \vee \\ &(\exists i' \in \text{GIB} : time(i') > time(i2) \wedge time(i') < time(i1) \wedge LCOrder(i1, i') \wedge LCOrder(i', i2))\end{aligned}$$

Condition $legalNormalWrite(r, w)$ defines whether an instruction w is an eligible write for the read request r. w provides a legal return value only if there does not exist another write w' on the same variable between r and w such that r is ordered to w' and w' is ordered to w following *LCOrder*.

Condition 5 *legalNormalWrite(r, w)* $\iff$

$$\begin{aligned}&op(w) = \text{WriteNormal} \wedge var(w) = var(r) \wedge \\ &(\neg\exists w' \in \text{GIB} : op(w') = \text{WriteNormal} \wedge var(w') = var(r) \wedge LCOrder(r, w') \wedge LCOrder(w', w))\end{aligned}$$

5.9 Non-Atomic Volatiles

Conditions $legalOldWrite(r, w)$ and $legalNewWrite(r, w)$ are used to specify the semantics of non-atomic volatile write operations. Suppose the value last written to a volatile variable is set by a WriteVolatile instruction w. After another write instruction w' is performed on the same variable, a ReadVolatile instruction from thread $t(w')$ must always observe the new value set by w' but other threads can get the value either from w' or w. However, once a thread sees the new value set by w', that thread can no longer see the previous value set by w. A special tag *useNew* in the ReadVolatile instruction is used to indicate whether the new value has been observed by the reading thread. Furthermore, the new value set by w' is "committed" if the writing thread $t(w')$ has completed any other instructions that follow w' in thread $t(w')$.

According to condition $legalNewWrite(r, w)$, any WriteVolatile instruction w can be an legal write if it is the *most recent* write for that variable. Condition $legalOldWrite(r, w)$ specifies that w can also be a legal result if (a) w is the *second most recent* WriteVolatile instruction on the same variable, (b) the most recent write has not been "committed" by its writing thread, and (c) the new value has not been observed by the reading thread.

Condition 6 *legalNewWrite(r, w)* $\iff$

$$\begin{aligned}&op(w) = \text{WriteVolatile} \wedge var(w) = var(r) \wedge \\ &(\neg\exists w' \in \text{GIB} : op(w') = \text{WriteVolatile} \wedge var(w') = var(r) \wedge time(w') > time(w))\end{aligned}$$

Condition 7 *legalOldWrite(r, w)* $\iff$

$$\begin{aligned}&op(w) = \text{WriteVolatile} \wedge var(w) = var(r) \wedge t(w) \neq t(r) \wedge \\ &(\exists i1 \in \text{GIB} : op(i1) = \text{WriteVolatile} \wedge var(i1) = var(r) \wedge time(i1) > time(w) \wedge \\ &(\neg\exists i2 \in \text{GIB} : op(i2) = \text{WriteVolatile} \wedge var(i2) = var(r) \wedge time(i2) > time(w) \wedge time(i2) \neq time(i1)) \wedge \\ &(\neg\exists i3 \in \text{GIB} : t(i3) = t(i1) \wedge pc(i3) > pc(i1))) \wedge \\ &(\neg\exists i4 \in \text{GIB} : op(i4) = \text{ReadVolatile} \wedge t(i4) = t(r) \wedge var(i4) = var(r) \wedge time(i4) > time(w) \wedge useNew(i4))\end{aligned}$$

5.10 Final Variable Semantics

In Java, a final field can either be a primitive value or a reference to another object or array. When it is a reference, the Java language only requires that the reference itself can not be modified in the Java code after its initialization but the elements it points to do not have the same guarantee. Also, there does not exist a mechanism in Java to declare array elements as final fields.

As mentioned in Section 4.2.5, JMM_{MP} proposes to add a special requirement for the elements pointed by a final field to support an immutable object that uses an array as its field. This requirement is that if an element pointed by a final field is initialized before the final field is initialized, the default value of this element must not be observable after normal object construction. JMM_{MP} uses a special mechanism to "synchronize" initialization information from the constructing thread to the final reference and eventually to the elements contained by the final reference. However, without explicit support for immutability from the Java language, this mechanism makes the memory semantics substantially more difficult to understand because synchronization information needs to be carried by every variable. It is also not clear how the *exact* semantics can be efficiently implemented by a Java compiler or a JVM since it involves runtime reachability analysis.

While still investigating this issue and trying to find the most reasonable solution, we implement a straightforward definition for final fields in the current UMM. It is slightly different from JMM_{MP} in that it only requires the final field itself to be a constant after being frozen. The observability criteria for final fields is shown in condition *legalFinalWrite*. The default value of the final field (when $t(w) = t_{init}$) can only be observed if the final field is not frozen. In addition, the constructing thread can not observe the default value after the final field is initialized.

Condition 8 *legalFinalWrite(r, w)* $\iff$
$op(w) = \text{WriteFinal} \wedge var(w) = var(r) \wedge$
$(t(w) \neq t_{init} \vee$
$(t(w) = t_{init} \wedge (\neg\exists i1 \in \text{GIB} : op(i1) = \text{Freeze} \wedge var(i1) = var(r)) \wedge$
$(\neg\exists i2 \in \text{GIB} : op(i2) = \text{WriteFinal} \wedge var(i2) = var(r) \wedge t(i2) = t(r))))$

5.11 Control Dependency Issues

The bypassing policy specified in the BYPASS table dictates ordering behaviors of the memory operations on global variables. Thread local data dependency is formally defined in *localDependent*. In addition, thread local control dependency on local variables should also be respected to preserve the meaning of the Java program. However, how to handle control dependency is a tricky issue. A compiler might be able to remove a branch statement if it can determine the control path through program analysis. A policy needs to be set regarding what the criteria is to make such a decision.

JMM_{MP} identifies some special cases and adds two more read actions, guaranteedRedundantRead and guaranteedReadOfWrite which can suppress prescient writes to enable *redundant load elimination* and *forward substitution* under specific situations. For example, the need for guaranteedRedundantRead is motivated by a program shown in Figure 5. In order to allow $r2 = a$ to be replaced by $r2 = r1$ in Thread 1, which would subsequently allow the removal of the if statement, $r2 = a$ must be guaranteed to get the previously read value.

Initially, a == b == 0

Thread 1	Thread 2

r1 = a;
r2 = a;
if(r1 == r2)
    b = 2;
	r3 = b;
	a = r3;

Finally, can r1 == r2 == r3 == 2?

Figure 5: Motiation for guaranteedRedundantRead

Although we could follow the same style by adding similar events in UMM, we do not believe it is a good approach to specify a memory model by enumerating special cases for every optimization need.

2nd ⇒ 1st ⇓	Read Normal	Write Normal	Lock	Unlock	Read Volatile	Write Volatile
Read Normal	yes	diffVar	yes	no	yes	no
Write Normal	diffVar	yes	yes	no	yes	no
Lock	no	no	no	no	no	no
Unlock	yes	yes	no	no	no	no
Read Volatile	no	no	no	no	no	no
Write Volatile	yes	yes	no	no	no	no

Table 3: The Relaxed Bypassing Table

Therefore, we propose a clear and uniform policy regarding control dependency: the compiler may remove a control statement only if the control condition can be guaranteed in *every* possible execution, including all interleaving results caused by thread interactions. This approach should still provide plenty of flexibility for compiler optimizations. If desired, global data flow analysis may be performed. UMM offers a great platform for such analysis. One can simply replace a branch instruction with an assertion. Then the model checker can be run to verify whether the assertion might be violated due to thread interactions.

5.12 Relaxing Ordering Constraints

Although JMM_{MP} does not explicitly relax ordering rules except for prescient writes, possible reordering can be inferred. As long as the reordering does not result in any illegal execution, an implementation is free to do so. In UMM, these effects can be directly described in the BYPASS table to provide a more intuitive view about what is allowed by the memory model. A high performance threading environment requires efficient supports from many components, such as compilation techniques, cache protocol designs, memory architectures, and processor pipelining. Because more liberal ordering rules provide more optimization opportunities at each intermediate layer, it is desired to have a clear view about the allowed reordering.

Table 3 outlines the relaxed bypassing policy for memory instructions except for final variable operations. It does not cover all possible relaxations but it illustrates some of the obvious ones. A ReadNormal instruction is allowed to bypass a previous WriteNormal instruction operated on a different variable or a ReadNormal instruction. Because a presciently performed read instruction would get a value from a subset of the legal results, its return data is still valid. A Lock instruction can bypass previous normal read/write instructions and normal read/write instructions can bypass a previous Unlock instruction. This is motivated by the fact that it is safe to move normal instructions into a synchronization block since it still generates legal results. This relaxation also applies to volatile variable operations which have similar synchronization effects on normal variables.

5.13 Murφ Implementation

The UMM is implemented in Murφ [24], a description language with a syntax similar to C that enables one to specify a transition system based on guarded commands. In addition, Murφ is also a model checking system that supports exhaustive state space enumeration. This makes it an ideal tool for verifying our shared memory system.

Our Murφ program consists of two parts. The first part implements the formal specification of JMM_{MP}, which provides a "black box" that defines Java thread semantics. The transition table in Table 1 is specified as Murφ rules. Ordering rules and observability rules are implemented as Murφ procedures. The second part comprises a suite of test cases. Each test program is defined by specific Murφ initial state and invariants. It is executed with the guidance of the transition system to reveal pivotal properties of the underlying model. Our system can detect deadlocks and invariant violations. To examine test results, two techniques can be applied. The first one uses Murφ invariants to specify that a particular scenario can never occur. If it does occur, a violation trace can be generated to help understand the cause. The second technique uses a special "thread completion" rule, which is triggered only when all threads are completed, to output all possible final results. Our executable specification is a configurable system that enables one to easily set up different test programs, abstract machine parameters, and memory model properties. Running on a PC with a 900 MHz Pentium III processor and 256 MB of RAM, most of our test programs terminate in less than 1 second.

6 Analysis of JMM_{MP}

By systematically exercising JMM_{MP} with idiom-driven test programs, we are able to gain a lot of insights about the model. Since we have developed formal executable models for both JMM_{CRF} [18] and JMM_{MP}, we are able to perform a comparison analysis by running the same test programs on both models. This can help us understand subtle differences between them. As an ongoing process of evaluating the Java Memory Models, we are continuing to develop more comprehensive test programs to cover more interesting properties. In this section we highlight some of the interesting findings based on our preliminary results.

6.1 Ordering Properties

6.1.1 Coherence

JMM_{MP} does not require *Coherence*. This can be detected by the program shown in Figure 6. If $r1 = 2$ and $r2 = 1$ is allowed, the two threads have to observe different orders of writes on the same variable a, which violates *Coherence*. For a normal variable a, this result is allowed by JMM_{MP} but prohibited by JMM_{CRF}.

Initially, $a == 0$

Thread 1	Thread 2
$a = 1;$ $r1 = a;$	$a = 2;$ $r2 = a;$

Finally,
can it result in $r1 == 2$ and $r2 == 1$?

Figure 6: Coherence Test

6.1.2 Write Atomicity for Normal Variables

JMM_{MP} does not require *Write Atomicity*. This can be revealed from the test in Figure 7. For a normal variable a, the result in Figure 7 is allowed by JMM_{MP} but forbidden by JMM_{CRF}. Because the CRF model uses the shared memory as the rendezvous point between threads and caches, it has to enforce *Write Atomicity*.

Initially, $a == 0$

Thread 1	Thread 2
$a = 1;$ $r1 = a;$ $r2 = a;$	$a = 2;$ $r3 = a;$ $r4 = a;$

Finally,
can it result in $r1 == 1$, $r2 == 2$, $r3 == 2$, and $r4 == 1$?

Figure 7: Write Atomicity Test

6.2 Synchronization Mechanism

JMM_{MP} follows *Location Consistency*, which does not require *Coherence*. When thread t issues a read instruction, any previous unsynchronized writes on the same variable issued by other threads can be observed, in any order. Therefore, JMM_{MP} is strictly weaker than *Lazy Release Consistency*. Without synchronization, thread interleaving may result in very surprising results. An example is shown in Figure 8.

Initially, $a == 0$

Thread 1	Thread 2
$a = 1;$ $a = 2;$	$r1 = a;$ $r2 = a;$ $r3 = a;$

Finally,
can it result in $r1 == r3 == 1$ and $r2 == 2$?

Figure 8: Legal Result under Location Consistency

6.3 Constructor Property

The constructor property is studied by the program in Figure 9. Thread 1 simulates the constructing thread. It initializes the field before releasing the object reference. Thread 2 simulates another thread trying to access the object field without synchronization. $Membar1$ and $Membar2$ are some hypothetic memory barriers that prevents instructions from acrossing them, which can be easily implemented in our program by simply setting some test specific bypassing rules. This program essentially simulates the construction mechanism used by JMM_{CRF}, where $Membar1$ is a special EndCon instruction indicating the completion of the constructor and $Membar2$ is the data dependency enforced by program semantics when accessing $field$ through $reference$. If $field$ is a normal variable, this mechanism works under JMM_{CRF} but fails under JMM_{MP}. In JMM_{MP} the default write to $field$ is still a valid write since there does not exists an ordering requirement on non-synchronized writes. However, if $field$ is declared as a final variable and the Freeze instruction is used for $Membar1$, Thread 2 would never observe the default value of $field$ if $reference$ is initialized.

This illustrates the different strategies used by the two models for preventing premature releases during object construction. JMM_{CRF} treats all fields uniformly and JMM_{MP} only guarantees fully initialized fields if they are final or pointed by final fields.

Initially, $reference == field == 0$

Thread 1	Thread 2
$field = 1;$ $Membar1;$ $reference = 1;$	$r1 = reference;$ $Membar2$ $r2 = field;$

Finally,
can it result in $r1 == 1$ and $r2 == 0$?

Figure 9: Constructor Test

6.4 Subtle Mistakes in JMM_{MP}

Using our verification approach, several subtle yet critical specification mistakes in JMM_{MP} are revealed.

6.4.1 Non-Atomic Volatile Writes

One of the proposed requirements for non-atomic volatile write semantics is that if a thread t has observed the new value of a volatile write, it can no longer observe the previous value. In order to implement this requirement, a special flag $readThisVolatile_{t,\langle w,info_t\rangle}$ is initialized to $false$ in initVolatileWrite [3, Figure 14]. When the new volatile value is observed in readVolatile, this flag should be set to $true$ to prevent the previous value from being observed again by the same thread. However, this critical step is missing and the flag is never set to $true$ in the original proposal. This mistake causes inconsistency between the formal specification and the intended goal.

6.4.2 Final Semantics

A design flaw for final variable semantics has also been discovered. This is about a corner case in the constructor that initializes a final variable. The scenario is illustrated in Figure 10. After the final field a is initialized, it is read by a local variable in the same constructor. The readFinal definition [3, Figure 15] would allow r to read back the default value of a. This is because at that time a has not been "synchronized" to be known to the object that it has been frozen. But the readFinal action only checks that information from the kF set which is associated with the object reference. This scenario compromises the program correctness because data dependency is violated.

```
class foo {
    final int a;

    public foo() {
        int r;
        a = 1;
        r = a;
        // can r == 0?
    }
}
```

Figure 10: Flaw in final variable semantics

7 Conclusions

As discussed in earlier sections, the importance of a clear and formal JMM specification is being increasingly realized. In this paper we have presented a uniform specification framework for language level memory models. This permits us to conduct formal analysis and pave the way towards future studies on compiler optimization techniques in a multithreaded environment. Comparing to traditional specification frameworks, UMM has several noticeable advantages.

1. It provides strong support for formal verification. This is accomplished by using an operational approach to describe memory activities, enabling the transition system to be easily integrated with a model checking tool. Formal methods can help one to better understand the subtleties of the model by detecting some corner cases which would be very hard to find through traditional simulation techniques. Because the specification is executable, the memory model can be provided to the users as a "black box" and the users are not necessarily required to understand all the details of the memory model. In addition, the mathematical rules in the transition table makes the specification more rigorous, which eliminates any ambiguities.

2. UMM addresses the special need from a language level memory model by reducing the gap between memory semantics and program semantics. This enables one to study the memory model implications in the context of data flow analysis. It offers the programmers, compiler writers, and hardware designers an end-to-end view of the memory consistency requirement.

3. The model is flexible enough to enforce most desired memory properties. Many existing memory models are specified in different notations and styles. This is due to the fact that the specification is often influenced by the actual architecture of its implementation and there lacks a uniform system that is flexible enough to describe all properties in a shared memory system. In UMM, any completed instructions that may have any future visibility effects are stored in the global instruction buffer along with the time stamps of their occurrence. This allows one to plug in different selection algorithms to observe the state. In a contrast to most processor level memory models that use a fixed size main memory, UMM applies a global instruction buffer whose size may be increased if necessary, which is needed to specify relaxed memory models that require to keep a trace of multiple writes on a variable.

The abstraction mechanism in UMM provides a feasible common design interface for any executable memory model with all the internal data structures and implementation details encapsulated from the user. Different ordering rules and observability rules can be carefully developed in order to enable a user to select from a "menu" of memory properties to assemble a desired formal memory model specification.

4. The architecture of UMM is very simple and intuitive. The devices applied in UMM, such as instruction buffers and arrays, are standard data structures that are easy for one to understand. Similar notations have been used in processor memory model descriptions [16] [23], making this model intuitive to hardware designers. Some traditional frameworks use multiple copies of the shared memory modules to represent non-atomic operations [11]. In UMM, these multiple modules are combined into a single global buffer which substantially simplifies state transitions.

Our approach also has some limitations. Based on the Model Checking techniques, it is exposed to the state explosion problem. Effective abstraction and slicing techniques need to be applied in order to use UMM to verify commercial multithreaded Java programs. Also, our UMM prototype is still under development. The optimal definition for final variables needs to be identified and specified.

A reliable specification framework may lead to many interesting future works. Currently people need to develop the test programs by hand to conduct verifications. To automate this process, programming pattern annotation and recognition techniques can play an important role.

Traditional compilation techniques can be systematically analyzed for JMM compliance. In addition, the UMM framework enables one to explore new optimization opportunities allowed by the relaxed memory consistency requirement.

Architectural memory models can also be specified in UMM. Under the same framework, memory model refinement analysis can be performed to aid the development of efficient JVM implementations.

Finally, we plan to apply UMM to study the various proposals to be put forth by the Java working group in their currently active discussions regarding Java shared memory semantics standardization. The availability of a formal analysis tool during language standardization will provide the ability to evaluate various proposals and foresee pitfalls.

Acknowledgments

We sincerely thank all contributors to the JMM mailing list for their insightful and inspiring discussions for improving the Java Memory Model.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, chapter 17. Addison-Wesley, 1996.

[2] William Pugh. Fixing the Java Memory Model. In *Java Grande*, pages 89–98, 1999.

[3] Jeremy Manson and William Pugh. Semantics of multithreaded Java. Technical report, UMIACS-TR-2001-09.

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

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

[6] The Java Memory Model mailing list. http://www.cs.umd.edu/˜pugh/java/memoryModel/archive.

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

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[9] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[10] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *the 19th International Symposium of Computer Architecture*, pages 13–21, May 1992.

[11] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, CSL-TR-95-685.

[12] Guang Gao and Vivek Sarkar. Location consistency - a new memory model and cache consistency protocol. Technical report, 16, CAPSL, University of Delaware, February, 1998.

[13] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, 1992.

[14] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying system requirements for memory consistency models. Technical report, CSL-TR93-594.

[15] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *the 1993 Symposium for Research on Integrated Systems*, pages 38–52, March 1993.

[16] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. Prentice Hall, 1994.

[17] Seungjoon Park and David L. Dill. An executable specification and verifier for Relaxed Memory Order. *IEEE Transactions on Computers*, 48(2):227–235, 1999.

[18] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Analyzing the CRF Java Memory Model. In *the 8th Asia-Pacific Software Engineering Conference*, pages 21–28, 2001.

[19] Abhik Roychoudhury and Tulika Mitra. Specifying multithreaded Java semantics for program verification. In *International Conference on Software Engineering*, 2002.

[20] A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations for Java memory behavior. In *the Workshop on Java for High-Performance Computing, Rhodes*, June 1999.

[21] Philip Bishop and Nigel Warren. *Java in Pratice: Design Styles and Idioms for Effective Java*, chapter 9. Addison-Wesley, 1999.

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

[23] Rob Gerth. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing*, 1995.

[24] David Dill. The Murφ verification system. In *8th International Conference on Computer Aided Verification*, pages 390–393, 1996.

Appendix: Equivalence Proof

Let the multithreaded Java semantics specified in Section 5 be referred as JMM_{UMM}. We present the equivalence proof between JMM_{UMM} and JMM_{MP} based on our straightforward implementation using the bypassing table shown in Table 2. For the sake of brevity, we only outline the equivalence proof for the core subset of the memory model including instructions ReadNormal, WriteNormal, Lock, and Unlock.

JMM_{UMM} and JMM_{MP} are *equivalent* if and only if the execution traces allowed by both models are the same. This is proven with two lemmas. We first demonstrate that both models impose the same ordering restrictions for issuing instructions within each thread. We then prove that the legal values resulted from the ReadNormal instruction in JMM_{UMM} is both sound and complete with respect to JMM_{MP}.

Lemma 1 *Instructions in each thread are issued under the same ordering rules by JMM_{UMM} and JMM_{MP}.*

Since prescient writes are the only ordering relaxation explicitly allowed by both models, it is sufficient to prove that the ordering requirement on prescient writes are the same.

1. Soundness of JMM_{UMM}: let w be any WriteNormal instruction allowed by JMM_{UMM}, we show that it must satisfy the conditions in JMM_{MP}, which is enforced by the assertion $w \notin previousReads_t$ in performWrite. There are only two ways to add w to $previousReads_t$.

   (a) *w is read from the same thread before where w would normally occur.*

   This can not happen in JMM_{UMM}. Because a write instruction w can not bypass a previous read instruction r issued by the same thread if they operate on the same variable, r would never observe a later write instruction from the same thread.

   (b) *w is added to $previousReads_{t'}$ by another thread t' and then synchronized to thread $t(r)$ before r is issued.*

   To make this happen, there must exists an acquire operation in $t(r)$ that happens between where w is issued and where w would normally occur. This is not allowed in JMM_{UMM} since w is not allowed to bypass a previous acquire operation.

2. Completeness of JMM_{UMM}: let w be any normal write instruction allowed by JMM_{MP}, we prove it is also permitted by JMM_{UMM}.

   Assume w is prohibited by JMM_{UMM}. According to conditions of the WriteNormal instruction in the transition table Table 1, w can only be prohibited when $ready(w) = false$. Therefore, w must have bypassed a previous instruction prohibited by the BYPASS table. The only reordering that is forbidden for a normal write instruction is the bypassing of a previous lock instruction or a previous read instruction operated on the same variable. The former case is prohibited in JMM_{MP} by the mutual exclusion requirement of a lock instruction. The latter case is also forbidden by JMM_{MP} because the assertion in performWrite would have failed if a readNormal instruction were allowed to obtain a value from a later write instruction in the same thread.

Lemma 2 *The normal read instructions from both models generate the same legal results.*

1. Soundness of JMM_{UMM}: if w is a legal result for a read instruction r under JMM_{UMM}, w is also legal in JMM_{MP}.

   We prove that w satisfies all requirements according to the definition of the readNormal operation defined in JMM_{MP}:

$$\begin{array}{lll} \text{readNormal(Variable } v) & \Longleftrightarrow & \text{Choose } \langle v, w, g \rangle \text{ from allWrites}(v) \\ & & \qquad - \text{ uncommitted}_t - \text{overwritten}_t \\ & & \text{previousReads}_t \mathrel{+}= \langle v, w, g \rangle \\ & & \text{return } w \end{array}$$

(a) *The result is from allWrites(v)*

This requirement is guaranteed by $var(w) = var(r)$ and $op(w) =$ WriteNormal in condition $legalNormalWrite(r, w)$.

(b) $w \notin$ *uncommitted*$_t$

Assume $w \in$ uncommitted$_t$. To make this happen, w must be a write instruction that follows r in program order but is observed by r. This is prohibited by JMM$_{\text{UMM}}$ because w is not allowed to bypass r in this situation.

(c) $w \notin$ *overwritten*$_t$

Assume $w \in$ overwritten$_t$. w can only be added to overwritten$_t$ in two ways.

i. *A write w', which is on the same variable and from the same thread, is performed with its corresponding* performWrite *operation. And* $w \in$ *previous*$_t$ *at that time.*

In order to have w exist in previous$_t$ when w' is performed, w' must be performed after w is performed. Therefore, w is not the most recent write, which is illegal according to condition $legalNormalWrite(r, w)$.

ii. *w is added to overwritten$_{t'}$ by another thread t' and later acquired by $t(r)$ via synchronization.*

w can only be added to overwritten$_{t'}$ by thread t' when t' performs another write w' on the same variable and w has been added to previous$_{t'}$ at that time. Furthermore, this must occur before the release operation issued in t' which is eventually acquired by t. Therefore, $LCOrder(w', w) = true$ and $LCOrder(r, w') = true$, which is prohibited in function $legalNormalWrite(r, w)$ by JMM$_{\text{UMM}}$.

2. Completeness of JMM$_{\text{UMM}}$: if w is a legal result for a read instruction r in JMM$_{\text{MP}}$, w is also legal in JMM$_{\text{UMM}}$.

Assume w is prohibited by JMM$_{\text{UMM}}$. According to the conditions for the readNormal event in Table 1, one of the following reasons must be true.

(a) $ready(r) = false$

This indicates that there exists at least one pending instruction i in the same thread such that i precedes r. Because JMM$_{\text{MP}}$ does not issue a read instruction out of program order, this scenario would not occur in JMM$_{\text{MP}}$ either.

(b) $legalNormalWrite(r, w) = false$

Condition $legalNormalWrite(r, w)$ only fails when w is not the most recent previous write on the same variable in a path of a sequence of partially ordered writes according to $LCOrder$. This is also forbidden by JMM$_{\text{MP}}$.