THE NATIONAL UNIVERSITY
of SINGAPORE

School *of* Computing
Lower Kent Ridge Road, Singapore 119260

TRB7/03

Performance Impact of Multithreaded Java Semantics on Multiprocessor Memory Consistency Models

Lei XIE, Abhik ROYCHOUDHURY and Tulika MITRA

July 2003

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Performance Impact of Multithreaded Java Semantics on Multiprocessor Memory Consistency Models

Lei Xie Abhik Roychoudhury Tulika Mitra

School of Computing
National University of Singapore
[xielei,abhik,tulika]@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 and describes the allowed re-orderings among the operations in a thread. However, multiprocessor platforms traditionally have a memory consistency model of their own. Consequently memory barriers may have to be inserted to ensure that the multiprocessor execution of a multithreaded Java program respects the Java Memory Model. In this paper, we study the impact of these additional memory barriers on multiprocessor performance. We also study how different choices of the Java Memory Model affect multiprocessor performance. Our experimental results are obtained by simulating multithreaded Java Grande benchmarks under various software and hardware memory models.

1. INTRODUCTION

Multithreading is a useful feature in a programming language. It can be used to structure parts of the program, or to program concurrent access of shared data structures by different threads. The Java programming language supports shared memory multithreading where threads can manipulate shared objects. Critical sections for shared variable access are supported by means of synchronized methods and statements.

The threads of a multithreaded Java program can be run either on a single processor or on a multiprocessor platform. When run on a single processor, these threads may be managed by a JVM scheduler (as in Kaffe) or can be managed by a thread library such as POSIX threads. Alternatively, the threads may be run on a shared memory multiprocessors connected by a bus or interconnection network. In these platforms, the writes to shared variable made by some thread are not immediately visible to other threads.

Java provides a semantics for multithreading irrespective of how the multithreading is implemented. This is called the Java Memory Model (henceforth called JMM). Conceptually, the JMM should be understood as the set of all possible behaviors that a multithreaded Java program can demonstrate on any implementation platform. Given a multithreaded program, the JMM defines all possible behaviors to be generated by: (a) arbitrary interleaving of the threads (b) certain re-orderings of operations in each thread. Each shared variable read or write as well as any lock or unlock is considered a distinct operation. Thus, a JMM can be described in terms of the allowed re-ordering of operations in a thread. This is in tune with the way in which hardware memory consistency models are conventionally defined.

The introduction of a memory model at the programming language level raises new challenges. In particular, note that a multiprocessor platform comes with its own hardware memory consistency model. This is the set of re-orderings which are allowed by the implementation of the multiprocessor platform (*e.g.* a write buffer allows reads to bypass writes). However, if the threads of a multithreaded Java program are run on multiple processors, then we also need to ensure that all the executions respect the JMM, the software memory model. Clearly if the hardware memory model allows *more* re-orderings than the JMM then we need additional mechanism to enforce the JMM. This can be done by inserting barriers at the Java Virtual Machine (JVM) level.

1.1 Contributions

In the absence of any software memory model, there exists a clear understanding of which hardware memory model is more efficient. Indeed, detailed experimental studies have been conducted to evaluate the relative performance of different hardware memory models [5]. In this paper, we study the relative performance of hardware memory models in the presence/absence of a JMM.

Furthermore, the specification of the Java Memory model is currently a topic of intense discussion [1]. An initial JMM was proposed in the Java Language Specification [7]. Subsequently, this was found to restrict various common optimizations, and too hard to understand [18]. Thus, an expert group has been formed to completely revise the JMM, and concrete new proposals have appeared (*e.g.* [14]). Clearly, a relative study of the performance impact of the old and new JMM is required. This can help in formulation of the new JMM, and pinpoint the performance-enhancing features of the new JMM proposals. In this paper, we take a step in this direction.

1.2 Organization of the Paper

The rest of this paper is organized as follows. In the next section, we review the technical background on JMM. Section 3 clarifies the relationship between and hardware and software memory models. Section 4 describe the methodology to identify the performance effects of JMM; this is done by identifying the memory barriers inserted. Section 5 describe the experimental setup for measuring the effects of a software memory model on multiprocessor platforms. Section 6 describes the experimental results obtained from evaluating the performance of multithreaded Java Grande benchmarks under various hardware and software memory models. Section 7 concludes the paper with a summary of the results.

2. BACKGROUND AND RELATED WORK

In this section, we present some brief background of the old JMM, the new JMM, and hardware memory models. For more details, the reader is referred to the existing concrete proposals of the new JMM as well as the discussion in the JMM mailing list [12]. We also discuss related work in the formalization and implementation of JMM.

2.1 The Old JMM

The old JMM is specified in Chapter 17 of The Java Language Specification [7]. This model is a set of abstract rules dictating the allowed ordering of read/write operations of shared variables. The Java threads interact among themselves via shared variables. For any shared variable v, each thread possesses a local copy of v and is allowed to access the global master copy of v in the main memory. The JMM essentially imposes constraints on the interaction of the threads with the master copy of the variables and thus with each other.

Two important points need to be noted here. First, the local copies of shared variables conceptually form a thread's "cache". Secondly, data transfer between the local and the master copy is not modeled as an atomic action. This is to model the realistic transit delay when the master copy is located in the hardware shared memory and the local copy is in the hardware cache.

The old JMM defines eight distinct actions. As a thread t executes a program, it operates on the local copy and the master copy via the following actions:

- **use**: Read from the local copy of a variable v in t
- **assign**: Write into the local copy of a variable
- **read** / **load** : Initiate / Complete reading from master copy to local copy
- **store** / **write** : Initiate / Complete writing from master copy to local copy

Apart from the above actions, each thread t may perform **lock/unlock** on shared variables. 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 specialized in the JMM. A major difficulty in reasoning about the JMM 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 an action. As a result, this framework is hard to understand and the lack of rigor in specification has led to some problems such as: important compiler optimization (*e.g.* fetch elimination) are prohibited [18]. A formal executable specification of the old JMM appears in our earlier work [20].

2.2 The JMM Revision

The JMM is currently under an official revision by an expert group JSR-133 [1]. New semantics have been proposed for Java threads as candidates of the JMM revisions. Proposals for revised JMM have been given by Manson & Pugh (appeared in [14] and subsequently revised further), Maessen et. al. [13], and recently by Adve [2]. These proposals have fundamental differences with the old JMM, and the JSR-133 expert group is now converging towards a revised JMM. Some of the characteristics that distinguishes the planned revision from the old JMM are as follows.

- An *acquire/release* semantics is prescribed for volatile variable reads/writes. This allows for the usage of volatile variables as synchronization flags.
- It prescribes relaxation of total order among write instructions on the same variable, so that compiler optimizations (*e.g.* fetch elimination) are enabled.
- Separate semantics is prescribed for final fields (fields which are written only once in the constructor).

Apart from the concrete proposals, a full fledged discussion on the planned features of the revised Java Memory Model appears in [12]. The interested reader is referred to this mailing list.

2.3 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) [10]. 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 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 re-ordering of operations within a thread. In this paper, we study the impact of Java Memory Model on the performance of four relaxed hardware memory models, namely: Total Store Order (TSO), Partial Store Order (PSO), Weak Ordering (WO) and Release Consistency (RC). Details of these memory models appear in [3]. Note that all memory models allow only those re-orderings which do not violate the data flow dependencies within a thread; thus the sequence ⟨ `write u, read u` ⟩ will never be re-ordered.

The TSO and PSO models (supported by SUN SPARC architecture [4]) differ from SC in the fact that they allow memory write operations to be bypassed. Memory read operations are blocking in TSO and PSO. Thus, the TSO model differs from SC only in the fact that it allows reads to bypass previous writes. PSO is a more relaxed model than TSO since 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 it classifies memory operations as data operations (normal reads/writes) and synchronization operations (lock/unlock). WO allows data operations between synchronization operations to be arbitrarily re-ordered. However, operations across a synchronization operation cannot be re-ordered. This raises the question of relaxing the order between lock and preceding reads/writes as well as unlock and following reads/writes. These two relaxations are allowed in the RC model, but not in the WO model.

2.4 Related Work

Some work has been done on the processor level to evaluate the performance of different hardware consistency models. The work of Gharachorloo et al. [5] shows that in a platform with blocking reads and delayed commit of writes, performance is substantially improved by allowing reads to bypass writes. It also shows that SC performs poorly relative to all other models. Pai et al. studied the implementation of SC and RC models on current generation processors with aggressive exploitation of instruction level parallelism (ILP) [17]. 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 implementation 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 Java Memory Model (JMM). Some work has been done to formalized the old JMM [6, 19, 20]. In particular, we developed an operational executable specification of the old JMM in [20]; its utility in verifying low-level multi-threaded program fragments such as "Double Checked Locking" was also shown. Yue Yang et al.[23] used an executable framework called Uniform Memory Model (UMM) for specifying a new JMM developed by Manson and Pugh [14].

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 [11] is related to ours. This work serves as a comprehensive guide for implementing the new JMM (as currently specified by JSR-133). It provides brief backgrounds about why various rules exist and concentrates on their consequences for compilers and JVMs with respect to instruction re-orderings, choice of multiprocessor barrier instructions, and atomic operations. It includes a set of recommended recipes for complying to JSR-133. However, no performance evaluation numbers are presented in this work.

3. HOW CAN WE EVALUATE MEMORY MODELS ?

In this section, we identify how the semantics of Java multithreading (henceforth called the Java Memory Model or JMM) can affect the performance of multithreaded Java programs. Multithreaded Java programs can be executed on uniprocessor or multiprocessor platforms (which have a memory consistency model of its own). In particular, given a number of threads accessing a shared store, the shared-memory multiprocessor memory consistency model places

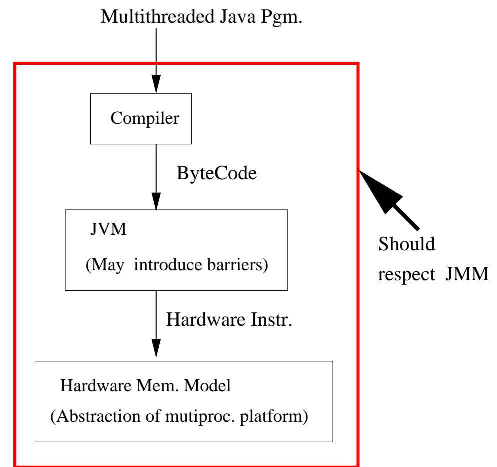


Figure 1: Multiprocessor Implementation of Java Multithreading

restrictions on the order in which the threads 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 model of execution supported by uniprocessor platforms is Sequential Consistency [10]. This model is more restrictive (allows less behaviors) than any proposed Java Memory Model.

We assume that our starting point is an unoptimized Java bytecode, i.e., any optimization will be allowed only on the bytecode. In case of uniprocessor and multiprocessor platforms, we need to ensure that the compiler optimizations do not violate JMM. In addition, for multiprocessor platforms which are not sequentially consistent, we need to ensure that the re-orderings allowed by the underlying hardware do not violate the JMM. In this case, the JVM could add memory barrier instructions to make sure that the JMM is not violated.

Before proceeding with our study, the relationship between the JMM and the underlying hardware memory model needs to be clarified (see [19] for details). Figure 1 shows a multiprocessor implementation of Java multithreading. Both the compiler re-orderings as well as the re-orderings introduced by the hardware memory model need to respect the JMM. Pugh has studied how an inappropriate choice of JMM can disable common compiler re-orderings [18]. In this paper, we systematically study how the choice of JMM can enable/disable re-orderings allowed by the hardware memory models. Note that if the hardware memory model is more relaxed (allows more instruction re-orderings thereby allowing more behaviors) than the JMM, then the JVM needs to insert memory barrier instructions in the program. If the JMM is too strong, a multithreaded Java program will execute with too many memory barriers on multiprocessor platforms. This explains how the choice of JMM can affect multithreaded program performance on multiprocessors.

We want to study the effect of the JMM in enabling or disabling re-orderings allowed by the multiprocessor memory models. To evaluate the impact of a JMM M on multiprocessor performance, we need to check whether M permits the relaxations allowed by the multi-processor memory

model concerned. If they are disallowed, then memory barriers will need to explicitly inserted. This will affect multithreaded program performance on multiprocessor platforms. Thus, to compare two Java Memory Models M and M' we need to study which of the re-orderings (allowed by the various hardware memory models) are disallowed by M and M'. This is done in the following section.

4. PREVENTING RE-ORDERINGS

We now discuss how implementing a JMM on specific hardware memory models involves preventing re-ordering of operations (by memory barrier insertion). We consider two candidate Java Memory models: (i) the old JMM (since outdated) given in the Java Language Specification [7] henceforth called JMM_{old}, and (ii) and a new JMM developed by Manson and Pugh [14], henceforth called JMM_{MP}. Note that there have been other candidate proposals for a new JMM (such as Maessen, Arvind and Shen's work in [13] and Adve's recent work [2]). Our study can be (and indeed should be) extended to these models as well. However, the purpose of our study is *not* to compare JMM_{old} and JMM_{MP} point-by-point. Instead we seek to identify the performance bottlenecks which can be created through various choices in designing JMM. We believe that such a study can be useful for identifying the performance impact of different features in the revised JMM.

For JMM_{old}, we refer to the operational style formal specification developed in [20]. For JMM_{MP} we use its formal executable description given in [22, 23]. In particular, the allowed re-orderings among operations in JMM_{MP} are explicitly specified as a *relaxed bypassing table* [22]. Similar sets of allowed re-orderings also appear in the re-ordering table of Doug Lea's cookbook [11].

To compare the effect of JMM on multiprocessor performance, we need to compare a candidate JMM (and its allowed behaviors) against various relaxed multiprocessor memory consistency models. We choose the following multiprocessor models (specified in order of relaxedness): Sequential Consistency (SC), Total Store Order (TSO) and Partial Store Order (PSO), Weak Ordering (WO) and Release Consistency (RC). To ascertain the effect of the software memory models (*i.e.* the candidate JMMs) we need to consider the re-orderings allowed by these models among the various classes of operations. We consider the following class of operations: shared variable reads, shared variable writes, lock, unlock, volatile variables reads and volatile variable writes. Furthermore, since JMM_{MP} gives special semantics for final fields, we consider final field writes separately.

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

4.1 Barriers due to Lock and Unlock

Irrespective of the choice of JMM, we never need to insert barriers after a lock operation. This is because a lock operation can be assumed to execute the following schematic code atomically.

```
label: r0 := test_and_set(lockvar)
       if r0 = 0 then go to label
```

We do not insert any barriers after this code, since any instruction after a lock possesses a control dependency with the condition `r0 = 0`, and hence cannot bypass the lock. Thus, in any hardware or software memory model, a lock operation is never associated with a requirement $Rd_{\downarrow}$ or $Wr_{\downarrow}$ or $RW_{\downarrow}$. However, we consider inserting barriers before a lock since operations before a lock (which is essentially an atomic read-and-write operation) can be completed after the lock under certain relaxed hardware memory models.

The schematic code for the unlock operation is of the form:

```
lockvar := 1
```

i.e. it is an atomic write to shared memory location. Therefore, operations after unlock may bypass unlock and operations before unlock may be bypassed by unlock. Therefore, we consider inserting barriers before and after unlock.

Let us now consider the barriers which need to be inserted before lock and before/after unlock under various relaxed models, so that a particular JMM is satisfied. First, we consider JMM_{old}; the results are summarized in Table 1(a). To explain how we derive the results, consider a particular hardware memory model, say TSO. TSO allows reads to bypass writes; no other bypassing is allowed. Note that lock is an atomic read-and-write operation. Since write cannot bypass other operations in TSO, no barriers need to be inserted before lock. Furthermore, unlock is a write operation and therefore unlock cannot bypass any previous instruction under TSO. Therefore, no barrier needs to be put before unlock. However, reads after unlock can bypass the unlock operation in TSO since unlock is simply an atomic write operation. This violates the program order restriction in JMM_{old} [20] which states that program operations must be issued in program order. Such a violation is prevented by associating the $Rd_{\downarrow}$ requirement with unlock.

Table 1 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 1. However, the effect of barriers is achieved by the memory model itself. To measure actual performance of multithreaded Java programs on various multiprocessor platforms, we have conducted simulation studies using multithreaded Java Grande benchmarks.

We now consider the barriers to be inserted under the various hardware memory models to satisfy JMM_{MP}. A formal description of JMM_{MP} appears in [23]. It describes the read/write/lock/unlock actions as guarded commands. In general, JMM_{MP} advocates a program order among actions with the exception that writes are allowed to bypass other operations. Even though JMM_{MP} advocates that the actions are "usually done in their original order" within a thread, the semantics of the read/write actions allows for other re-orderings to be deduced. This is because the value returned by a read operation on variable v in thread t (*i.e.* which write(s) of v is visible to thread t) is non-deterministic. This fact is explicitly mentioned [14, 15] and summarized in [22] in the form of a relaxed bypassing table. As per the relaxed bypassing table of JMM_{MP} no memory barriers need

Operation	SC	TSO	PSO	WO	RC
Lock	No	No	$Wr^{\uparrow}$	No	$RW^{\uparrow}$
Unlock	No	$Rd^{\downarrow}$	$Wr^{\uparrow} \wedge RW^{\downarrow}$	No	$RW^{\downarrow}$

(a) Satisfying JMM_{old}

Operation	SC	TSO	PSO	WO	RC
Lock	No	No	No	No	No
Unlock	No	No	$Wr^{\uparrow}$	No	No

(b) Satisfying JMM_{MP}

Table 1: Re-ordering Requirements for Lock and Unlock w.r.t. normal reads/writes in various JMMs

to be inserted before lock or after unlock under any hardware memory model (refer Table 1(b)). A $Wr^{\uparrow}$ requirement is needed for unlock in the PSO model, since unlock is not allowed to bypass reads/writes even in the relaxed bypassing table of JMM_{MP}.

Note that Table 1(b) only summarizes the re-ordering requirements of normal reads/writes w.r.t. synchronization operations (lock/unlock) for satisfying JMM_{MP}. In addition JMM_{MP} does not allow synchronization operations to bypass each other. This is ensured anyway in SC (no bypassing allowed), TSO (writes cannot bypass any preceding instruction and both lock/unlock involve writing[1]), WO and RC (synchronization operations are special, and they do not bypass each other). In PSO, we can ensure that synchronizations do not bypass each other by either adding a $Wr^{\uparrow}$ requirement with a lock operation, or adding a $Rd^{\downarrow}$ requirement with an unlock operation.

4.2 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 according to JMM_{old}. These re-orderings are subject to satisfying the data dependencies within a thread, that is ⟨read v; write v⟩ can never be re-ordered. Note that this re-ordering is achieved in JMM_{old} by breaking up each read/write operation. Consequently even though operations are issued in program order, they can be completed out-of-order thereby achieving the effect of bypassing.

Since JMM_{old} freely allows re-ordering of shared variable read/writes among themselves, no memory barriers need to be inserted before read/write instructions (in the absence of lock/unlock/volatile variables) to satisfy the JMM. For the new JMM also, no memory barriers need to inserted among shared variable reads/writes for any of the hardware memory models.

4.3 Barriers for Volatile Reads / Writes

JMM_{old} does not allow read/writes of volatile variables to be re-ordered among themselves. However, read/write of volatile variables may be re-ordered w.r.t. read/write of normal variables. In the following pseudo-code

Thread 1	Thread 2
write a,1	readvolatile v
write a,2	read a
writevolatile v,1	

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 [21] using volatile variables.

In JMM_{MP}, this problem is rectified by assigning acquire-release semantics to volatile variable operations. Thus a volatile write cannot be re-ordered w.r.t. previous reads and writes; similarly a volatile read cannot be re-ordered w.r.t. following reads and writes. However certain other re-orderings are allowed (*e.g.* a volatile read can bypass previous normal reads/writes). Re-ordering requirements for volatile reads / writes w.r.t. normal reads / writes in order to satisfy JMM_{MP} are exactly as in Table 1(b) (volatile read is like lock and volatile write is like unlock). In addition, volatile variable reads/writes are not allowed to bypass each other (just like synchronization operations). This is simply ensured by treating volatile reads and writes as analogous to acquire and release operations, which have special support at the level of hardware memory models.

This clever improvement in the volatile variable semantics in JMM_{MP} provides additional safety without introducing substantial performance overheads (under relaxed hardware models. Note that even in JMM_{old} since volatile variable operations cannot be re-ordered, the compiler has to introduce memory barriers before volatile reads/writes (thus the read of a volatile variable cannot read from the write buffer). This is because a memory barriers 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 barrier which simply forces the incomplete volatile reads/writes to commit. Consequently, the re-ordering requirements of volatile reads / writes under JMM_{old} are:

	SC	TSO	PSO	WO	RC
Volatile Read	No	$Wr^{\uparrow}$	$Wr^{\uparrow}$	$RW^{\uparrow}$	$RW^{\uparrow}$
Volatile Write	No	No	$Wr^{\uparrow}$	$RW^{\uparrow}$	$RW^{\uparrow}$

This leads to substantial performance difference across the two software memory models in benchmarks involving large number of volatile variable accesses.

4.4 Barriers due to Final Fields

Final fields are the fields of an object which are written only once (in the constructor). JMM_{old} does not prescribe any special semantics for final fields, and treats them as normal variables. However, JMM_{MP} provides specialized semantics for final fields. In the following, we only consider the semantics of final fields which are not visible to other threads before the constructor terminates (called "properly constructed" final fields in JMM_{MP} [14]). This semantics requires *all* the writes (write 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 JMM_{MP} it is not just enough to measure the time taken by these additional barriers. Due to the additional safety

[1]Lock involves reading as well as writing

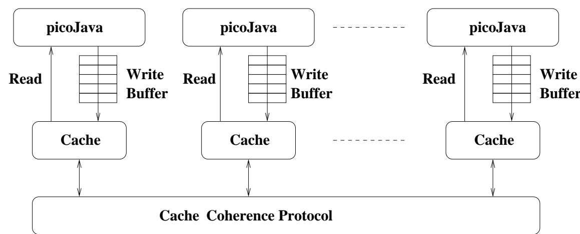


Figure 2: Multiproc. implementation of picoJava

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 to be not very substantial.

5. EXPERIMENTAL SETUP

In this section, we discuss the experimental setup used to compare the impact of the original JMM as well as the new JMM on multithreaded Java program performance. We want to study the effect of JMM_{old} and JMM_{MP} in disabling the re-orderings allowed by the various multiprocessor memory consistency models. One way to achieve this is to assume a machine with the instruction set architecture (ISA) same as that of JVM, such as picoJava microprocessor from SUN [16] (picoJava can directly execute Java bytecode). One advantage of using picoJava's execution mechanism is that we can separate out the effect of JMM from the effect of memory instructions due to the JVM implementation (such as memory management, garbage collection etc.).

Our experimental set-up is shown in Figure 2. This is very similar to the experimental setup used in [5] for evaluating the performance of various hardware memory models. We will assume that the picoJava processor is a simple in-order processor that can issue only one instruction per clock cycle. Each thread in a mutlithreaded Java program is run on a different processor for the purposes of performance evaluation. The input trace for each thread in the program is the bytecode stream for the corresponding thread generated by the Java compiler. By varying the re-orderings allowed by the multiprocessor platform, we can implement different hardware memory consistency models. For example, a write buffer with FIFO policy can implement TSO while non-FIFO usage of the write-buffer implements PSO.

Note that we are interested in the performance of the memory read/write instructions when executed in the above experimental setup. However, in case of a multiprocessor, the performance of the memory depends not only on the memory instructions of each thread (processor), but also on the interleaving of the different threads' instructions. Since we assume a bus-based multiprocessor, the time to service a memory instruction depends on (a) contents of write buffer / processor cache as well as (b) multiprocessor bus protocol & traffic on the bus and (c) delay of the memory controller in servicing a request. To evaluate the performance of various hardware memory models, we implement the MESI invalidation based cache coherence protocol on the bus.

5.1 Simulator

Our simulator contains four different kind of modules: processor, write buffer, cache and main memory. Each processor has its own corresponding write buffer and cache while all the processors share the same main memory. All the caches and the main memory forms the memory system of the simulator. This setup builds an abstract but typical multiprocessor environment. A global clock is maintained for the purpose of measuring the operation time and help the modules to cooperate with each other properly. Different modules communicate with each other through *requests*. In particular, a write buffer receives write request from its corresponding processor; a cache receives read request and write request from its corresponding processor and write buffer respectively; the shared main memory receives requests from write buffers, caches and processors.

Cache status information is also exchanged between the caches and the main memory as requested by the cache coherence protocol. A four-layer hierarchy is thus formed. In each clock cycle, the modules are invoked in order of processors, write buffers, caches and main memory. Notice that this implicitly gives higher priority to the modules that are higher in the hierarchy. All the requests exchanged between the modules in the hierarchy are time-stamped, *i.e.*, the cycle when a request is generated is sent with the request. This allows the receiver to decide if a request can be served from the same cycle. Thus it prevents scenarios where a processor writes a value to the write buffer and the write buffer flushes it to the cache in the same cycle.[2] Once a module finishes processing a request, it will change its status from busy to free, so that in the next cycle, the sender of the request knows that the request has been processed.

Processor. The processor is in the highest level of the simulator architecture, each processor executes the JVM bytecode trace of *one* thread. A processor executes an JVM instruction according to the type of the instruction. Generally, we divide all the JVM instructions into two kinds. One kind is the *memory access* instruction which requires an memory operation. We have listed all the JVM opcodes which form this kind of instructions in Table 3. The rest belongs to the other kind, *non-memory access* instructions. We assign each of the non-memory access instruction opcode a cycle number as in Table 2, when processing this kind of instructions, the processor simply spends the corresponding cycle count.

Write Buffer. Write buffer serves as a connection between processor and memory system (cache and memory). In our simulator, the write buffer is implemented as a queue with the size of 5. Based on different hardware memory models, two kinds of write buffer are implemented: in-order write buffer and out-of-order write buffer.

Cache. We have implemented a 2-way set associative cache with 2^{11} sets and the MESI cache coherence protocol. The cache is modeled as an array of cache sets and each cache set contains 2 cache blocks which have 2^5 bytes each.

[2]Without the timestamps this can happen since in any cycle the processor is invoked first, followed by write buffer, and then cache.

OPCODE	CYCLES	OPCODE	CYCLES
DADD	3	FDIV	8
DCMPG	3	FMUL	4
DCMPL	3	FNEG	3
DDIV	8	FREM	8
DMUL	4	FSUB	3
DNEG	3	IDIV	8
DREM	8	IMUL	4
DSUB	3	IREM	8
FADD	3	LDIV	8
FCMPG	3	LMUL	4
FCMPL	3	LREM	8
other non-memory access opcodes	1		

Table 2: Cycles assigned to non-memory access opcodes

READ	WRITE	SYNCHRONIZATION
GETFIELD	PUTFIELD	MONITORENTER
GETSTATIC	PUTSTATIC	MONITOREXIT
AALOAD	AASTORE	
BALOAD	BASTORE	
CALOAD	CASTORE	
DALOAD	DASTORE	
FALOAD	FASTORE	
IALOAD	IASTORE	
LALOAD	LASTORE	
SALOAD	SASTORE	

Table 3: Memory operations in JVM opcodes

Main Memory. In our simulator, lock variables are not cached. Thus, the requests reaching the main memory module are: lock requests from processors, unlock requests from write buffers and read/write requests from caches.

5.2 Benchmarks

We chose five different multithreaded Java Grande benchmarks: SYNC, LU, SOR, SERIES and RAYTRACER which are suitable for parallel execution on shared memory multiprocessors [9]. The SYNC benchmark is a low-level operation, it measures performance of synchronized methods and synchronized blocks. LU, SERIES and SOR are moderate-sized kernels. In particular, LU solves an 40×40 linear system using LU factorization followed by a "triangular solve" operation. It is a Java version of the well known Linpack benchmark. The SOR benchmark performs 100 iterations of successive over-relaxation on a 50×50 grid. The SERIES Benchmark computes the first 30 Fourier coefficients of the function $f(x) = (x + 1)^x$ on the interval $0 \ldots 2$. The RAYTRACER benchmark is a *large scale application*, it measures the performance of a 3D raytracer. The scene rendered contains 64 spheres, and is rendered at a resolution of 5×5 pixels. The LU and SOR benchmarks have substantial number of volatile variable reads/writes, accounting for $2 - 15\%$ of the operations.

Table 4 shows the number of volatile reads/writes, synchronization operations and final field writes in the traces of our benchmarks. These traces are the input to our simulator. Each benchmark is run with four parallel threads.

6. EXPERIMENTAL RESULTS

In this section, we compare the impact of the original JMM as well as the new JMM on multithreaded Java program performance. 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 old and new JMM (JMM_{old} and JMM_{MP}).

Hardware Memory Models. We consider the following hardware memory models: SC, TSO, PSO, WO and RC. Beforehand, we divide the five consistency models into three groups according to their levels of strictness. **Group A** contains only SC which is the strictest consistency model which does not allow any instruction bypassing. **Group B**, formed of TSO and PSO, is more relaxed than Group A since TSO and PSO allow non-blocking writes. **Group C**, formed of WO and RC, is the least strict group which allows non-blocking reads as well as non-blocking writes.

6.1 CPU Utilization

CPU utilization of a consistency model is measured by the ratio of necessary CPU cycles to the total running time which is also recorded in cycles. Note that "necessary CPU cycles" denote the least number of CPU cycles to execute an instruction trace, not including the memory barrier cycles or the CPU stall cycles waiting for memory operations to complete.

The CPU utilization numbers are summarized in Figure 3. These numbers were obtained by running each benchmark with four threads. For each benchmark we show the CPU utilization across the multiprocessor consistency models when (a) no software memory model is enforced, (b) JMM_{old} is enforced, and (c) JMM_{MP} is enforced. Clearly, the multiprocessor consistency models in Group C perform better than Group A and B.

The results show that for SC, the CPU utilizations are similar under JMM_{old} and JMM_{MP}. However, for the relaxed consistency models, the CPU utilizations have differences under the two different software memory models: JMM_{MP} performs better than JMM_{old}. This is particularly the case for benchmarks with large number of volatile operations such as SOR. The volatile variable semantics of the two JMMs are substantially different: JMM_{old} inserts barriers before volatile reads/writes for most hardware consistency models. This is not the case for JMM_{MP}. The difference between the volatile variable semantics of the two JMMs shows up significantly in *all* the relaxed multiprocessor consistency models. These models can take advantage of the relaxed volatile variable semantics of JMM_{MP} which allows re-orderings such as: "volatile reads can bypass normal reads/writes".

To verify the above claim, we also checked the distance between normal reads/writes and volatile reads in the SOR and LU benchmark. Volatile reads comprise 14% of the total operations in SOR. Furthermore, 98% of these volatile reads are within a distance of 5 from a normal read/write, that is, there are four or less bytecodes between a volatile read and its immediately preceding normal read/write. Consequently, significant performance improvement is obtained by allowing volatile reads to bypass normal reads/write as in JMM_{MP}. As for LU, the other benchmark with substantial number of volatile operations, volatile reads comprise 2% of

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

Table 4: Characteristics of Benchmarks used

the operations. Furthermore, about 60% of these volatile reads are within a distance of 5 from a normal read/write. Consequently the benefits in CPU utilization by adopting JMM_{MP} instead of JMM_{old} are less substantial in LU.

6.2 Read Miss Stall Cycles

Read miss stall cycles are the total number of CPU stall cycles due to read misses. Figure 4 shows the differences of read miss stalling across the consistency models.

We can see from Figure 4 that for benchmarks with large number of volatile operations, in Groups B and C, JMM_{MP} has less read miss stall cycles than JMM_{old}. In fact, the numbers for JMM_{MP} are quite close to those without any JMM being enforced. This is because of the difference in volatile variable semantics under the two JMMs: JMM_{MP} allows a volatile read operation to bypass its previous normal write operation. However, this bypassing is not allowed in JMM_{old}. As a result, a read volatile operation is more likely to be read hit in JMM_{MP} than JMM_{old} (a blocked read in processor p_i is more likely to be a miss because the cached copy of p_i may be invalidated by a write in another processor p_j while the read in p_i is blocked). Consequently, the read miss stall cycles are less in JMM_{MP}.

6.3 Write Miss Stall Cycles

Write miss stall cycles are the total number of CPU stall cycles due to write misses. The results are summarized in Figure 5 but because the SC has much more write miss stall cycles than other consistency models, we ignore SC in the figure to show the difference of other consistency models more clearly. Table 5 shows the write miss stall cycles under SC enforced by no software memory model, JMM_{old} and JMM_{MP}.

Benchmark	SC		
	No JMM	JMM_{old}	JMM_{MP}
LU	399574	399574	399574
SOR	110836	110836	110836
SERIES	6588	6588	6588
SYNC	43059	43059	43059
RAYTRACER	1350287	1350287	1350287

Table 5: Write Miss Stall Cycles of SC

Figure 5 shows that under RC, write miss stall cycles under JMM_{MP} are somewhat less than those under JMM_{old}. This is because the RC model takes advantage of some of the relaxations of the JMM_{MP}. For example the JMM_{MP} allows lock to bypass normal read or normal write which is not allowed by JMM_{old}. Another example is that JMM_{MP} allows volatile read operation to bypass normal write and this again is not allowed by JMM_{old}. The advantage in terms of write stall cycles is however not very drastic under the benchmarks which do not have large number of volatile memory operations.

6.4 Synchronization Stall Cycles

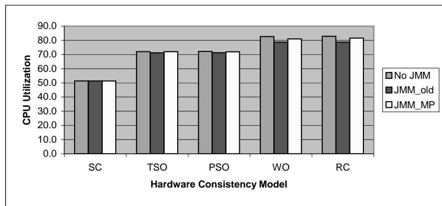
Synchronization stall cycles are the total number of CPU stall cycles due to synchronization operations: lock and unlock. Figure 6 shows the synchronization stall cycles under JMM_{old} and JMM_{MP}. Under JMM_{old}, the synchronization stall cycles are quite close in all the relaxed consistency models. This is because JMM_{old} requires that synchronization operations can not bypass all the previous memory operations and all the memory operations cannot bypass their previous synchronization operations. But JMM_{MP} allows (1) lock to bypass the previous memory operation. (2) unlock to be bypassed by the following memory operation. This is shown by the fact that under JMM_{MP}, there are less synchronization stall cycles under PSO and RC.

6.5 Memory Barrier Processing Cycles

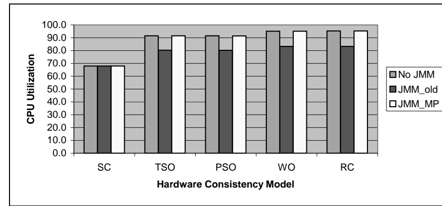
Since our benchmarks do not contain too many synchronization operations, most of the memory barriers are introduced due to volatile variable accesses and final fields. These barriers are introduced in order to ensure compliance to JMM_{old} and JMM_{MP}. Memory barrier processing cycles are the total number of CPU cycles that were contributed to the executing memory barrier instructions (1 CPU cycle each) and waiting for the finish of other operations as required by the memory barriers. Table 6 shows the memory barrier processing cycles under JMM_{old} and JMM_{MP}. SC does not involve any memory barrier because it is more strict than any of the two JMMs. In the relaxed hardware models, the time overhead due to memory barriers is substantially less under JMM_{MP} as compared to JMM_{old}. Under the JMM_{old} with the benchmarks which have large number of volatile variables, we can observe that the hardware consistency models PSO, WO, and RC need more cycles due to barriers than TSO. This is because TSO introduces memory barrier before volatile read operation while PSO, WO and RC introduce memory barriers before both volatile read and volatile write operations under JMM_{old}.

6.6 Difference in Volatile Variable Semantics

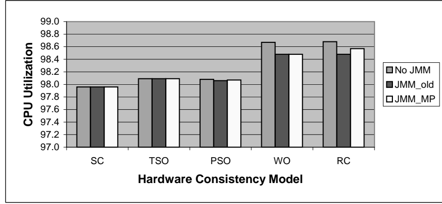
Because of the different semantics for volatile operations of JMM_{old} and JMM_{MP}, it is important to study the performance of volatile operations under the two different JMMs. For this purpose, we select the two benchmarks in our benchmark suite which have substantial number of volatile operations: LU and SOR.



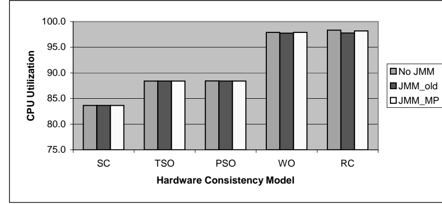
LU



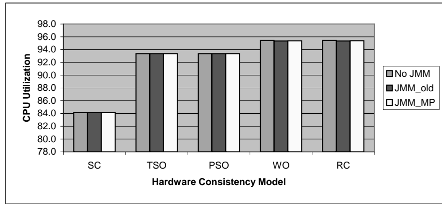
SOR



SERIES

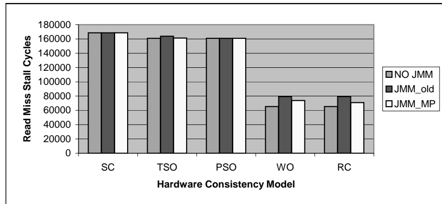


SYNC

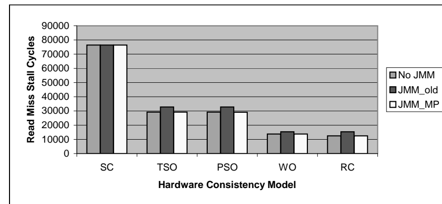


RAYTRACER

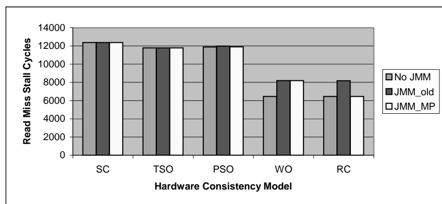
Figure 3: CPU Utilization percentage in different memory models



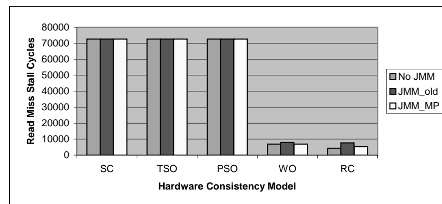
LU



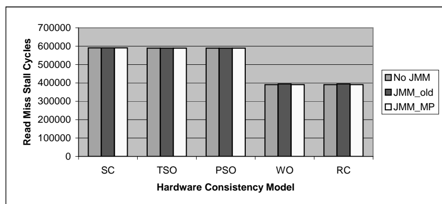
SOR



SERIES

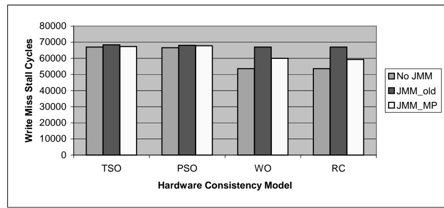


SYNC

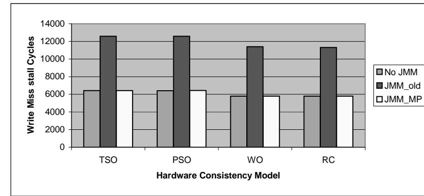


RAYTRACER

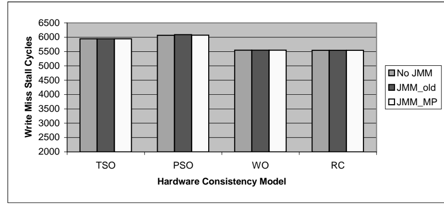
Figure 4: Read Miss Stall Cycles in different memory models



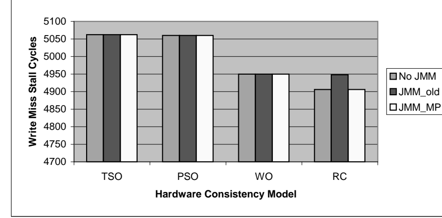
LU



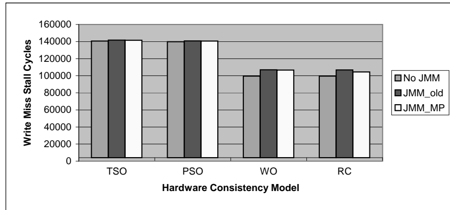
SOR



SERIES

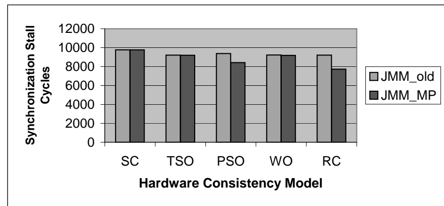


SYNC

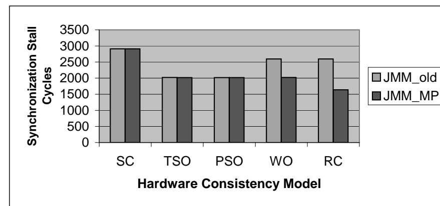


RAYTRACER

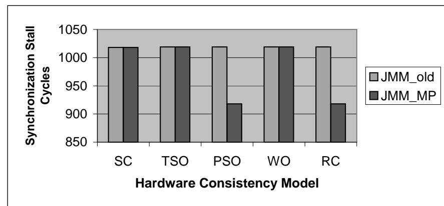
Figure 5: Write Miss Stall Cycles in different memory models



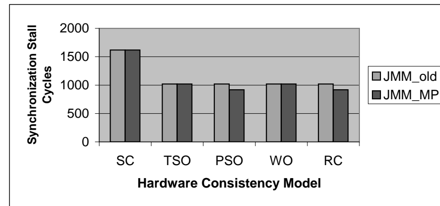
LU



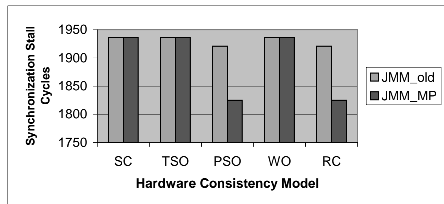
SOR



SERIES



SYNC

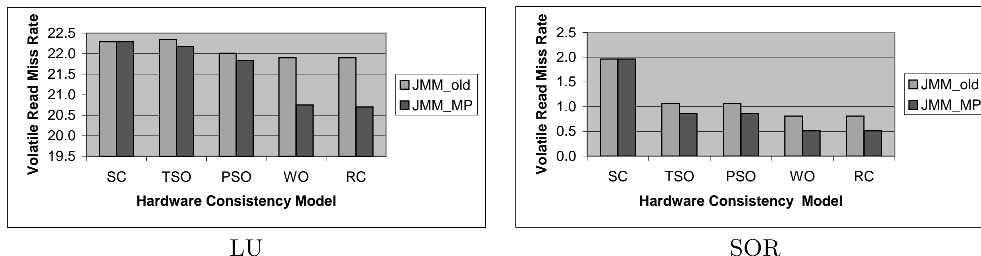


RAYTRACER

Figure 6: Synchronization Stall Cycles in different memory models

Benchmark	TSO		PSO		WO		RC	
	old	MP	*old*	MP	*old*	MP	*old*	MP
LU	12182	76	13695	1340	24095	428	26456	455
SOR	61295	32	61900	696	62956	144	63262	161
SERIES	14	36	27	39	0	187	30	187
SYNC	17	6	34	10	0	34	44	36
RAYTRACER	96	1132	149	1177	196	1997	268	2016

Table 6: Memory Barrier Processing Cycles in different memory models



LU SOR

Figure 7: Volatile Read Miss Rate percentage in different memory models

Volatile Read Miss Rate. Volatile read miss rate is the ratio of total volatile read misses to the total number of volatile read operations. Figure 7 shows the volatile read miss rate of LU and SOR. In SC, volatile read miss rates are the same both under JMM_{old} and JMM_{MP}. However, in all the relaxed hardware consistency models, volatile read miss rates are lower under JMM_{MP} than JMM_{old} (*i.e.*, a volatile read operation are more likely to be a read hit under JMM_{MP} than JMM_{old}). This result is compatible to the results in section 6.2 and seems to confirm the reason we gave for the better performance of JMM_{MP} to JMM_{old} with regard to volatile variable accesses.

Volatile Processing Cycles. Volatile processing cycles are the total number of CPU cycles that were contributed to the processing of volatile operations including the busy cycles to execute volatile operations and stall cycles caused by volatile operations such as memory barrier cycles due to volatile operations. Figure 8 shows the volatile processing cycles under JMM_{old} and JMM_{MP}. The figure indicates that under all the relaxed hardware consistency models, less CPU cycles are contributed to the volatile operations with JMM_{MP} than JMM_{old}. This fact is caused by two reasons.

- To ensure there is no violation of JMM, more memory barriers due to volatile accesses are added to enforce JMM_{old} than JMM_{MP}, consequently, more CPU cycles are needed to process these memory barriers.
- Under JMM_{old} and a relaxed hardware model M, volatile reads are more likely to be misses, as compared to under JMM_{MP} and the same hardware model M. The explanation for this was given in Section 6.2.

6.7 Final Fields

In JMM_{old}, final fields are treated as normal variables so no memory barriers need to be inserted due to final fields. To enforce JMM_{MP}, we insert barrier to the end of constructor with final field writes. Among the benchmarks, RAYTRACER has substantial number of constructors with final field writes (768 in total). We measured the memory barrier processing cycles due to final field writes in the RAYTRACER with JMM_{MP} as software memory model and TSO, PSO, WO, RC as hardware memory models. The overall stall due to final field barriers in RAYTRACER is not very high: less than 2100 cycles for 768 barriers (under TSO, PSO, WO, RC). Since one cycle is spent in executing each barrier, this means an average stall of $(2100/768 - 1)$, that is, approximately two cycles per barrier.

7. DISCUSSION

In this paper we have studied the performance impact of Java Memory Model on hardware consistency models of multiprocessor platforms. A hardware consistency model describes the behaviors allowed by multiprocessor implementations while Java Memory Model (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 on the multiprocessor with some hardware consistency model does not violate the JMM, we add memory barriers to enforce ordering. A multiprocessor simulator is implemented to simulate the execution of multithreaded Java Grande Benchmarks under different software memory models and hardware consistency models. The results show that the new JMM achieves a performance improvement (as compared to the old JMM) with regard to the volatile variable accesses. Overall, the new JMM performs quite close to the case where no JMM is enforced.

In terms of future work, our study needs to be extended to other proposals for new JMM such as [13] and [2]. Furthermore, for more comprehensive simulation results we need to simulate the execution of multi-threaded Java benchmarks on out-of-order processors (as in the RSIM multi-processor simulator [8]). We also need to consider other commercial hardware multiprocessor platforms for our experiments such as DEC-Alpha, PowerPC and IA64.

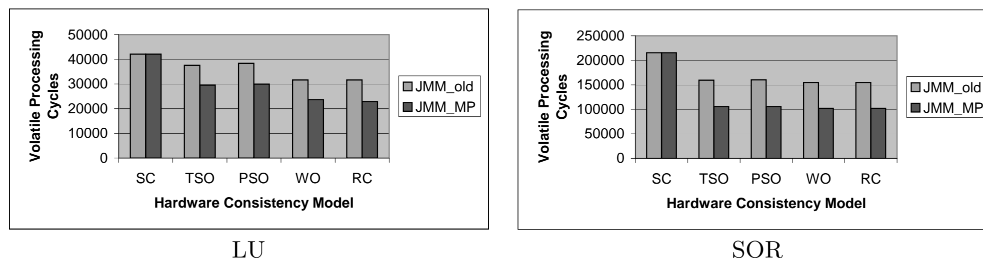


Figure 8: Volatile Processing Cycles in different memory models

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

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

[4] David L. Weaver and Tom Germond, Prentice Hall Publishers. *The SPARC Architecture Manual : Version 9*, 1994.

[5] 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.

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

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

[8] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer: Special issue on high performance simulators*, 35(2), 2002.

[9] JGF. 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`.

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

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

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

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

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

[15] J. Manson and W. Pugh. Semantics of multithreaded Java. Technical report, Department of Computer Science, University of Maryland, College Park, CS-TR-4215, 2002.

[16] SUN Microsystems. picojava[tm] microprocessor cores, 1998. `http://www.sun.com/microelectronics/picoJava/`.

[17] V.S. Pai, P. Ranganathan, S.V. 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.

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

[19] A. Roychoudhury. Formal reasoning about hardware and software memory models. In *International Conference on Formal Engineering Methods (ICFEM), LNCS 2495*. Springer Verlag, 2002.

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

[21] 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.

[22] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Formalizing the Java memory model for multithreaded program correctness and optimization. Technical report, School of Computing, University of Utah, April 2002.

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