Impact of Java Memory Model on Out-of-Order Multiprocessors

Tulika Mitra Abhik Roychoudhury Qinghua Shen
School of Computing
National University of Singapore
{tulika,abhik,shenqing}@comp.nus.edu.sg

Abstract

The semantics of Java multithreading dictates all possible behaviors that a multithreaded Java program can exhibit on any platform. This is called the Java Memory Model (JMM) and describes the allowed reorderings among the memory operations in a thread. However, multiprocessor platforms traditionally have memory consistency models of their own. In this paper, we study the interaction between the JMM and the multiprocessor memory consistency models. In particular, memory barriers may have to be inserted to ensure that the multiprocessor execution of a multithreaded Java program respects the JMM. We study the impact of these additional memory barriers on program performance. Our experimental results indicate that the performance gain achieved by relaxed hardware memory consistency models far exceeds the performance degradation due to the introduction of JMM.

1. Introduction

The Java programming language supports shared memory multithreading where threads can manipulate shared objects. The threads can execute on a single processor or a multiprocessor. However, multiprocessors have different memory consistency models. A memory consistency model defines the values that can be returned on the read of a shared variable, and thereby provides a model of execution to the programmer. The most restrictive memory consistency model is sequential consistency [9] which does not allow any reordering of the memory operations within a thread. Other memory consistency models allow certain reordering of memory operations within a thread to improve performance as long as uniprocessor control/data dependencies are not violated (e.g., a memory read can be re-ordered w.r.t. a memory write as long as they access two different locations). If the reordering within a thread becomes visible to other threads, it may produce "undesirable" results from the programmer's viewpoint. Therefore, for each platform, the programmer needs to explicitly disable certain reorderings to produce the desired result. However, this kind of platform-dependent programming violates Java's *write once, run everywhere* policy. To solve this problem, the Java Language Specification provides its own software memory consistency model. This model should be supported by any implementation of multithreaded Java irrespective of the underlying hardware memory consistency model. This is called the Java Memory Model (henceforth called JMM).

Conceptually, the JMM defines the set of all possible behaviors that a multithreaded Java program can demonstrate on any implementation platform. These possible behaviors are generated by: (a) arbitrary interleaving of the operations from different threads, and (b) certain reordering of memory operations within each thread. Note that only the reordering of *shared* memory operations can change a multithreaded program's output.

The introduction of a memory model at the programming language level raises new challenges. In particular, if the threads are run on multiple processors, then we need to ensure that all the executions respect the JMM. Clearly, if the hardware memory model allows more reorderings than the JMM, then we have to disable them in order to enforce the JMM. The reorderings are disabled by inserting memory barrier instructions (processors execute operations across the barriers in program order) at the cost of degrading performance. As a result, the performance of a multithreaded Java program depends on the subtle interaction of the software and hardware memory models. Even though the relative performance of different hardware memory models is well understood [4, 14], there has been no effort to quantify the performance impact of the interaction between the software and hardware memory models. We seek to fill this gap. We study the effect of enforcing JMM on five different hardware memory models: Sequential Consistency (SC), Total Store Order (TSO), Partial Store Order (PSO), Weak Ordering (WO), and Release Consistency (RC) [2].

We note that the specification of the JMM is currently a topic of intense discussion [8]. An initial JMM was proposed in the Java Language Specification [6]. Subsequently, this was found to be restrictive in allowing various common compiler optimizations, and too hard to understand [16]. Thus, an expert group has been formed to completely revise the JMM, and a concrete proposal has emerged (but not yet been finalized) [15]. In this paper, we study the old and new proposals for JMM and their impact on multi-processor execution of multi-threaded Java programs.

Related Work Some work have been done at the processor level to evaluate the performance of different hardware consistency models. The work of Gharachorloo et al. [4] showed that in a platform with blocking reads and delayed commit of writes, performance is substantially improved by allowing reads to bypass writes. It also showed that SC performs poorly relative to all other models. Pai et al. [14] studied the implementation of SC and RC models on current generation processors with aggressive exploitation of instruction level parallelism (ILP). They found that hardware prefetching and speculative loads dramatically improve the performance of SC. However, the gap between SC and RC depends on the cache write policy and the complexity of the cache-coherence protocol, and in most cases, RC significantly outperforms SC.

Recently there have been many efforts to study software memory models for the Java programming language. These works primarily focus on understanding the allowed behaviors of the JMM. Some work has been done to formalize the old JMM [5, 17]. We have previously developed an operational executable specification of the old JMM in [17]. Also, [22] has used an executable framework called uniform memory model to specify a new JMM developed by Manson and Pugh [12].

To the best of our knowledge, there has been little systematic study to measure the performance impact of JMM on multiprocessor platforms. From this perspective, the recent work by Doug Lea [10] is closest to ours. This work serves as a comprehensive guide for implementing the new JMM (as currently specified by JSR-133). It provides brief background about why various rules in JMM exist and concentrates on the consequences of these rules for compilers and Java virtual machines with respect to instruction reorderings, choice of memory barrier instructions, and atomic operations. It includes a set of recommended recipes for complying to JSR-133. However, no quantitative performance evaluation results are presented.

Contributions Concretely, the contributions/findings of this paper can be summarized as follows.

- We study the interaction of hardware and software memory models and their impact on the performance of multithreaded Java programs running on out-of-order multi-processors. This involves (a) a theoretical study of how a software memory model adds barriers at the hardware level, and (b) experimental evaluation of how much performance degradation occurs due to these added memory barriers.
- Our experimental results indicate that the performance gain achieved by relaxed hardware memory consistency models far exceeds the performance degradation due to the introduction of JMM. That is, given a relaxed hardware model, the performance reduction due to the introduction of a software memory model is typically not substantial. Overall, the differences between the old and new JMMs also do not produce substantial difference in performance in our benchmarks.
- The main feature of JMM that leads to appreciable performance difference in our benchmarks is the treatment of volatile variables[1]; both the old and new JMM provide special semantics for volatile variables. The introduction of volatile variable semantics leads to up to 5% performance degradation in some of the benchmarks. We also note that the semantics of volatile variables is different in the old and new JMM. However this difference in semantics does not translate to substantial difference in performance for our benchmarks.

Section Organization The rest of this paper is organized as follows. In the next section, we review the technical background on hardware memory consistency models and JMMs. Section 3 describes the theoretical methodology to identify the performance effects of JMM; this is done by identifying the memory barriers to be inserted in order to enforce JMM. Section 4 describes the experimental setup for measuring the effects of a software memory model on multiprocessor platforms. Section 5 describes the experimental results obtained from evaluating the performance of multithreaded Java Grande benchmarks under various hardware and software memory models. Discussions and future work appear in Section 6.

2. Background

In this section, we describe the multiprocessor memory consistency models (hardware memory models) as well as the JMMs (software memory models).

2.1. Hardware Memory Models

Memory consistency models have been used in shared-memory multiprocessors for many years. The simplest model of memory consistency was proposed by Lamport and is called Sequential Consistency (SC) [9]. This model allows operations across threads to be interleaved in any order. Operations within each thread are however constrained to proceed in program order. SC serves as a very

1 A variable whose access always leads to access of its master copy.

	2nd operation			
1st operation	Read	Write	Lock	Unlock
Read	WO,RC	WO,RC	RC	
Write	TSO,PSO,WO,RC	PSO,WO,RC	PSO,RC	PSO
Lock				
Unlock	TSO,PSO,RC	PSO,RC	PSO	PSO

Table 1: Reorderings between memory operations for hardware memory models

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, which allow certain reordering of operations within a thread. In this paper, we study the performance impact of enforcing the JMM on four relaxed hardware memory models: Total Store Order (TSO), Partial Store Order (PSO), Weak Ordering (WO) and Release Consistency (RC). Details of these memory models appear in [2, 3]. Note that all the memory models only allow reorderings which do not violate the uniprocessor data/control flow dependencies within a thread.

The TSO and PSO models (supported by SUN SPARC architecture [21]) differ from SC in that they allow memory write operations to be bypassed. By bypassing we mean that even if a write operation is stalled, a succeeding memory operation can execute. Memory read operations, however, are blocking in TSO and PSO. The TSO model only allows reads to bypass previous writes. PSO is a more relaxed model than TSO as it allows both reads and writes to bypass previous writes. Unlike TSO and PSO, the WO and RC models allow non-blocking reads (i.e., reads may be bypassed). However they classify memory operations as data operations (normal reads/writes) and synchronization operations (lock/unlock). Both WO and RC allow the data operations between two synchronization operations to be arbitrarily re-ordered. However, they differ in handling the reorderings between a data operation and a synchronization operation. Both WO and RC maintain the order between a lock and its succeeding data operations as well as unlock and its preceding data operations for proper implementation of synchronization semantics. WO, in addition, maintains the order between a lock and its preceding data operations as well as unlock and its succeeding data operations. RC relaxes these two restrictions.

Table 1 shows the reordering of operations allowed by different hardware memory models. A blank entry indicates that this reordering is not allowed by any memory model. The entries associated with lock and unlock in Table 1 require some explanation. Note that lock involves an atomic read-modify-write operation. Therefore, operations after lock cannot bypass it under TSO and PSO. For WO and RC, on the other hand, lock is identified as a special memory operation and memory reads/writes after a lock are not allowed to bypass it. However, it is possible for lock to bypass previous reads/writes under certain memory models such as PSO and RC. The situation with unlock is different. Unlock is just an atomic write to shared memory location (synchronization variable). Therefore, an unlock can be re-ordered w.r.t. both the preceding and succeeding reads/writes under certain hardware memory models. In particular, note that as PSO does not distinguish unlock as a special operation, it is possible for an unlock to bypass previous writes, which violates the semantics of unlock. This bypassing is prevented by including a memory barrier instruction in the software implementation of unlock routine on a multiprocessor with PSO model (such as SUN SPARC architecture [21]).

2.2. Java Memory Models

We consider two candidate Java Memory models: (a) $\mathrm{JMM}_{\mathrm{old}}$, the old JMM (since outdated) given in the Java Language Specification [6] and (b) a revised JMM developed by Manson and Pugh [12], henceforth called $\mathrm{JMM}_{\mathrm{new}}$. Note that there have been other candidate proposals for a new JMM (such as Maessen, Arvind and Shen's work [11] and Adve's work [1]). Our study can be (and indeed should be) extended to these models as well. However, the purpose of our study is *not* to compare $\mathrm{JMM}_{\mathrm{old}}$ and $\mathrm{JMM}_{\mathrm{new}}$ point-by-point. Instead we seek to evaluate the performance impact of software memory models on multithreaded Java program performance.

JMM_{old} This model is specified in Chapter 17 of The Java Language Specification [6]. It is a set of abstract rules dictating the allowed reorderings of read/write operations of shared variables. The Java threads interact among themselves via shared variables. The JMM essentially imposes ordering constraints on the interaction of the threads with the master copy of the variables and thus with each other. A major difficulty in reasoning about $\mathrm{JMM}_{\mathrm{old}}$ seems to be 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 a reordering. As a result, this framework is hard to understand. In this paper, we use the easy-to-read operational style formal specification developed by us [17] to decide the enabling/disabling of different reorderings.

Reorder?	2nd operation			
1st operation	Read	Write	Lock	Unlock
Read	Yes	Yes	No	No
Write	Yes	Yes	Yes	No
Lock	No	No	No	No
Unlock	Yes	No	No	No

Table 2: Reorderings between memory operations for $\mathrm{JMM}_{\mathrm{old}}$

Reorder?	2nd operation			
1st operation	Read	Write	Lock	Unlock
Read	Yes	Yes	Yes	No
Write	Yes	Yes	Yes	No
Lock	No	No	No	No
Unlock	Yes	Yes	No	No

Table 3: Reorderings between memory operations for $\mathrm{JMM}_{\mathrm{new}}$

Table 2 shows the allowed reorderings for read, write, lock, and unlock operations in $\mathrm{JMM}_{\mathrm{old}}$. In addition $\mathrm{JMM}_{\mathrm{old}}$ provides special treatment to volatile variables that we will describe later.

JMM_{new} The lack of rigor in the specification of $\mathrm{JMM}_{\mathrm{old}}$ has led to some problems. For example, some important compiler optimizations, such as fetch elimination (elimination of a memory read operation if it is preceded by a read/write operation to the same variable), are prohibited in $\mathrm{JMM}_{\mathrm{old}}$ [16]. Therefore, the JMM is currently going through an official revision by an expert group JSR-133 [8]. There have been multiple proposals for revised JMM: $\mathrm{JMM}_{\mathrm{new}}$ by Manson and Pugh (appeared in [12] and subsequently revised further), Maessen et al. [11], and Adve [1]. The JSR-133 expert group is now converging towards a revised JMM by drawing on the concrete proposals. A full fledged discussion on the planned features of the revised JMM appears in [15]. In this paper, we choose $\mathrm{JMM}_{\mathrm{new}}$ as the candidate revised JMM and use its formal executable description given in [23, 22]. Table 3 shows the allowed reorderings for read, write, lock, and unlock operations in $\mathrm{JMM}_{\mathrm{new}}$. Similar sets of allowed reorderings also appear in the reordering table of Doug Lea's cookbook [10]. $\mathrm{JMM}_{\mathrm{new}}$ also provides special treatment for both volatile variables and final fields.

Major Differences Both the JMMs allow arbitrary reordering of shared variable read/write operations. However, some other characteristics distinguish $\mathrm{JMM}_{\mathrm{old}}$ from $\mathrm{JMM}_{\mathrm{new}}$ in terms of performance as follows.

- **Synchronization Operations:** $\mathrm{JMM}_{\mathrm{old}}$ does not allow locks to be re-ordered w.r.t preceding reads and unlocks w.r.t. following writes. $\mathrm{JMM}_{\mathrm{new}}$ relaxes this constraint (compare Tables 2 and 3).
- **Volatile Variables:** A volatile variable is one for which the Java Virtual Machine (JVM) always accesses the shared copy. $\mathrm{JMM}_{\mathrm{old}}$ does not allow reads/writes of volatile variables to be re-ordered among themselves. Thus read/write of volatile variable u cannot bypass a preceding read/write of another volatile variable v. In contrast, $\mathrm{JMM}_{\mathrm{new}}$ simply treats a volatile variable read as the acquire of a lock and volatile variable write as the release of a lock.
- **Final Fields:** $\mathrm{JMM}_{\mathrm{new}}$ has separate semantics for final fields (fields which are written only once, i.e., in the constructor). In particular, it proposes that values written to a final field f of an object within the constructor be visible to reads by other threads that use f.

3. Interaction of Memory Models

In this section, we identify how the JMM is enforced in a multiprocessor platform and how it can affect the performance of multithreaded Java programs.

3.1. Overview

Figure 1 shows the relationship between the JMM and the underlying hardware memory model in a multiprocessor implementation of Java multithreading. Both the compiler reorderings as well as the reorderings introduced by the hardware memory model need to respect the JMM. Pugh has studied how an inappropriate choice of JMM can disable common compiler reorderings [16]. In this paper, we systematically study how the choice of JMM can enable/disable reorderings allowed by the hardware memory models. Note that if the hardware memory model is more relaxed (allows more reorderings and thereby allows more behaviors) than the JMM, then the Java Virtual Machine (JVM) needs to disable these reorderings. This disabling is done via inserting memory barrier instructions at appropriate places. A memory barrier instruction forces the processor to execute memory operations across the barrier in program order. If the JMM is more restrictive than the hardware memory model, a multithreaded Java program will execute with too many memory barriers on multiprocessor platforms. On the other hand, if the hardware memory model is too restrictive compared to the JMM, the performance enhancing features of the JMM cannot be exploited in that framework. This explains how the choice of JMM can affect the multithreaded program performance on multiprocessors.

We want to study the effect of the JMM in enabling or disabling reorderings allowed by the hardware memory models. To evaluate the impact of a JMM on multiprocessor performance, we need to check whether the JMM permits the relaxations allowed by the multiprocessor memory

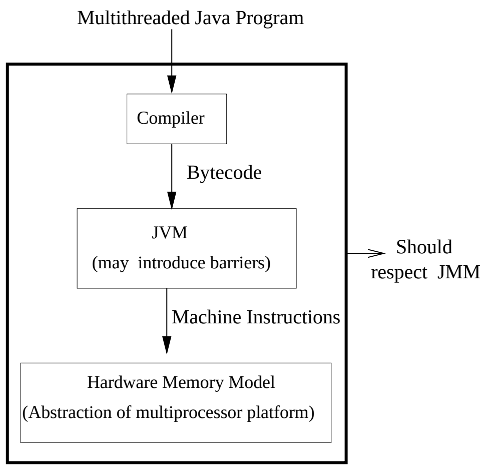


Figure 1: Multiprocessor Implementation of Java Multithreading

model concerned. If they are disallowed, then memory barriers need to be explicitly inserted degrading performance. We do so in this paper for two JMMs on five different hardware memory models.

3.2. Memory Barrier Insertion

We now discuss which reorderings of operations by hardware need to be prevented (by inserting memory barrier instructions) to maintain the JMM semantics on specific hardware memory models. First, we partition the operations in question into two classes: shared variable reads/writes, volatile variable reads/writes and synchronization operations (lock/unlock). Furthermore, as $\mathrm{JMM}_{\mathrm{new}}$ gives special semantics for final fields, we consider final field writes separately.

Notations We will employ the following notations to classify the reorderings that need to be prevented. If we associate a requirement $Rd^{\uparrow}$ ($Wr^{\uparrow}$) with operation x, it means that all read (write) operations occurring before x must be completed before x starts. On the other hand, if a requirement of $Rd_{\downarrow}$ ($Wr_{\downarrow}$) is associated with operation x, then all read (write) operations occurring after x must start after x completes. Finally, $RW^{\uparrow} \equiv Rd^{\uparrow} \wedge Wr^{\uparrow}$ and $RW_{\downarrow} \equiv Rd_{\downarrow} \wedge Wr_{\downarrow}$.

Barriers for Reads/Writes In the absence of locks, unlocks, and volatile variables, reads/writes to shared variables can be arbitrarily re-ordered within a thread for $\mathrm{JMM}_{\mathrm{old}}$. These reorderings are subject to satisfying the uniprocessor control/data dependencies. Even though $\mathrm{JMM}_{\mathrm{new}}$ advocates that the actions are "usually done in their original order" within a thread, the semantics of the read/write actions allows for other reorderings to be deduced. This fact is explicitly mentioned in [12, 13] and summarized in [10] in the form of a reordering table. According to the reordering table, shared variable accesses can be arbitrarily re-ordered within a thread. Since both the JMMs freely allow reordering of shared variable reads/writes among themselves, no memory barrier needs to be inserted before read/write instructions (in the absence of lock, unlock, and volatile variables) to satisfy the JMM semantics under any of the hardware memory models.

Barriers for Lock/Unlock In both the JMMs, the order between a lock and the succeeding reads/writes as well as unlock and previous reads/writes are maintained for proper implementation of synchronization semantics. In addition, $\mathrm{JMM}_{\mathrm{old}}$ does not allow locks to be re-ordered w.r.t preceding reads and unlocks w.r.t. following writes. $\mathrm{JMM}_{\mathrm{new}}$ has a more relaxed model for synchronization operations which is similar to RC hardware memory model. It relaxes the order between a lock and the preceding reads/writes as well as unlock and the following reads/writes. Finally, both the JMMs do not allow synchronization operations to bypass each other.

The memory barrier insertion requirements for lock and unlock to satisfy $\mathrm{JMM}_{\mathrm{old}}$ and $\mathrm{JMM}_{\mathrm{new}}$ are summarized in Table 4. These results are derived by comparing Table 1 with Table 2 and Table 3, respectively. For example, consider a particular hardware memory model, say PSO, and a software memory model, say $\mathrm{JMM}_{\mathrm{new}}$. By comparing Table 1 with Table 3, we can see that PSO allows the following reorderings that are not allowed by $\mathrm{JMM}_{\mathrm{new}}$

- write followed by unlock
- unlock followed by unlock
- unlock followed by lock

The first two reorderings are disabled by associating $Wr^{\uparrow}$ with unlock and the last reordering is disabled by associating $Wr^{\uparrow}$ with lock.

Table 4 is not a conclusive guide on the expected performance of multithreaded Java benchmarks on various hardware memory models. For example, the WO memory model does not introduce any barriers before/after unlock as shown in Table 4. However, the effect of barriers is achieved by the hardware itself. Therefore, to measure actual performance of multithreaded Java programs on various multiprocessor platforms, a simulation study is essential.

Barriers for Volatile Reads/Writes Recall that a volatile variable is one for which the JVM always accesses the shared copy. $\mathrm{JMM}_{\mathrm{old}}$ does not allow reads/writes of volatile variables to be re-ordered among themselves though they may be re-ordered w.r.t. reads/writes of normal variables. Thus the compiler has to introduce memory barriers with volatile reads/writes. Note that a memory barrier before an instruction I simply forces all reads and/or writes before I to commit before I starts. It is not possible to introduce a

Operation	SC	TSO	PSO	WO	RC
Lock					$Rd^{\uparrow}$
Unlock			$Wr^{\uparrow} \wedge Wr^{\downarrow}$		$Wr^{\downarrow}$

(a) $\mathrm{JMM_{old}}$

Operation	SC	TSO	PSO	WO	RC
Lock			$Wr^{\uparrow}$		
Unlock			$Wr^{\uparrow}$		

(b) $\mathrm{JMM_{new}}$

Table 4: Satisfying the reordering requirements for lock and unlock in the JMMs

barrier that selectively forces only the incomplete volatile reads/writes to commit. The memory barrier insertion requirement for volatile reads/writes under $\mathrm{JMM_{old}}$ are:

Operation	SC	TSO	PSO	WO	RC
VRd		$Wr^{\uparrow}$	$Wr^{\uparrow}$	$RW^{\uparrow}$	$RW^{\uparrow}$
VWr			$Wr^{\uparrow}$	$RW^{\uparrow}$	$RW^{\uparrow}$

Allowing arbitrary reordering of normal and volatile reads/writes creates some problems for $\mathrm{JMM_{old}}$. Consider the following pseudo-code.

Thread 1	Thread 2
write a,1	read volatile v
write a,2	read a
write volatile v,1	

Assuming v and a are initialized to 0, it is possible to read $v = 1$ and $a = 1$ in the second thread. Indeed it is this weakness of the volatile variable semantics which prevents an easy fix of the "Double Checked Locking" idiom [18] using volatile variables.

In $\mathrm{JMM_{new}}$, this problem is rectified by assigning "acquire-release" semantics to volatile variable operations. Thus a volatile write behaves like a "release" operation and cannot bypass the previous normal reads/writes; similarly a volatile read behaves like an "acquire" operation and cannot be bypassed by the following normal reads/writes. Volatile variable reads/writes are still not allowed to bypass each other, just like $\mathrm{JMM_{old}}$. Reordering requirements for volatile reads/writes w.r.t. normal reads/writes in order to satisfy $\mathrm{JMM_{new}}$ are shown in the following.

Op	SC	TSO	PSO	WO	RC
VRd		$Wr^{\uparrow}$	$Wr^{\uparrow}$	$RW^{\uparrow}\wedge RW^{\downarrow}$	$RW^{\uparrow}\wedge RW^{\downarrow}$
VWr			$Wr^{\uparrow}$	$RW^{\uparrow}$	$RW^{\uparrow}$

As we can see, for WO and RC we have the additional $RW_{\downarrow}$ requirement for volatile reads owing to its treatment as a lock acquisition operation.

Barriers for Final Fields Final fields are the fields of an object which are written only once (i.e., in the constructor). $\mathrm{JMM_{old}}$ does not prescribe any special semantics for final fields, and treats them as normal variables. However, $\mathrm{JMM_{new}}$ provides specialized semantics for final fields. Here we only consider final fields which are not visible to other threads before the constructor terminates (called "properly constructed" final fields in $\mathrm{JMM_{new}}$). This semantics requires that *all* the writes (writes to final as well as non-final fields) in a constructor to be visible when a final field is frozen (i.e., initialized). Since we can assume that all final fields are frozen before the termination of the constructor, the effect of this semantics can be achieved by inserting a barrier at the end of a constructor. Thus, the return statement of the constructor comes with the requirement $Wr^{\uparrow}$.

To measure the performance impact of this semantics of $\mathrm{JMM_{new}}$, it is just not enough to measure the time taken by these additional barriers. Due to the additional safety provided by the semantics, certain synchronizations can now be eliminated. However, identifying these neo-redundant synchronizations is rather difficult, and we have not done so in this paper. Nevertheless, we found the overhead due to additional barriers for final field writes is not substantial.

4. Performance Evaluation

In this section, we compare the impact of $\mathrm{JMM_{old}}$ and $\mathrm{JMM_{new}}$ on multithreaded Java program performance through simulation study. First, we consider the performance of various Java Grande benchmarks on different hardware memory models. We then study the performance of the same benchmarks when these hardware memory models are required to comply to $\mathrm{JMM_{old}}$ and $\mathrm{JMM_{new}}$.

4.1. Benchmarks

We choose five different benchmarks from multithreaded Java Grande suite, `SOR`, `LU`, `Series`, `Sync`, and `Ray`, which are suitable for parallel execution on shared memory multiprocessors [7]. `Sync` is a low-level benchmark that measures the performance of synchronized methods and blocks. `SOR`, `LU`, and `Series` are moderate-sized kernels. `LU` solves a 40×40 linear system using LU factorization followed by a triangular solve. It is a Java version of the well known Linpack benchmark. `SOR` performs 100 iterations of successive over-relaxation on a 50×50 grid. `Series` computes the first 30 Fourier coefficients of the function $f(x) = (x + 1)^x$ on the interval $0 \ldots 2$. `Ray` is a large scale application that renders a 3D scene containing 64 spheres. Each benchmark is run with four parallel threads. Table 5 shows the number of volatile reads/writes,

Benchmark	Volatile Read	Volatile Write	Constructors with Final Field Writes	Lock	Unlock
SOR	51604	480	20	4	4
LU	8300	936	52	4	4
Series	0	0	24	4	4
Sync	0	0	4	4	4
Ray	48	20	768	8	8

Table 5: Characteristics of Benchmarks used

synchronization operations and final field writes for our benchmarks. The LU and SOR benchmarks have substantial number of volatile variable reads/writes, accounting for up to 15% of the total memory operations.

4.2. Methodology

We use Simics [20], a full-system simulator for multiprocessor platforms in our performance evaluation. Simics is a system-level architectural simulator developed by Virtutech and supports various processors like SPARC, Alpha, x86 etc. It can run completely unmodified operating systems. We take advantage of the set of application programming interfaces (API) provided by Simics to write new components, add new commands, and write control/analysis routines.

Multiprocessor platform We simulate a shared memory multiprocessor (SMP) consisting of four SUN UltraSPARC II. The processors are configured as 4-way superscalar out-of-order execution engines with 64-entry reorder buffers. We use separate 256KB instruction and data caches: each is 2-way set associative with 32-byte line size. The caches use write-back, write-allocate policy with cache miss penalty of 100 clock cycles. We use MESI cache coherence protocol.

Simics provides two basic execution modes: an in-order execution mode and an out-of-order mode. In in-order execution mode, instructions are scheduled sequentially in program order. Thus there are no reorderings among memory operations in this mode. The out-of-order execution mode has the feature of a modern pipelined out-of-order processor. That is, the instructions need not be issued to the functional units and completed in program order. In Simics, this is achieved by breaking instructions into several phases that can be scheduled independently. This mode can produce multiple outstanding memory requests that do not necessarily occur in program order. Clearly, we must use out-of-order execution mode in our experiments so that we can simulate multiprocessor platform with different hardware memory models.

Multithreading Support We use Linux SMP kernel as the operating system and Kaffe as the JVM. Since our simulated platform is a four-processor SMP machine, some measures are taken for the multithreaded programs to make best use of the multiprocessors. Linux supports multithreading through POSIX Threads (Pthreads), message passing libraries and multiple processes. Since both message passing libraries and multiple processes typically communicate either by means of Inter-Process Communications (IPC) or a messaging API, they do not map naturally to SMP machines. Only Pthreads enforce a shared memory paradigm and so we decide to use Pthreads.

Moreover, Kaffe provides several methods for the implementation of Java multithreading including kernel-level and application-level threads. However, threads at application-level do not take advantage of the kernel threading and all the threads are mapped to a single process. As a result, application level threads cannot take advantage of SMP. Consequently, we decide to use kernel Pthreads library so that the Java threads are scheduled to different processors.

Hardware Memory Models We configure Simics consistency controller [19] to simulate various hardware memory models. The consistency controller ensures that the architecturally defined consistency model is not violated. The consistency controller can be constrained through the following attributes (setting an attribute to zero will imply no constraint):

- **load-load**, if set to non-zero loads are issued in program order
- **load-store**, if set to non-zero program order is maintained for stores following loads
- **store-load**, if set to non-zero program order is maintained for loads following stores
- **store-store**, if set to non-zero stores are issued in program order

Obviously, if all the four attributes are set to non-zero, program order is maintained for all the memory operations. In this case, the hardware memory model is SC. For TSO where writes can be reordered w.r.t. the following reads, we only need to set store-load to zero and other attributes to non-zero. For PSO, store-load and store-store are set to zero and the other two to non-zero. In addition, for implementing PSO, we had to modify the Simics consistency controller. The default consistency controller stalls a store operation if there is an earlier instruction that can cause an exception and all instructions are considered to be able to raise an exception. Therefore, in effect, a store instruction cannot bypass any previous instruction in the original implementation. We allowed the store to go ahead even if there are uncommitted earlier instructions. If the earlier instructions raise exception, we simply aborted the simulation. However, this happened very infrequently.

For WO and RC, it is not sufficient to just set the four attributes to zero. We need to distinguish between synchronization and memory operations. However, the synchronization operations cannot be identified in Simics as there is no special instruction for synchronization operation unlock. Instead, we identify synchronization operations at JVM level as Java bytecode has two specific opcodes MONITORENTER and MOINTOREXIT for lock and unlock, respectively.

For WO, lock/unlock cannot be reordered w.r.t. normal read/write operations. Thus we need to put memory barriers before and after lock/unlock. Since a lock is essentially an atomic read-modify-write operation and any operations following the lock can execute only when the lock is completed successfully, operations following a lock are dependent on the lock and thus they can't bypass the lock. Therefore no memory barrier is required after a lock operation. In Kaffe, WO is achieved by inserting one memory barrier just before the implementation of MONITORENTER, one just before and one just after MONITOREXIT. For RC, we only need to insert one memory barrier before the implementation of MONITOREXIT.

Software Memory Models The software memory models are implemented by inserting memory barriers for the different models in the Java programs. As Java is platform independent, we need to make use of the Java Native Interface (JNI) to insert architecture-dependent assembly language instructions (memory barriers). The JNI allows Java code running in JVM to co-exist with applications and libraries written in other languages, such as C, C++, and assembly. The following steps create a native method containing the memory barrier instructions which can be invoked (inserted) into any Java program.

1. Write a Java program that declares the native method.
2. Compile the Java program into a class that contains the declaration for the native method.
3. Generate a header file for the native method using *kaffeh* provided by the JVM.
4. Write the implementation of the native method in the desired language. We use C with inline assembly.
5. Compile the header and implementation files into a shared library file.

5. Experimental results

As discussed before, all the five multithreaded Java Grande benchmarks are modified to observe the $\mathrm{JMM}_{\mathrm{old}}$ and $\mathrm{JMM}_{\mathrm{new}}$ specifications respectively. Then they are executed on the simulated system configured with SC, TSO, PSO, WO or RC hardware memory models. The performance is measured by the number of clock cycles required for a benchmark under certain combination of software and hardware memory models. Simics introduces non-determinism in scheduling threads. Therefore, for each combination of hardware and software memory models, we run a benchmark three times and take the average. We first present the number of memory barriers required for both $\mathrm{JMM}_{\mathrm{old}}$ and $\mathrm{JMM}_{\mathrm{new}}$ for relaxed hardware memory models.

5.1. Memory Barriers

The number of memory barriers greatly influences the performance of the benchmarks and reflects the requirements of the software memory models. The memory barriers are due to volatile variable accesses, synchronization operations and final fields. Since the benchmarks we use do not contain many synchronization operations, most of the memory barriers are introduced because of volatile variables and final fields. The overhead due to memory barriers arise not just from the clock cycles needed to execute them but also from the clock cycles spent in waiting for the pending memory operations to complete.

Table 6 shows the number of memory barriers for $\mathrm{JMM}_{\mathrm{old}}$ and $\mathrm{JMM}_{\mathrm{new}}$ under relaxed hardware memory models. We also provide a breakup of why these barriers are introduced. In some cases, the numbers of memory barriers required are the same for two different models, but the barriers are introduced due to different reasons.

Since SC is stricter than both of the JMMs, no memory barrier is required for it. From Table 6 we can see that `LU`, `SOR` and `Ray` need more memory barriers than `Series` and `Sync`. This is because `LU`, `SOR` and `Ray` all have a large number of volatile reads/writes. As $\mathrm{JMM}_{\mathrm{new}}$ imposes more restrictions on volatile variables, generally $\mathrm{JMM}_{\mathrm{new}}$ needs more barriers than $\mathrm{JMM}_{\mathrm{old}}$ for these three benchmarks under certain hardware memory models. Moreover, for $\mathrm{JMM}_{\mathrm{old}}$ we can observe that the hardware memory models PSO, WO and RC need more barriers than TSO. The reason is that TSO needs memory barriers to be inserted before volatile read operations while PSO, WO and RC need memory barriers to be inserted before both volatile read and volatile write operations under $\mathrm{JMM}_{\mathrm{old}}$. Similarly, for $\mathrm{JMM}_{\mathrm{new}}$, PSO introduces more memory barriers than TSO while WO and RC introduce more barriers than PSO.

SOR		TSO	PSO	WO	RC
	volatile rd/wr	2004	2812	2808	2812
	lock/unlock	0	4	0	8
	final field wr	0	0	0	0
JMM_{old}	*Total*	**2004**	**2816**	**2808**	**2820**
	volatile rd/wr	2004	2810	4813	4813
	lock/unlock	0	4	0	0
	final field wr	2	2	2	2
JMM_{new}	*Total*	**2006**	**2816**	**4815**	**4815**

LU		TSO	PSO	WO	RC
	volatile rd/wr	3979	4924	4920	4824
	lock/unlock	0	4	0	8
	final field wr	0	0	0	0
JMM_{old}	*Total*	**3979**	**4928**	**4920**	**4832**
	volatile rd/wr	3979	4922	6124	6124
	lock/unlock	0	4	0	0
	final field wr	6	6	6	6
JMM_{new}	*Total*	**3985**	**4932**	**6130**	**6130**

Series		TSO	PSO	WO	RC
	volatile rd/wr	0	0	0	0
	lock/unlock	0	6	0	12
	final field wr	0	0	0	0
JMM_{old}	*Total*	**0**	**6**	**0**	**12**
	volatile rd/wr	0	0	0	0
	lock/unlock	0	6	0	0
	final field wr	0	0	0	0
JMM_{new}	*Total*	**0**	**6**	**0**	**0**

Sync		TSO	PSO	WO	RC
	volatile rd/wr	0	0	0	0
	lock/unlock	0	4	0	8
	final field wr	0	0	0	0
JMM_{old}	*Total*	**0**	**4**	**0**	**8**
	volatile rd/wr	0	0	0	0
	lock/unlock	0	4	0	0
	final field wr	1	1	1	1
JMM_{new}	*Total*	**1**	**5**	**1**	**1**

Ray		TSO	PSO	WO	RC
	volatile rd/wr	35	84	49	84
	lock/unlock	0	8	0	16
	final field wr	0	0	0	0
JMM_{old}	*Total*	**35**	**92**	**49**	**100**
	volatile rd/wr	35	92	73	100
	lock/unlock	0	8	0	0
	final field wr	863	863	863	863
JMM_{new}	*Total*	**898**	**963**	**936**	**963**

Table 6: Number of memory barriers inserted in benchmarks for different memory models

This is because TSO needs memory barriers before volatile read operations, and PSO needs barriers before both volatile read and write operations. WO and RC, on the other hand, need barriers before volatile read/write operations as well as after volatile read operations.

The other two benchmarks `Series` and `Sync` have no volatile variables. The memory barriers are due to synchronization operations and final fields. For synchronization operations, JMM_{old} needs more memory barriers than JMM_{new}. Thus for these two benchmarks, more memory barriers are inserted for JMM_{old} than JMM_{new}.

Among the benchmarks, only `Ray` has substantial number of constructors with final fields. Hence the number of memory barriers due to final fields is only observable in `Ray`. This causes JMM_{new} to have more memory barriers than JMM_{old}. Also note that the final fields are treated in the same way for all the hardware memory models.

5.2. Performance

In our simulation, the execution time of a benchmark is affected by two factors: the inserted memory barriers due to JMMs and the reorderings allowed by the hardware memory models. We now present the speedup of a benchmark's execution for a particular combination of hardware memory model (TSO/PSO/WO/RC) and software memory model (JMM_{old}/JMM_{new}) with respect to SC. We use SC as the base case as it is the strictest hardware memory model and no reordering is allowed among the memory operations. Thus the execution time under SC is not affected by the software memory model.

Figure 2 illustrates the performance of the benchmarks for different combinations of software and hardware memory models. The "OLD" and "NEW" correspond to JMM_{old} and JMM_{new} software memory models, respectively. "NO" represents a hypothetical case where we do not follow any software memory model semantics. In other words, we do not insert any memory barrier to respect the JMM. Note that this may result in an execution which violates the JMM semantics. However, we introduce "NO" to evaluate the impact of the software memory model on program performance. In particular, we are interested to see if the JMM offsets the performance improvement offered by hardware memory models. The Y-axes in Figures 2 represent the speedup with respect to SC. Note that for SC, the performance remains the same irrespective of the software memory model.

For all the five benchmarks the hardware memory models have crucial impact on the overall performance; the more relaxed the hardware memory model, the better the performance. These results are consistent with those results of [4]. It is possible to achieve 20–60% performance improvement for RC compared to SC even in the presence of software memory models.

The comparison between *no JMM* and $\mathrm{JMM}_{\mathrm{old}}$, $\mathrm{JMM}_{\mathrm{new}}$ show that for most benchmarks (`Ray`, `Sync`, and `Series`), the software memory model has minimal impact on performance. However, for benchmarks with large number of volatile variables, there can be as much as 5% performance difference due to the presence of JMM. As the hardware cannot distinguish between volatile and normal variables, it is difficult to map the software memory model semantics to hardware and we need to introduce unnecessary barriers.

Finally, the performance difference between $\mathrm{JMM}_{\mathrm{old}}$ and $\mathrm{JMM}_{\mathrm{new}}$ is negligible. We note than in general the difference is larger under WO and RC than under TSO and PSO. This is because under WO and RC more memory barriers are introduced due to volatile variables for $\mathrm{JMM}_{\mathrm{new}}$. That is why the benchmarks with significant number of volatile variables have worse performance under $\mathrm{JMM}_{\mathrm{new}}$ than $\mathrm{JMM}_{\mathrm{old}}$.

6. Discussion

In this paper, we have studied the performance impact of Java Memory Model (JMM) on hardware consistency models of multiprocessor platforms. A hardware consistency model describes the behaviors allowed by the multiprocessor implementations while the JMM describes behaviors allowed by Java multithreading semantics. The existing JMM and a new JMM by Manson and Pugh (which is close to the planned revision by the JSR-133 expert group) are used in this study to show how different choices of JMM can affect multithreading performance. To ensure that the execution of the multithreaded Java program on the multiprocessor with some hardware consistency model does not violate the JMM, we add memory barriers to enforce ordering. We use the Simics multi-processor simulator to quantitatively evaluate the effect of these memory barriers on overall performance of multi-threaded Java programs.

In terms of future work, our study needs to be extended to other similar proposals such as [1]. Furthermore, note that our study only captures the relationship of JMM and hardware memory models. The effect of compiler optimizations (i.e., whether a JMM enables/disables compiler optimizations) is not included in this study. For a more comprehensive understanding of the effect of a JMM on overall system performance, we need an integrated study of its effect on compilers as well as multiprocessor architectures.

7. Acknowledgments

This was was partially supported by National University of Singapore research project R-252-00-095-112.

References

[1] S. Adve. A memory model for Java and its rationale. In *Communication to Java Memory Model mailing list*, 2003.

[2] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, Dec. 1996.

[3] D. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.

[4] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessor. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[5] A. Gontmakher and A. Schuster. Java consistency: nonoperational characterizations for java memory behavior. *ACM Transactions on Computer Systems (TOCS)*, 18(4):333–386, 2000.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Chapter 17, Addison Wesley, 1996.

[7] Java Grande Forum. *The Java Grande Forum Benchmark Suite*, 2001. Multi-threaded benchmarks available from `http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/thre%ads.html`.

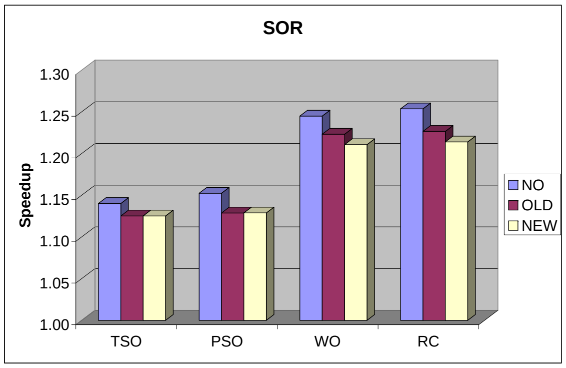
[8] Java Specification Request (JSR) 133. *Java Memory model and thread specification revision*, 2001. `http://jcp.org/jsr/detail/133.jsp`.

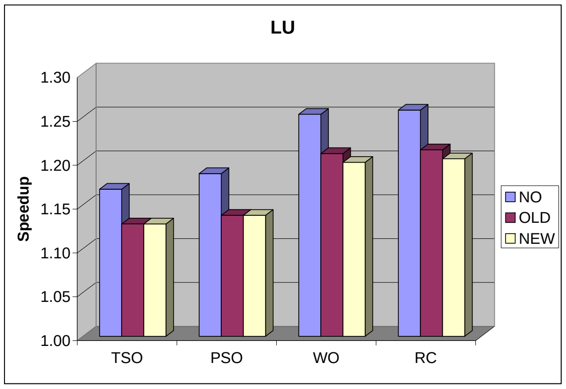
[9] L. Lamport. How to make a multiprocessor computer that correctlt executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

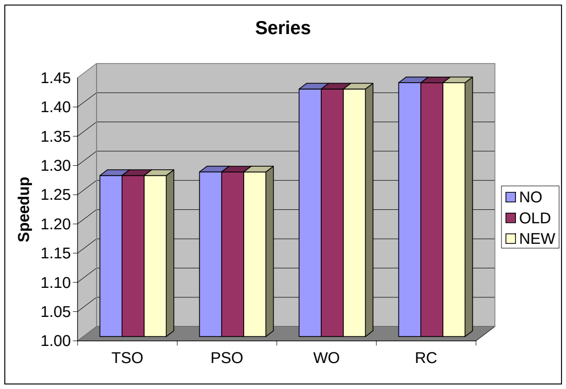
[10] D. Lea. The JSR-133 cookbook for compiler writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

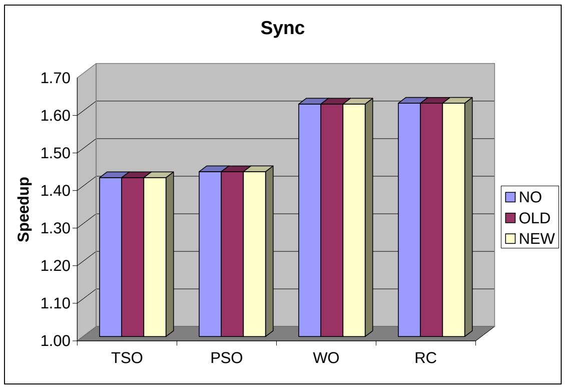
[11] J. Maessen, Arvind, and X. Shen. Improving the java memory model using CRF. In *ACM OOPSLA*, 2000.

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









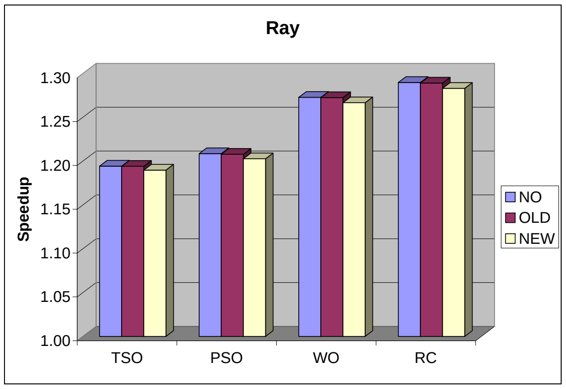


Figure 2: Performance impact of JMM. OLD, NEW, and NO correspond to $\mathrm{JMM_{old}}$, $\mathrm{JMM_{new}}$, and no software memory models, respectively. The speedups are with respect to the hardware memory model SC.

[13] J. Manson and W. Pugh. Semantics of multithreaded Java. Technical report, Department of Computer Science, University of Maryland, College Park, CS-TR-4215, 2002. `http://www.cs.umd.edu/~pugh/java/memoryModel`.

[14] V. Pai, P. Ranganathan, S. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[15] W. Pugh. *Java Memory Model Mailing List*. `http://www.cs.umd.edu/~pugh/java/memoryModel/archive`.

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

[17] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2002.

[18] 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.

[19] Virtutech. *Simics Out-of-order Processor Model*, 2003.

[20] Virtutech. *Simics User Guide for UNIX*, 2003.

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

[22] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Joint ACM Java Grande/ISCOPE conference*, 2002.

[23] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: An operational memory model specification framework with integrated model checking capability. *Concurrency and Computation: Practice and Experience*, 2003.