Formal Reasoning about Hardware and Software Memory Models[⋆]

Abhik Roychoudhury

Department of of Computer Science, School of Computing,
S16 Level 5, 3 Science Drive 2,
National University of Singapore, Singapore 117543.
`abhik@comp.nus.edu.sg`

Abstract. The Java programming language allows multithreaded programming, where threads can be run on multiprocessor or uniprocessor platforms. The allowed behaviors of any multithreaded Java program on any implementation platform (multi- or uni-processor), are described in terms of a memory consistency model called the Java Memory Model (JMM). However, shared memory multiprocessors have a memory model of their own. To reason about the behavior of multithreaded Java programs on multiprocessors, we need a formal basis for understanding both the hardware memory model (of the multiprocessor platform) and the software memory model (the JMM). For this purpose, we have implemented formal executable specifications of the JMM and certain hardware memory models (such as TSO/PSO from SPARC). These executable specifications can be used for exhaustive search *i.e.* computing *all allowed behaviors* of test programs under the JMM and the hardware memory models. Consequently, we can compare the JMM with the hardware memory models (in terms of allowed behaviors). We show that such a comparison can help efficient and reliable multithreaded programming on multiprocessors. Results from comparing the current JMM with SPARC architecture memory models are presented.

1 Introduction

Memory consistency models have been used in shared-memory multiprocessors for many years. Given a number of processes accessing a shared store, a memory consistency model places restrictions on the order in which the processes can access (read/write) the shared store. This effectively restricts the values that can be returned on the read of a shared variable, and thereby provides a model of execution to the programmer. The simplest model of memory consistency was proposed by Lamport, and is called Sequential Consistency [11]. This model allows operations across threads to be interleaved in any order. Operations within each thread are however constrained to proceed in *program order*. For example, in the following multithreaded program initially we have `u = v = 0`.

[⋆] This work was partially supported by National University of Singapore Research Project R-252-000-095-112.

```
u:=1      x:=v
v:=1      y:=u
```

Then, a Sequentially Consistent execution of this program cannot return `x = 1`, `y=0`. This is possible if the writes to `u`, `v` are re-ordered.

Sequential consistency serves as a very simple and intuitive model of execution to the programmer. However, it disallows most compiler and hardware optimizations. For this reason, shared memory multiprocessors have employed *relaxed memory models* [2]. Examples of relaxed memory models include Total Store Order (TSO), Partial Store Order (PSO) and Relaxed Memory Order (RMO) in Sun SPARC architectures [6]. These memory models allow certain reordering of operations within a process/thread and allow more behaviors than Sequential Consistency *e.g.* `x=1,y=0` in the above example is allowed under SPARC PSO and RMO models. This complicates the programming model at the cost of increased execution efficiency. Thus, people writing multithreaded programs for a shared-memory multiprocessor platform view the hardware memory model as an abstract description of the behaviors supported by the system.

Two recent developments have significantly increased the importance of multithreaded program usage on shared memory multiprocessors. First of all, the widespread use of commercial Symmetric Multiprocessors (SMP) clusters [3] has given shared memory parallelism new life. Secondly, multithreading has been integrated as a key feature of the popular Java programming language. Java supports multithreaded programming, where multiple threads can communicate via read/write of shared objects. These threads can then run on a single processor via a thread library, or on hardware multiprocessors.

Problem addressed Execution of multithreaded Java programs on shared memory multiprocessors introduces a *new problem.* The semantics of multithreaded Java is given by a language level memory model, called the Java Memory model (henceforth called JMM) [9]. As in hardware memory models, the JMM is a set of abstract rules dictating the allowed ordering of read/write of shared variables. Any uniprocessor/multiprocessor implementation of Java multithreading must respect the JMM. The JMM is the first serious attempt to introduce a memory model at the language/software level. In this paper, we compare the behaviors allowed by the JMM with the behaviors allowed by hardware memory consistency models.

Typically, memory models are given as a set of abstract rules. A comparison of models M_1 and M_2 would proceed via human reasoning about which re-orderings are allowed by M_1 but disallowed by M_2 (and vice-versa). This approach is completely informal and extremely error-prone. In this paper, we advocate the use of formal specification and checking techniques for this purpose. In [16], we have developed a formal executable specification of the current JMM, while specification for hardware memory models have been developed in [8, 14]. In this paper, we use these formal specifications for comparing hardware and software memory models. In particular, we craft test programs (using our informal understanding of the memory models), and then use the formal specifications to automatically generate all possible observed behaviors of the test programs under the two memory models. Thus, our method extends informal reasoning with formal specification and verification techniques.

Motivation There are several reasons for studying such a comparison between software and hardware memory models. If the hardware memory model is weaker than the JMM (*i.e* allows more re-orderings than the JMM) then the Java Virtual Machine needs to insert memory barriers in the hardware instruction sequence. A memory barrier [5] prevents re-ordering of operations across the barrier and can make the multiprocessor execution comply to the JMM. Inserting these time-expensive memory barriers without understanding the JMM and hardware memory models can introduce unacceptable performance overheads. If all re-orderings allowed by the hardware memory model are also allowed by the JMM, then the JVM need not insert any memory barriers.

Comparing the JMM with hardware memory models is also useful for reasoning about low-level unsynchronized Java programs. Often, low-level libraries do not require a synchronization (i.e. lock acquisition) for every shared variable access. Examples of such programs include popular multithreaded Java software construction idioms *e.g.* the “Double-Checked Locking” idiom [17]. These programs allow different sets of behaviors on different multiprocessor platforms (with different memory models). We can formally reason about such low-level code by studying the memory models.

Organization The rest of this paper is organized as follows. Section 2 recapitulates salient features of the Java Memory Model (JMM). Section 3 introduces the SPARC memory models, and our checker for these models. Section 4 outlines our methodology for studying the relationship between hardware and software memory models. Finally, section 5 describes the related work and conclusions.

2 A Checker for Java Memory Model

The Java programming language allows the user to write multithreaded programs. Java threads interact among themselves via shared variables. For any shared variable v, each thread (a) possesses a local copy of v and (b) is allowed to access the global master copy of v in main memory. The Java Memory Model (JMM) essentially imposes constraints on the interaction of the threads with the master copy of the variables and thus with each other. The model defines the following *actions* for reading/writing the local/master copy of v in thread t.

- $\texttt{use}_t(v)$: Read from the local copy of v in t
- $\texttt{assign}_t(v)$: Write into the local copy of v in t
- $\texttt{read}_t(v)$: Initiate reading from master copy of v to local copy of v in t.
- $\texttt{load}_t(v)$: Complete reading from master copy of v to local copy of v in t.
- $\texttt{store}_t(v)$: Initiate writing the local copy of v in t into master copy of v
- $\texttt{write}_t(v)$: Complete writing the local copy of v in t into master copy of v

Apart from the above actions, each thread t may perform lock/unlock on shared variables, denoted $\texttt{lock}_t$ and $\texttt{unlock}_t$ respectively.

Among the eight actions mentioned above, a thread in a Java program invokes only four of them: `use`, `assign`, `lock`, and `unlock`. Each thread invokes these actions in its program order. The other four (`load`, `store`, `read`, and `write`) are invoked arbitrarily by the multithreading implementation, subject to *temporal ordering constraints* specified in the JMM. For example, let the program running in a thread be `assign u, 1`; `assign v, 2` (a sequence of two writes). Then, the two `assign` statements are executed in program order. However, this does not affect the master copy of the shared variables `u`, `v`. The master copy is updated based on `store`/`write` actions. These are issued in any order which preserves the temporal ordering constraints specified in the JMM. For example, the JMM allows the following execution `assign u,1; assign v,2; store v,2; write v,2; store u,1; write u,1` where writes to `u`, `v` are completed out of order.

A major difficulty in reasoning about the JMM (as reported in [15]) lies in these ordering constraints. They are given in an informal, rule-based, declarative style. It is difficult to reason how multiple rules determine the applicability/non-applicability of an action. Our formal operational specification (presented in [16]) avoids this difficulty by modeling each action as a guarded command. We present a brief overview of this work below.

We model each action as a guarded command of the form $\mathcal{G} \rightarrow \mathcal{B}$, where the guard $\mathcal{G}$ is first evaluated; if $\mathcal{G}$ is true, then the body $\mathcal{B}$ is executed atomically. Our model is an asynchronous concurrent composition of n Java threads $\texttt{Th}_1, \ldots, \texttt{Th}_n$ and a single main memory process `MM`. Communication among processes takes place via shared data. Each process can perform a set of *actions*, each of which is modeled by a guarded command. The asynchronous concurrent composition of these processes is the union of the guarded commands of the constituent processes. At any time step, the processes $\texttt{Th}_i$ and `MM` can execute either a *program action* or a *platform action*. In particular, an action invoked by the program running as thread $\texttt{Th}_i$ is called a program action. The actions $\texttt{use}_i$, $\texttt{assign}_i$, $\texttt{lock}_i$, and $\texttt{unlock}_i$ are program actions. On the other hand, an action which is performed by the underlying multithreading implementation is called a platform action. The actions $\texttt{load}_i$, $\texttt{store}_i$, $\texttt{read}_i$, and $\texttt{write}_i$ are platform actions. Typically, the purpose of executing platform actions is to enable those program actions which are currently disabled.

Since our model is expressed in guarded-command notation, the Murφ model checker [7] is a candidate implementation vehicle[1]. However, we want to program the traversal strategy of the search space of multithreaded executions (for efficient checking and validation). This programming capability is very naturally supported in a general purpose logic programming system where computation proceeds by search. A prototype checker based on our executable memory model has been built using XSB, a memo-table based logic programming system [18]. The checker could be used in two modes. Either we could search the entire search space consisting of all allowed execution traces of program actions and platform

[1] Murφ supports a guarded-command based specification language

actions in the threads of a program; or we could input rules to prune the search space based on some scheduling algorithm.

3 A Checker for SPARC Memory Models

In this section, we give a brief overview of memory models appearing in hardware multiprocessors, in particular SPARC memory models. Based on formal executable specification of these memory models [8, 14], we have developed an invariant checker. This checker can be used to verify invariants or to generate all possible behaviors of low-level SPARC code.

The SPARC multiprocessor architecture defines three different memory models: Total Store Order (TSO), Partial Store Order (PSO) and Relaxed Memory Order (RMO) [6]. In terms of allowed behaviors $TSO \subseteq PSO \subseteq RMO$. That is, for any program P, the set of possible behaviors of P under TSO (PSO) is included in the set of possible behaviors of P under PSO (RMO). In describing each of the memory models, we assume that the instructions in each processor are issued in program order. However, these instructions may be completed out of order. Thus, each of these memory models are *weaker* than Sequential Consistency [11] where instructions must always be completed in the order in which they are issued. In the subsequent discussions we denote `ld` to denote a memory read instruction and `st` to denote a memory write instruction. Furthermore, note that all the memory models allow only those re-orderings which do not violate the data-flow dependencies in a processor *e.g.* the sequence `st u`; `ld u` can never be completed out-of-order since this would change the value of `u` read by `ld`.

In the TSO memory model, the restrictions are as follows: (a) a `ld` operation is ordered with respect to subsequent `ld` and `st` operations (b) a `st` operation is ordered with respect to subsequent `st` operations. Thus this execution model allows a sequence of instructions `st u`; `ld v` (a write to variable `u`, followed by a read of variable `v`) to be completed out-of-order. The PSO memory model relaxes the TSO model by removing the second restriction of TSO. Thus, two write operations to different variables `st u`; `st v` may be executed out-of-order. The RMO model relaxes PSO by removing both the restrictions of TSO.

An executable model of a shared memory multiprocessor system considers each of the processors as well as the shared memory as a separate process. The combined model is the asynchronous concurrent composition of these processes. Given a parallel program $P_1 \parallel \ldots \parallel P_k$ which is run on processors $Proc_1, \ldots, Proc_k$, processor $Proc_i$ executes the instructions in P_i in program order. The effect of the memory model (delaying/re-ordering of certain instructions) is captured in the following manner. Any processor $Proc_i$ maintains a buffer of incomplete instructions called the *Store buffer*, denoted SB_i. These are non-blocking instructions, *i.e.* instructions which were issued but which have not been completed. The shared memory process can then complete any instruction in any store buffer SB_i in a manner consistent with the memory model.

As a concrete example, let us consider the TSO memory model. It allows writes (`st` instructions) to be delayed, but not re-ordered. Given a multiprocessor program $P_1 \parallel \ldots \parallel P_k$, any processor $Proc_i$ issues instructions of P_i in program order as follows:

- a `st` instruction (write operation) is appended into the store buffer SB_i,
- a `ld u` instruction (reading some variable `u`) executes as follows. If SB_i contains incomplete writes to `u` then the value of the last incomplete write to `u` in SB_i is returned. Otherwise, the value of `u` is read from the memory.
- all other instructions (corresponding to computation) proceed as usual.

Concurrently, the shared memory process is allowed to complete the first instruction of any of the k store buffers $SB_1 \ldots SB_k$. This corresponds to the delaying of writes as allowed in TSO. To model the re-ordering of writes as allowed in PSO, the shared memory process can be allowed to complete a write instruction in SB_i which is not the first instruction in SB_i.

Based on these executable specifications, we have implemented a checker for the TSO and PSO memory models from SPARC. Given a multiprocessor program, it can generate all observed behaviors under TSO/PSO by computing the set of reachable states. Therefore, it can be used for model checking [4] of invariant properties *i.e.* checking whether an invariant holds in all reachable states. We have used the checker to automatically verify invariants in low-level SPARC code such as verifying mutual exclusion in implementation of spin locks (a lock where the check for whether lock is acquired is done by busy waiting), Dekker's algorithm etc. Many of these code fragments are available in the SPARC architecture manual [6]. However, at this point we move away from the description of the JMM/SPARC checkers and concentrate on how to use these checkers for comparing software/hardware memory models.

4 Relationship between Memory Models

In this section, we present our methodology for comparing the current JMM with hardware memory models. As a concrete example, we consider the SPARC memory models which were discussed in the last section. We show that such a comparison is useful for (a) obtaining efficient multithreading implementation on multiprocessors, and (b) reasoning about low-level unsynchronized multithreaded Java programs.

4.1 Avoiding Redundant Memory Barriers

We want to find out which hardware memory models are stronger than the JMM. A JVM can then execute without introducing *memory barriers* [5] on all such multiprocessor platforms. Recall that a memory barrier is a time-expensive hardware instruction such that in any code if a memory barrier I appears between instructions I_1 and I_2, instruction I_1 must complete before I_2 begins. To clarify the different components of a multiprocessor implementation of Java multithreading, refer to Figure 1. The JVM implementer needs to ensure that reorderings caused by the multiprocessor platform do not violate the Java Memory Model.

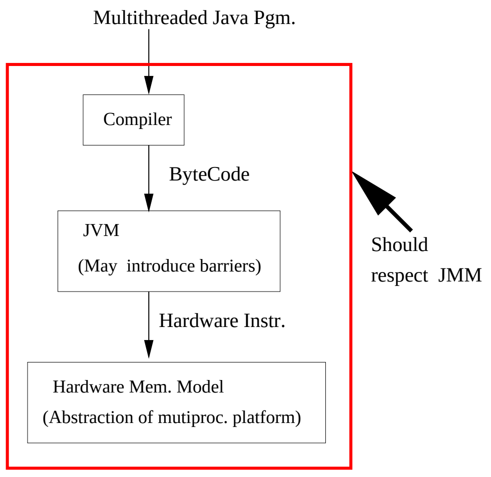

Fig. 1. Multiprocessor Implementation of Java Multithreading

Our methodology for comparing hardware and software memory models consists of the following steps:

1. Develop formal executable specifications of both the software and the hardware memory models M_s and M_h.
2. Select one/several terminating test program(s) P to expose the re-orderings allowed by the hardware memory model M_h. These test programs are obtained by our *informal* understanding of which re-orderings allowed by M_h.
3. Use the executable specifications of (1) to perform exhaustive state space exploration (as in model checking of invariants) of (a) P executed on M_s and (b) P executed on M_h. This step performs automated *formal* reasoning.
4. Check the set of possible values of the data variables of P on termination in the two cases. This allows us to compare the allowed behaviors of M_h and M_s.

Comparing JMM with TSO As an illustrative example, let us compare the current JMM with the SPARC TSO memory model. In the TSO model, re-orderings of the following are not allowed: `ld` $\rightarrow$ `ld`/`st`, `st` $\rightarrow$ `st`. Only the re-ordering of `st` $\rightarrow$ `ld` is allowed. Thus, if `st u` appears before `ld v` in program order, they may be completed out of order.

To capture the delayed completion of stores allowed by TSO, consider the test program in Figure 2. It forms the heart of Dekker's algorithm. Assuming sequential consistency, we verify that the values of `reg1` and `reg2` at the end of execution must satisfy `reg1 = 1` $\vee$ `reg2 = 1`. This is because in any sequentially consistent execution at least one of the loads will be executed last; this load will return 1 (instead of 0). We then use our implementation of the executable model of TSO to find all possible behaviors allowed by TSO. We find that under

Initially: `u = v = 0`

```
Thread 1        Thread 2
st u, 1         st v, 1
ld v, reg1      ld u, reg2
```

Seq. Consistency: `reg1 = 1` $\vee$ `reg2 = 1`

TSO : `reg1` = 0 or 1, `reg2` = 0 or 1

Fig. 2. Test program showing delayed completion of writes

TSO there is one *additional* behavior: `reg1` = `reg2` = `0`. This is possible only because both the `ld` instructions may complete ahead of the `st` instructions.[2]

We now want to check whether the Java Memory Model (JMM) allows delayed completion of writes as in TSO. In particular, we want to perform exhaustive state space exploration of the test program in Figure 2 to find out all possible values of `reg1`, `reg2` allowed by JMM. To do so, we must express each of the threads as a sequence of JMM actions (*i.e.* `use`, `assign`, `lock`, `unlock`). Note that in the JMM, `assign` denotes the beginning of a write operation, whereas `use` denotes the end of a read operation. On the other hand, in weak memory models, `ld`/`st` denote the beginning of read/write operations; these operations may complete out-of-order. Thus, we can map a `st` instruction to an `assign` action (both denote beginning of a write). Also, we can map a `ld` instruction to a `use` action provided the `ld` is blocking (which is the case in TSO and PSO models). Consequently, the test program of Figure 2 is reduced to

```
assign u,1      assign v,1
use v           use u
```

We use our JMM checker to find *all* possible values of `u` and `v` in the local copies of the two threads. Our checker shows that the local copy of `v` in thread 1 as well as the local copy of `u` in thread 2 can be 0 or 1 at the end of execution. This shows that the JMM allows delayed completion of writes as in TSO.

In a similar fashion we have compared the JMM with other memory models such as SPARC PSO. We find that JMM also allows the re-ordering of writes which are allowed in PSO (and not in TSO). Since both delaying and re-ordering of writes are allowed by JMM, therefore a JVM can execute on TSO/PSO memory models without inserting memory barriers. Such a conclusion cannot be reached for the RMO memory model, for which memory barriers will be needed.

A crucial step in the above methodology is the selection of the terminating test programs, which is done informally. However, we can formally check whether a set of selected test programs is complete with respect to the re-orderings allowed by a given hardware memory model M_h. In particular, for each of the

[2] In fact, to ensure that Dekker's algorithm works correctly on TSO, a memory barrier instruction needs to be inserted. This is however orthogonal to the memory model comparison we discuss here.

re-orderings allowed by M_h we construct a separate terminating test program. Thus, for the PSO model which allows the re-orderings `st` $\rightarrow$ `ld` and `st` $\rightarrow$ `st`, we will construct two separate test programs. To check whether a given test program P exercises a given re-ordering r, we can simply use exhaustive state space exploration. Each test program P comes with a set of observable variables $\overline{V}$ whose values we will observe on termination. We then exhaustively check all possible values of $\overline{V}$ when P is executed under (a) sequential consistency (b) relaxation of sequential consistency with re-ordering r enabled.[3] If the two sets of possible values of $\overline{V}$ are different, then we can conclude that test program P can be used to check the presence/absence of re-ordering r.

4.2 Reasoning about Unsynchronized programs

We have discussed how the formal executable specification of hardware/software memory models can help avoid time-consuming barrier instructions in multi-threaded program execution. We now discuss how it can be used to analyze the behavior of low-level unsynchronized Java code. Typically, most Java programs are "*properly synchronized*", that is, locks are acquired before any shared variable access. Since synchronization (acquiring/releasing of locks) is an time-consuming operation, low-level libraries sometimes avoid synchronization. In these programs, the programmers avoid synchronization with the assumption of sequential consistency *i.e.* they assume that the program will behave expectedly if the underlying platform is sequentially consistent. Unfortunately, the current JMM (as well as any future improvements) is (will be) weaker than Sequential Consistency.[4] Hence, it is necessary to incorporate an executable specification of JMM into verification of unsynchronized Java programs [16].

Initially `A = B = 0`

Thread 1	Thread 2
`A = 1;`	`if (B == 1)`
`B = 1;`	`C = A`

Fig. 3. An example of Unsynchronized Code

Verifying unsynchronized code with respect to the JMM is not enough. Consider the simple example in Figure 3 where `A` and `B` are shared variables. This program is unsynchronized since shared variables `A`, `B` are read/written without acquiring locks. The programmer will however expect the value of `C` to be `1` on

[3] The underlying model allows a re-ordering $r = a \rightarrow b$ if it generates *more* behaviors by allowing an operation of type b to bypass an operation of type a.

[4] This is because the JMM represents all possible program behaviors on all possible platforms: uni- and multi-processor; see [1] for potential revisions to the current JMM.

termination. This arises from the programmer's expectation of Sequential Consistency: each thread is expected to proceed in program order. By considering the formal executable specification of the JMM we can conclude that `C` can be `0` or `1`. This is because JMM allows the writes to `A` and `B` to be re-ordered. However, this merely means that the returned value of `C` may be `0` or `1` on certain (not all) implementations. Uni-processor implementations guarantee Sequential Consistency. To find out which multi-processor platforms allow `C` to be both `0` or `1` we need to consider the hardware memory model of the platform in question.

Methodology To reason about behaviors of an unsynchronized program P we first use our JMM checker. This returns *all possible behaviors* of P in any implementation of Java multithreading. If some of these behaviors are marked by the programmer as "*undesirable*", then we can find whether these undesirable behaviors appear in the specific multiprocessor platform(s) we are interested in. This is because all the behaviors allowed by the JMM may not be manifested on all platforms. To find whether a specific "undesirable behavior" appears in a given multiprocessor platform, we then use the checker for the corresponding hardware memory model.

For example, for the program in figure 3, we first use our JMM checker to find that the returned value of `C` can be `0` or `1`. However, if we convert the program to SPARC instructions and check whether `C` can be `0` in the TSO model, our TSO checker returns "no". This is because TSO allows writes to be delayed, not re-ordered. By executing writes to `A`, `B` in program order we must return `C = 1`. We then use our PSO checker to find whether `C` can be `0` in PSO implementations. The PSO checker returns "yes" since PSO allows write re-ordering.

An Example We have used our JMM and TSO/PSO checkers to find allowed behaviors of a widely-known multithreaded program fragment: the "Double Checked Locking" idiom.[5] This program fragment (shown in Figure 4) is used for efficient lazy instantiation of a singleton class (a class with only one instance) by multiple threads. Any thread which invokes `getInstance` executes the code in Figure 4. If `instance` is `null` (*i.e.*, an instance of `Singleton` class has not yet been created), then the code forces synchronization and checks whether `instance` is `null` again within the critical section. In between the first `instance == null` check and the synchronization, another thread may invoke `getInstance`, find that `instance` is `null`, and then create an instance of the `Singleton` class. Hence we need the second `instance == null` check.

Our JMM checker finds *all possible behaviors* of two threads running the Double Checked Locking program in less than a second. This set of allowed behaviors (which are generated by our JMM checker) includes an execution trace in which a singleton object with garbage datafields is returned by a thread. This amounts to an instantiation routine returning a partially instantiated object. The question now is whether this undesirable behavior is manifested in all multiprocessor platforms. We verify the absence of this undesirable behavior in the

[5] See [10, 17] for a discussion of its use, and [15, 16] for a detailed explanation

```
private static Singleton instance = null;
....    // the other fields
public static Singleton getInstance()
{
   if (instance == null){
       synchronized (Singleton.class) {
          if (instance == null)
             instance = new Singleton();
       }
   }
   return instance;
}
```

Fig. 4. Double-Checked Locking

TSO model using our TSO checker in 0.01 seconds. On the other hand, our PSO checker detects in 0.02 seconds that this undesirable behavior is allowed in the PSO memory model. All experiments were conducted on a Pentium-4 1.3 GHz workstation with 1 GB of memory.

5 Discussions

In this paper, we employed formal specification and verification techniques to study the relationship between hardware and software memory models. Hardware memory models describe the behaviors allowed by multiprocessor implementations, while software/language level memory models (such as the JMM) describe behaviors allowed by multithreading implementations. Efficient and reliable execution of multithreaded programs on multiprocessors is the main motivation of our study on memory models. We showed how a formal understanding of the memory models can (a) avoid unnecessary memory barrier instruction executions (leading to higher efficiency) and (b) allow reasoning about low-level multithreaded code (leading to higher reliability). To the best of our knowledge, such an comparison between hardware and software memory models has not been studied formally. Performance/reliability issues in running Java on multiprocessor architectures have been informally discussed in [15]. We have used formal executable specification of the hardware/software memory models to compare their allowed behaviors. Our comparison employs state space exploration (as in model checking) to compute all possible behaviors of programs on two different memory models.

Note that in this paper we chose the current Java Memory Model (JMM) as the software memory model and the SPARC TSO/PSO as hardware memory models. However our methodology for comparing the behaviors of software/hardware memory models is *not* restricted to this choice. The JMM is currently undergoing revision by an expert group [1] and formal/informal specifications are being developed for the various candidates for the revised JMM [12, 13, 19]. Once the JMM revision is finalized, we plan to perform a full-fledged comparison of the revised JMM with various existing multiprocessor memory models (SPARC TSO, SPARC PSO, DEC Alpha, IBM 370 etc) using the methodology presented in this paper.

References

1. Java Specification Request (JSR) 133. Java Memory Model and Thread Specification revision. In `http://jcp.org/jsr/detail/133.jsp`, 2001.
2. S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, December 1996.
3. A. Charlesworth. Starfire: Extending the SMP envelope. *IEEE Micro*, 1998.
4. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.
5. D.E. Culler and J. Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.
6. D.L. Weaver and T. Germond, Prentice Hall Publishers. *The SPARC Architecture Manual : Version 9*, 1994.
7. D. L. Dill. The Murφ verification system. In *Computer Aided Verification (CAV), LNCS 1102*, 1996.
8. D.L. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*. MIT Press, 1993.
9. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Chapter 17, Addison Wesley, 1996.
10. A. Holub. *Taming Java Threads*. Berkeley CA, APress, 2000.
11. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.
12. J. Maessen, Arvind, and X. Shen. Improving the Java Memory Model using CRF. In *ACM OOPSLA*, 2000.
13. J. Manson and W. Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, 2001.
14. S. Park and D.L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2), 1999.
15. W. Pugh. Fixing the Java Memory Model. In *ACM Java Grande Conference*, 1999.
16. A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 2002.
17. D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *3rd Annual Pattern Languages of Program Design conference*, 1996.
18. XSB. The XSB logic programming system v2.2, 2000. Available for downloading from `http://xsb.sourceforge.net/`.
19. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Formalizing the Java Memory Model for multithreaded program correctness and optimization. Technical Report UUCS-02-011, University of Utah, Department of Computer Science, 2002.