IMPACT OF JAVA MEMORY MODEL
ON OUT-OF-ORDER MULTIPROCESSORS

SHEN QINGHUA

NATIONAL UNIVERSITY OF SINGAPORE
2004
IMPACT OF JAVA MEMORY MODEL
ON OUT-OF-ORDER MULTIPROCESSORS

SHEN QINGHUA
(B.Eng., Tsinghua University)

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2004
Acknowledgements

I owe a debt of gratitude to many people for their assistance and support in the preparation of this thesis. First I should like to thank my two supervisors, Assistant Professor Abhik Roychoudhury and Assistant Professor Tulika Mitra. It is them who guided me into the world of research, gave me valuable advice on how to do research and encouraged me to overcome various difficulties throughout my work. Without their help, the thesis can not be completed successfully.

Next, I am especially grateful to the friends in the lab, Mr. Xie Lei, Mr. Li Xianfeng and Mr. Wang Tao, many thanks for their sharing research experience and discussing all kinds of questions with me. It is their supports and encouragements that helped me solve lots of problems.

I also would like to thank Department of Computer Science, the National University of Singapore for providing me research scholarship and excellent facilities to study here. Many thanks to all the staffs.

Last but not the least, I am deeply thankful to my wife and my parents, for their loves, cares and understandings through my life.
Contents

Acknowledgements i

List of Tables v

List of Figures vi

Summary viii

1 Introduction 1

1.1 Overview ......................................................... 1
1.2 Motivation ..................................................... 1
1.3 Contributions ................................................... 3
1.4 Organization .................................................... 5

2 Background and Related Work 6

2.1 Hardware Memory Model ........................................ 6
2.1.1 Sequential Consistency ...................................... 7
2.1.2 Relaxed Memory Models .................................... 9
2.2 Software Memory Model ........................................ 12
2.2.1 The Old JMM ................................................ 13
2.2.2 A New JMM ......................................................... 16
2.3 Other Related Work ................................................. 20

3 Relationship between Memory Models ........... 22
  3.1 How JMM Affect Performance ............................. 22
  3.2 How to Evaluate the Performance ....................... 26

4 Memory Barrier Insertion ............................. 29
  4.1 Barriers for normal reads/writes ......................... 31
  4.2 Barriers for Lock and Unlock ............................ 32
  4.3 Barriers for volatile reads/writes ....................... 36
  4.4 Barriers for final fields ................................ 38

5 Experimental Setup .................................. 39
  5.1 Simulator ....................................................... 41
    5.1.1 Processor .............................................. 41
    5.1.2 Consistency Controller .............................. 42
    5.1.3 Cache .................................................. 44
    5.1.4 Main Memory .......................................... 45
    5.1.5 Operating System .................................... 46
    5.1.6 Configuration and Checkpoint ..................... 46
  5.2 Java Virtual Machine .................................... 47
  5.3 Java Native Interface ..................................... 48
  5.4 Benchmarks ................................................ 50
  5.5 Validation .................................................. 51

iii
6 Experimental Results

6.1 Memory Barriers ...................................................... 54
6.2 Total Cycles ........................................................... 57

7 Conclusion and Future Work ........................... 66

7.1 Conclusion ............................................................ 66
7.2 Future Work .......................................................... 67
List of Tables

4.1 Re-orderings between memory operations for $JMM_{new}$ . . . . . . . 32
4.2 Memory Barriers Required for Lock and Unlock Satisfying $JMM_{old}$ 33
4.3 Memory Barriers Required for Lock and Unlock Satisfying $JMM_{new}$ 35
4.4 Memory Barriers Required for Volatile Variable Satisfying $JMM_{old}$ 37
4.5 Memory Barriers Required for Volatile Variable Satisfying $JMM_{new}$ 38
6.1 Characteristics of benchmarks used . . . . . . . . . . . . . . . . . . . 54
6.2 Number of Memory Barriers inserted in different memory models . 56
6.3 Total Cycles for SOR in different memory models . . . . . . . . . . 59
6.4 Total Cycles for LU in different memory models . . . . . . . . . . 59
6.5 Total Cycles for SERIES in different memory models . . . . . . . . 59
6.6 Total Cycles for SYNC in different memory models . . . . . . . . . 59
6.7 Total Cycles for RAY in different memory models . . . . . . . . . . 60
### List of Figures

2.1 Programmer’s view of sequential consistency .......................... 8
2.2 Ordering restrictions on memory accesses ............................. 11
2.3 Memory hierarchy of the old Java Memory Model .................... 13
2.4 Surprising results caused by statement reordering .................... 16
2.5 Execution trace of Figure 2.4 ........................................ 19

3.1 Implementation of Java memory model ................................. 23
3.2 Multiprocessor Implementation of Java Multithreading ............... 25

4.1 Actions of lock and unlock in $JMM_{old}$ .............................. 34

5.1 Memory hierarchy of Simics ............................................ 45

6.1 Performance difference of $JMM_{old}$ and $JMM_{new}$ for SOR ...... 61
6.2 Performance difference of $JMM_{old}$ and $JMM_{new}$ for LU ...... 61
6.3 Performance difference of $JMM_{old}$ and $JMM_{new}$ for SERIES ... 62
6.4 Performance difference of $JMM_{old}$ and $JMM_{new}$ for SYNC ... 62
6.5 Performance difference of $JMM_{old}$ and $JMM_{new}$ for RAY ...... 63
6.6 Performance difference of SC and Relaxed memory models for SOR 63
6.7 Performance difference of SC and Relaxed memory models for LU 64
6.8 Performance difference of SC and Relaxed memory models for SERIES 64
6.9 Performance difference of SC and Relaxed memory models for SYNC 64
6.10 Performance difference of SC and Relaxed memory models for RAY 65
Summary

One of the significant features of the Java programming language is its built-in support for multithreading. Multithreaded Java programs can be run on multiprocessor platforms as well as uniprocessor ones. Java provides a memory consistency model for the multithreaded programs irrespective of the implementation of multithreading. This model is called the Java memory model (JMM). We can use the Java memory model to predict the possible behaviors of a multithreaded program on any platform.

However, multiprocessor platforms traditionally have memory consistency models of their own. In order to guarantee that the multithreaded Java program conforms to the Java Memory Model while running on multiprocessor platforms, memory barriers may have to be explicitly inserted into the execution. Insertion of these barriers will lead to unexpected overheads and may suppress/prohibit hardware optimizations.

The existing Java Memory Model is rule-based and very hard to follow. The specification of the new Java Memory Model is currently under community review. The new JMM should be unambiguous and executable. Furthermore, it should consider exploiting the hardware optimizations as much as possible.
In this thesis, we study the impact of multithreaded Java program under the old JMM and the proposed new JMM on program performance. The overheads brought by the inserted memory barriers will also be compared under these two JMMs. The experimental results are obtained by running multithreaded Java Grande benchmark under Simics, a full system simulation platform.
Chapter 1

Introduction

1.1 Overview

Multithreading, which is supported by many programming languages, has become an important technique. With multithreading, multiple sequences of instructions are able to execute simultaneously. By accessing the shared data, different threads can exchange their information. The Java programming language has a built-in support for multithreading where threads can operate on values and objects residing in a shared memory. Multithreaded Java programs can be run on multiprocessor or uniprocessor platforms without changing the source code, which is a unique feature that is not present in many other programming languages.

1.2 Motivation

The creation and management of the threads of a multithreaded Java program are integrated into the Java language and are thus independent of a specific platform.
But the implementation of the Java Virtual Machine (JVM) determines how to map the user level threads to the kernel level threads of the operating system. For example, SOLARIS operating system provides a many-to-many model called SOLARIS Native Threads, which uses lightweight processes (LWPs) to establish the connection between the user threads and kernel threads. While for Linux, the user threads can be managed by a thread library such as POSIX threads (Pthreads), which is a one-to-one model. Alternatively, the threads may be run on a shared memory multiprocessors connected by a bus or interconnection network. In these platforms, the writes to the shared variable made by some threads may not be immediately visible to other threads.

Since the implementations of multithreading vary radically, the Java Language Specification (JLS) provides a memory consistency model which imposes constraints on any implementation of Java multithreading. This model is called the Java Memory Model (henceforth called JMM)[7]. The JMM explains the interaction of threads with shared memory and with each other. We may rely on the JMM to predict the possible behaviors of a multithreaded program on any platform. However, in order to exploit standard compiler and hardware optimizations, JMM intentionally gives the implementer certain freedoms. For example, operations of shared variable reads/writes and operations of synchronization like lock/unlock within a thread can be executed completely out-of-order. Accordingly, we have to consider arbitrary interleaving of the threads and certain re-ordering of the operations in the individual thread so as to debug and verify a multithreaded Java program.
Moreover, the situation becomes more complex when multithreaded Java programs are run on shared memory multiprocessor platforms because there are memory consistency models for the multiprocessors. This hardware memory model prescribes the allowed re-orderings in the implementation of the multiprocessor platform (e.g. a write buffer allows writes to be bypassed by read). Now many commercial multiprocessors allow out-of-order executions at different level. We must guarantee that the multithreaded Java program conforms to the JMM while running on these multiprocessor platforms. Thus, if the hardware memory model is more relaxed than the JMM (which means hardware memory model allows more re-orderings than the JMM), memory barriers have to be explicitly inserted into the execution at the JVM level. Consequently, this will lead to unexpected overheads and may prohibit certain hardware optimizations. That is why we will study the performance impact of multithreaded Java programs from out-of-order multiprocessor perspective. This has become particularly important in the recent times with commercial multiprocessor platforms gaining popularity in running Java programs.

### 1.3 Contributions

The research on memory models began with hardware memory models. In the absence of any software memory model, we can have a clear understanding of which hardware memory model is more efficient. In fact, some work has been done on the processor level to evaluate the performance of different hardware memory models. The experimental results showed that multiprocessor platforms with relaxed hardware memory models can significantly improve the overall performance com-
pared to sequential consistent memory model[1]. But this study only described the impact of hardware memory models on performance. In this thesis, we study the performance impact of both hardware memory models and software memory model (JMM in our case).

To the best of our knowledge, the research of the performance impact of JMM on multiprocessor platforms mainly focused on theory but not implementations on system. The research work of Doug Lea is related to ours [6]. His work provides a comprehensive guide for implementing the newly proposed JMM. However, it only includes a set of recommended recipes for complying to the new JMM. And there is no actual implementation on any hardware platform. However, it provides 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. This will help us have a better understanding of the new JMM and provide a guideline for our implementation.

Previously, Xie Lei[15] has studied the relative performance of hardware memory models in the presence/absence of a JMM. However, he implemented a simulator to execute bytecode instruction trace under picoJava microprocessor of SUN. It is a trace-driven execution on in-order processor. In our study, we implement a more realistic system and use a execution-driven out-of-order multiprocessor platform. As memory consistency models are designed to facilitate out-of-order processing, it is very important to use out-of-order processor. We run unchanged Java codes on this system and compare the performance of these two JMMs on different hardware memory models. Our tool can also be used as a framework for estimating
Java program performance on out-of-order processors.

1.4 Organization

The rest of the thesis is organized as follows. In chapter 2, we review the background of various hardware memory models and the Java memory models and discuss the related work on JMM. Chapter 3 describes the methodology for evaluating the impact of software memory models on multiprocessor platform. Chapter 4 analyzes the relationship between hardware and software memory models and identifies the memory barriers inserted under different hardware and software memory models. Chapter 5 presents the experimental setup for measuring the effects of the JMM on a 4-processor SPARC platform. The experimental results obtained from evaluating the performance of multithreaded Java Grande benchmarks under various hardware and software memory models are given in Chapter 6. At last, a conclusion of the thesis and a summary of results are provided in Chapter 7.
Chapter 2

Background and Related Work

2.1 Hardware Memory Model

Multiprocessor platforms are becoming more and more popular in many domains. Among them, the shared memory multiprocessors have several advantages over other choices because they present a more natural transition from uniprocessors and simplify difficult programming tasks. Thus shared memory multiprocessor platforms are being widely accepted in both commercial and scientific computing. However, programmers need to know exactly how the memory behaves with respect to read and write operations from multiple processors so as to write correct and efficient shared memory programs. The memory consistency model of a shared memory multiprocessor provides a formal specification of how the memory system will present to the programmers, which becomes an interface between the programmer and the system. The impact of the memory consistency model is pervasive in a shared memory system because the model affects programmability, performance and portability at several different levels.
The simplest and most intuitive memory consistency model is sequential consistency, which is just an extension of the uniprocessor model applied to the multiprocessor case. But this model prohibits many compiler and hardware optimizations because it enforces a strict order among shared memory operations. So many relaxed memory consistency models have been proposed and some of them are even supported by commercial architectures such as Digital Alpha, SPARC V8 and V9, and IBM PowerPC. I will illustrate the sequential consistency model and some relaxed consistency models that we are concerned with in detail in the following sections.

2.1.1 Sequential Consistency

In uniprocessor systems, sequential semantics ensures that all memory operations will occur one at a time in the sequential order specified by the program (i.e., program order). For example, a read operation should obtain the value of the last write to the same memory location, where the “last” is well defined by program order. However, in the shared memory multiprocessors, writes to the same memory location may be performed by different processors, which have nothing to do with program order. Other requirements are needed to make sure a memory operation executes atomically or instantaneously with respect to other memory operations, especially for the write operation. For this reason, write atomicity is introduced, which intuitively extends this model to multiprocessors. Sequential consistency memory model for shared memory multiprocessors is formally defined by Lamport as follows[3].
4 Understanding Sequential Consistency

The most commonly assumed memory consistency model for shared memory multiprocessors is sequential consistency, formally defined by Lamport as follows [16].

**Definition:**

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

There are two aspects to sequential consistency: (1) maintaining program order among operations from individual processors, and (2) maintaining a single sequential order among operations from all processors. The latter aspect makes it appear as if a memory operation executes atomically or instantaneously with respect to other memory operations.

Sequential consistency provides a simple view of the system to programmers as illustrated in Figure 3. Conceptually, there is a single global memory and a switch that connects an arbitrary processor to memory at any time step. Each processor issues memory operations in program order and the switch provides the global serialization among all memory operations.

Figure 4 provides two examples to illustrate the semantics of sequential consistency. Figure 4(a) illustrates the importance of program order among operations from a single processor. The code segment depicts an implementation of Dekker’s algorithm for critical sections, involving two processors (P1 and P2) and two flag variables (Flag1 and Flag2) that are initialized to 0. When P1 attempts to enter the critical section, it updates Flag1 to 1, and checks the value of Flag2. The value 0 for Flag2 indicates that P2 has not yet tried to enter the critical section; therefore, it is safe for P1 to enter. This algorithm relies on the assumption that a value of 0 returned by P1’s read implies that P1’s write has occurred before P2’s write and read operations. Therefore, P2’s read of the flag will return the value 1, prohibiting P2 from also entering the critical section. Sequential consistency ensures the above by requiring that program order among the memory operations of P1 and P2 be maintained, thus precluding the possibility of both processors reading the value 0 and entering the critical section.

Figure 4(b) illustrates the importance of atomic execution of memory operations. The figure shows three processors sharing variables A and B, both initialized to 0. Suppose processor P2 returns the value 1 (written by P1) for its read of A, writes to variable B, and processor P3 returns the value 1 (written by P2) for B. The atomicity aspect of sequential consistency allows us to assume the effect of P1’s write is seen by the entire system at the same time. Therefore, P3 is guaranteed to see the effect of P1’s write in the above execution and must return the value 1 for its read of A (since P3 sees the effect of P2’s write after P2 sees the effect of P1’s write to A).

5 Implementing Sequential Consistency

This section describes how the intuitive abstraction of sequential consistency shown in Figure 3 can be realized in a practical system. We will see that unlike uniprocessors, preserving the order of operations on a per-location basis...
Sequential consistency provides a simple view of the system to programmers as illustrated in Figure 2.1. From that, we can think of the system as having a single global memory and a switch that connects only one processor to memory at any time step. Each processor issues memory operations in program order and the switch ensures the global serialization among all the memory operations.

2.1.2 Relaxed Memory Models

Relaxed memory consistency models are alternatives to sequential consistency and have been accepted in both academic and industrial areas. By enforcing less restrictions on shared-memory operations, they can make a better use of the compiler and hardware optimizations. The relaxation can be introduced to both program order requirement and write atomicity requirement. With respect to program order relaxations, we can relax the order from a write to a following read, between two writes, and finally from a read to a following read or write. In all cases, the relaxation only applies to operation pairs with different addresses. With respect to write atomicity requirements, we can allow a read to return the value of another processor’s write before the write is made visible to all other processors. In addition, we need to regard lock/unlock as special operations from other shared variable read/write and consider relaxing the order between a lock and a preceding read/write, and between a unlock and a following read/write.

Here we are only concerned with 4 relaxed memory models, which are Total Store Ordering, Partial Store Ordering, Weak Ordering and Release Consistency listed by order of relaxation.
Total Store Ordering (henceforth called TSO) is a relaxed model that allows a read to be reordered with respect to earlier writes from the same processor. While the write miss is still in the write buffer and not yet visible to other processors, a following read can be issued by the processor. The atomicity requirement for writes can be achieved by allowing a processor to read the value of its own write early, and prohibiting a processor from reading the value of another processor’s write before the write is visible to all the other processors [1]. Relaxing the program order from a write followed by a read can improve performance substantially at the hardware level by effectively hiding the latency of write operations [2]. However, this relaxation alone isn’t beneficial in practice for compiler optimizations [1].

Partial Store Ordering (henceforth called PSO) is designed to further relax the program order requirement by allowing the reordering between writes to different addresses. It allows both reads and writes to be reordered with earlier writes by allowing the write buffer to retire writes out of program order. This relaxation enables that writes to different locations from the same processor can be pipelined or overlapped and are permitted to be completed out of program order. PSO uses the same scheme as TSO to satisfy the atomicity requirement. Obviously, this model further reduces the latency of write operations and enhances communication efficiency between processors. Unfortunately, the optimizations allowed by PSO are not so flexible so as to be used by a compiler [1].

Weak Ordering (henceforth called WO) uses a different way to relax the order of memory operations. The memory operations are divided into two types: data operations and synchronization [1]. Because reordering memory operations to data
regions between synchronization operations doesn’t typically affect the correctness of a program, we need only enforce program order between data operations and synchronization operations. Before a synchronization operation is issued, the processor waits for all previous memory operations in the program order to complete and memory operations that follow the synchronization operation are not issued until the synchronization completes. This model ensures that writes always appear atomic to the programmer so write atomicity requirement is satisfied [1].

Release Consistency (henceforth called RC) further relaxes the order between data operations and synchronization operations and needs further distinctions between synchronization operations. Synchronization operations are distinguished as acquire and release operations. An acquire is a read memory operation that is performed to gain access to a set of shared locations (e.g., a lock operation). A release is a write operation that is performed to grant permission for access to a
set of shared locations (e.g., a unlock operation). An acquire can be reordered with respect to previous operations and a release can be reordered with respect to following operations. In the models of WO and RC, a compiler has the flexibility to reorder memory operations between two consecutive synchronization and special operations [8].

Figure 2.2 illustrates the five memory models graphically and shows the restrictions imposed by these memory models. From the figure we can see the hardware memory models become more and more relaxed since there are less constraints imposed on them.

2.2 Software Memory Model

Software memory models are similar to hardware memory models, which are also a specification of the re-ordering of the memory operations. However, since they present at different levels, there are some important difference. For example, processors have special instructions for performing synchronization (e.g., lock/unlock) and memory barrier (e.g., membar); while in a programming language, some variables have special properties (e.g., volatile or final), but there is no way to indicate that a particular write should have special memory semantics [7]. In this section, we present the memory model of the Java programming language, Java memory model (henceforth called JMM) and compare the current JMM and a newly proposed JMM.
Figure 2.3: Memory hierarchy of the old Java Memory Model

2.2.1 The Old JMM

The old JMM, i.e. the current JMM, is described in Chapter 17 of the Java Language Specification [4]. It provides a set of rules that guide the implementation of the Java Virtual Machine (JVM), and explains the interaction of threads with the shared main memory and with each other.

Let us see the framework of the JMM first. Figure 2.3 shows the memory hierarchy of the old JMM. A main memory is shared by all threads and it contains the master copy of every variable. Each thread has a working memory where it keeps its own working copy of variables which it operates on when the thread executes a program. The JMM specifies when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy and vice versa.
Some new terms are defined in the JMM to distinguish the operations on the local copy and the master copy. Suppose an action on variable $v$ is performed in thread $t$. The detailed definitions are as follows [4, 13]:

- **use$_t$(v)**: Read from the local copy of $v$ in $t$. This action is performed whenever a thread executes a virtual machine instruction that uses the value of a variable.

- **assign$_t$(v)**: Write into the local copy of $v$ in $t$. This action is performed whenever a thread executes a virtual machine instruction that assigns to a variable.

- **read$_t$(v)**: Initiate reading from master copy of $v$ to local copy of $v$ in $t$

- **load$_t$(v)**: Complete reading from master copy of $v$ to local copy of $v$ in $t$

- **store$_t$(v)**: Initiate Writing from master copy of $v$ to local copy of $v$ in $t$

- **write$_t$(v)**: Complete Writing from master copy of $v$ to local copy of $v$ in $t$

Besides these, each thread $t$ may perform lock/unlock on shared variable, denoted by lock$_t(t)$ and unlock$_t(t)$ respectively. Before unlock, the local copy is transferred to the master copy through store and write actions. Similarly, after lock actions the master copy is transferred to the local copy through read and load actions. These actions are atomic themselves. But data transfer between the local and the master copy is not modeled as an atomic action, which reflects 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 actions of use, assign, lock and unlock are dictated by the semantics of the program. And the actions of load, store, read and write are performed by the underlying implementation at proper time, subject to temporal ordering constraints specified in the JMM. These constraints describe the ordering requirements between these actions including rules about variables, about locks, about the interaction of locks and variables, and about volatile variables etc. However, these ordering constraints seem to be a major difficulty in reasoning about the JMM because they are given in an informal, rule-based, declarative style [6]. Research papers analyzing the Java memory model interpret it differently and some disagreements even arise while investigating some of its features. In addition to the difficulty in understanding, there are two crucial problems in the current JMM: it is too weak somewhere and it is too strong somewhere else. It is too strong in that it prohibits many compiler optimizations and requires many memory barriers on some architectures. It is too weak in that much of the code that has been written for Java, including code in Sun’s Java Development Kit (JDK), is not guaranteed to be valid according to the JMM [11].

Clearly, a new JMM is in need to solve these problems and make everything unambiguous. At present time, the proposed JMM is under community review [5] and is expected to revise substantially Chapter 17 of ”The Java Language Specification” (JLS) and Chapter 8 of ”The Java Virtual Machine Specification”.

15
2.2.2 A New JMM

The revisions of the JMM are contributions of the research efforts from a number of people. Doug Lea discussed the impact of the JMM on concurrent programming in section 2.2.7 of his book, Concurrent Programming in Java 2nd edition [7] and also proposed revision to Wait Sets and Notification, section 17.4 of the JLS. Jeremy Manson and William Pugh provided a new semantics for multithreaded Java programs that allows aggressive compiler optimization, and addressed the safety and multithreaded issues [10]. Jan-Willem Maessen, Arvind and Xiaowei Shen described alternative memory semantics for Java programs and used an enriched version of the Commit/Reconcile/Fence (CRF) memory model [18].

The aim of the JMM revisions is to make the semantics of correctly synchronized multithreaded Java programs as simple and intuitive as feasible, and ensure the semantics of incompletely synchronized programs are defined securely so that such programs can’t be used to attack the security of a system. Additionally, it should be possible for the implementation of JVM to obtain high performance across a wide range of popular hardware architectures.
However, we should know that optimizations allowed by the Java programming language may produce some paradoxical behaviors for incorrectly synchronized code. To see this, consider, for example, Figure 2.4. This program contains local variables $r1$ and $r2$; it also contains shared variables $A$ and $B$, which are fields of an object. It may appear that the result $r2 == 2$, $r1 == 1$ is impossible. Intuitively, if $r2$ is 2, then instructions 2 came before instruction 3. So, if $r2 == 2$ and $r1 == 1$, then instruction 4 came before instruction 1, which comes before instruction 2, which came before instruction 3, which came before instruction 4. This is obviously impossible. However, compilers are allowed to reorder the instructions in each thread. If instruction 3 is made to execute after instruction 4, and instruction 1 is made to execute after instruction 2, then result $r2 == 2$ and $r1 == 1$ is quite reasonable.

It seems that this behavior is caused by Java. But in fact the code is not properly synchronized. We can see there is a write in one thread and a read of the same variable by another thread. And the write and read are not ordered by synchronization. This situation is called data race. It is often possible to have such surprising results when code contains a data race. Although this behavior is surprising, it is allowed by most JVMs [5]. That is one important reason that the original JMM needed to be replaced.

The new JMM gives a new semantics of multithreaded Java programs, including a set of rules on what value may be seen by a read of shared memory that is written by other thread. It works by examining each read in an execution trace and checking that the write observed by that read is legal. Informally, a read $r$ can see the value
of any write \( w \) such that \( w \) doesn’t occur after \( r \) and \( w \) is not seen to be overwritten by another write \( w' \) (from \( r \)’s perspective) [16].

The actions within a thread must obey the semantics of that thread, called intra-thread semantics, which are defined in the remainder of the JLS. However, threads are influenced by each other, so reads from one thread can return values written by writes from other threads. The new JMM provides two main guarantees for the values seen by reads, \textit{Happens-Before Consistency} and \textit{Causality}.

\textit{Happens-Before Consistency} requires that behavior is consistent with both intra-thread semantics and the write visibility enforced by the happens-before ordering [5]. To understand it, let’s see two definitions first.

\textbf{Definition 2.2} If we have two actions \( x \) and \( y \), we use \( x \xrightarrow{\text{hb}} y \) to represent \( x \) happens before \( y \). if \( x \) and \( y \) are actions of the same thread and \( x \) comes before \( y \) in program order, then \( x \xrightarrow{\text{hb}} y \).

The happens-before relationship defines a partial order over the actions in an execution trace; one action is ordered before another in the partial order if one action happens-before the other.

\textbf{Definition 2.3} A read \( r \) of a variable \( v \) is allowed to observe a write \( w \) to \( v \) if, in the happens-before partial order of the execution trace: \( r \) is not ordered before \( w \) (i.e., it is not the case that \( r \rightarrow w \)), and there is no intervening write \( w' \) to \( v \) (i.e., no write \( w' \) to \( v \) such that \( w \rightarrow w' \rightarrow r \)).

A read \( r \) is allowed to see the result of a write \( w \) if there is no happens-before ordering to prevent that read. An execution trace is \textit{happens-before consistent} if
all of the reads in the execution trace are allowed.

Figure 2.5 shows an example of this simple model and the corresponding program is in Figure 2.4. The solid lines represent happens-before relations between two actions. The dotted lines between a write and a read indicate a write that the read is allowed to see. For example, the read at $r1 = B$ is allowed to see the write at $B = 0$ or the write $B = 1$. An execution is happens-before consistent, and valid according to the Happens-Before Consistency, if all reads see writes they are allowed to see. So, for example, an execution that has the result $r1 == 1$ and $r2 == 2$ would be a valid one.

The constraints of Happens-Before Consistency are necessary but not sufficient. It is too weak for some programs and can allow situations in which an action causes
itself to happen. To avoid problems like this, \textit{causality} is brought in and should be respected by executions. \textit{Causality} means that an action cannot cause itself to happen [5]. In other words, it must be possible to explain how an execution occurred and no values can appear out of thin air. The formal definition of causality in a multithreaded context is tricky and subtle; so we are not going to present it here.

Apart from these two guarantees, new semantics are provided for final fields, double and long variables, and wait sets and notification etc. Let’s take the treatment of final fields as an example. The semantics of final fields are somewhat different from those of normal fields. Final fields are initialized once and never changed, so the value of a final field can be kept in a cache and needn’t be reloaded from main memory. Thus, the compiler is given a great deal of freedom to move the read of final fields [5]. The model for final fields is simple and the detail is as follows. Set the final fields for an object in that object’s constructor. Do not write a reference to the object being constructed in a place where another thread can see it before the object is completely initialized [5]. When the object is seen by another thread, that thread will always see the correctly constructed version of that object’s final fields.

\section{2.3 Other Related Work}

The hardware memory model has been studied extensively. There are various simulators for multiprocessors from execution-driven to full system. The performance of different hardware memory models can be evaluated using these simulators.
The research results show that the hardware memory models influence the performance substantially [1] and the performance can be improved dramatically with pre-fetching and speculative loads [2]. Pai et al. studied the implementation of SC and RC models under current multiprocessors with aggressive exploitation of instruction level parallelism (ILP) [19]. They found the performance of RC significantly outperforms that of SC.

The need for a new JMM has stimulated wide research interests in software memory models. Some work focuses on understanding the old JMM, and some has been done to formalize the old JMM and provide an operational specification [13]. There are also some work giving new semantics for multithreaded Java [10] and some of them have been accepted as candidates of the new JMM revisions. Yang et al. [24] used an executable framework called Uniform Memory Model (UMM) for specifying a new JMM developed by Manson and Pugh [17].

The implementation and performance impact of the JMM on multiprocessor platforms is an important and new topic, which can be a guide for implementing the new JMM (as currently specified by JSR-133). In the cookbook [6], Douglas Lea describes how to implement the new JMM, including re-orderings, memory barriers and atomic operations. It briefly depicts the backgrounds of those required rules and concentrates on their consequences for compilers and JVMs. It includes a set of recommended recipes for complying to JSR-133. However, he didn’t provide any implementation and performance evaluation in this work.
Chapter 3

Relationship between Memory Models

The aim of this work is to study the performance impact of the JMM from out-of-order multiprocessor perspective. Therefore the JMM and hardware memory model should be investigated jointly. We will evaluate the performance of the old JMM and the new JMM on multiprocessor with sequential consistency (SC) and with some relaxed consistency models such as TSO, PSO, WO and RC.

3.1 How JMM Affect Performance

First, let us see how multithreaded Java programs are implemented. The source programs are compiled into bytecodes, and then the bytecodes are converted into hardware instructions by the JVM, and at last the hardware instructions are executed by the processor. This process is illustrated in Figure 3.1. Some optimizations may be introduced in this process. For example, the compiler may reorder...
the bytecode to make it shorter and more efficient. However, the JMM should be respected in the whole process. We need to ensure the following: (a) the compiler does not violate the JMM while optimizing Java bytecodes, and (b) the JVM implementation does not violate the JMM. In addition, the execution on processors also needs to be considered under different situations. For uniprocessor, the supported model of execution is Sequential Consistency [1]. The SC model is the strictest memory model and is more restrictive than all the JMMs. Therefore the uniprocessor platform and multiprocessor platform with SC memory model never violate the JMM. But if the multiprocessor is not sequential consistent, then some measures should be adopted on either the compiler or JVM to make sure that the JMM is not violated. In this project, we focus on the performance impact of different JMMs from out-of-order multiprocessor perspective and do not consider
uniprocessor.

Memory barrier instruction is introduced here to guarantee that the JMM is respected. If a memory barrier I appears between instructions $I_1$ and $I_2$, instruction $I_1$ must complete before $I_2$ begins. We can insert memory barrier instructions on compiler or JVM to disable some re-orderings allowed by the hardware memory model but not allowed by the JMM. However, a memory barrier is a time-expensive hardware instruction. We should put as few memory barriers as possible to reduce the overheads. Therefore, it is important for us to clarify the relationship between the JMM and the underlying hardware memory model.

Conceptually, the JMM and hardware memory models are quite similar, and they both describe a set of rules dictating the allowed reordering of read/write of shared variables in a memory system. Figure 3.2 shows a multiprocessor implementation of Java multithreading. Both the compiler re-orderings as well as the re-orderings introduced by the hardware memory consistency model need to respect the JMM. In other words, they both consist of a collection of behaviors that can be seen by programmers. So if a hardware memory model has more allowed behaviors than the JMM, it is possible that the hardware memory model may violate the JMM. On the other hand, if the hardware memory model is more restrictive, then it is impossible for the hardware memory model to violate the JMM. Because SC is more restrictive than both the old JMM and the new JMM, SC has fewer allowed behaviors than both the JMMs. Thus SC hardware memory model can guarantee that the JMMs are never violated. However, if the relaxed hardware memory models are used, this is not guaranteed. This is because some relaxed memory model
may allow some behaviors which are not allowed by the JMMs. In this case, we must ensure that the used hardware consistency model does not violate the JMMs. Let us explain this using an example.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>write b, 0</td>
<td>lock n</td>
</tr>
<tr>
<td>write a, 0</td>
<td>read b</td>
</tr>
<tr>
<td>lock m</td>
<td>unlock n</td>
</tr>
<tr>
<td>write a, 1</td>
<td>lock n</td>
</tr>
<tr>
<td>unlock m</td>
<td>read a</td>
</tr>
<tr>
<td>write b, 1</td>
<td>unlock n</td>
</tr>
</tbody>
</table>

Note that in Thread 2, we use ”lock n” and ”unlock n” to ensure that ”read a” is
executed only after "read b" has completed. If we use RC as hardware consistency model and do not take the JMM into account, it is possible to read b = 1 and a = 0 in the second thread. That is because for the first thread, RC allows "write b, 1" to bypass "unlock m" and "write a, 1" But the old JMM does not allow this result to happen because it requires that "write b, 1" can only be issued after "unlock m" is completed. In this case, the hardware consistency model is "weaker" than the JMM; so barrier instructions must been inserted to make sure that the JMM is not violated. Naturally, this instruction insertion will add overhead in the execution of the program.

The problem caused by this has been indicated by Pugh: an inappropriate choice of JMM can disable common compiler re-orderings [11]. In this project, we study how the choice of JMM can influence the performance under different hardware memory models. Note that if the hardware memory is more relaxed (i.e., allows more behaviors) than the JMM, 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 and reduce the efficiency of the system. This explains the performance impact brought by the different JMMs on multiprocessors.

### 3.2 How to Evaluate the Performance

To evaluate the performance of JMM under various hardware memory models, we need to implement the old JMM and new JMM as well as multiprocessor platform. For JMM, it can be achieved by inserting memory barriers through programming.
While it is expensive to get a real multiprocessor platform with various hardware memory models and also it is not very suitable for our experiment since we need to get various statistic data. Therefore, we tend to use a multiprocessor simulator. Now there are lots of multiprocessor simulators from event-driven level to system level. Using simulator has several advantages. First, it is much easier to get a simulator than a real one. Although the price of computer has dropped dramatically, multiprocessor computers are still much more expensive than uniprocessor ones because of their complex architecture and special use. Second, simulators can be freely configured to get different platforms. We need to use five different hardware memory models so we need to choose an appropriate simulator to achieve this. Moreover, it provides lots of API functions and it is possible for us to change the configuration and get the required measures for the evaluation of performance under different situations. In this experiment, we use a system-level simulator, Simics, to simulate a four-processor platform. The details about this simulator will be discussed in Chapter 5.

Next, we need to consider Java Memory Models. JMMs must be based on the above hardware memory models to get the evaluation of performance. First, we need to compare a JMM with a relaxed hardware memory model and check whether the relaxed hardware memory model allows more re-orderings. If more re-orderings are allowed, then memory barrier instructions need to be explicitly inserted to ensure the JMM isn’t violated. 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 which are allowed
by the various hardware consistency models are disallowed by $M$ and $M'$. In this work, we choose the old JMM and the new JMM as the objects of our study. The issue of inserting barriers to implement these two JMMs on different hardware memory models is discussed in the next chapter.
Chapter 4

Memory Barrier Insertion

As described in previous chapter, when multithreaded Java programs run on multiprocessor platforms with a relaxed memory model, we need to insert memory barrier instructions through JVM to ensure that the JMM is not violated. Two JMMs are considered here: (a) the old JMM (the current JMM) described in the Java Language Specification (henceforth called $JMM_{old}$), and (b) the new JMM proposed to revise the current JMM (henceforth called $JMM_{new}$). These two JMMs are different in many places, but we do not compare them point by point. Instead the purpose of the study is to compare the overall performance difference.

In addition, we run the programs on multiprocessor platform without any software memory model. Thus we can find the performance bottlenecks brought by the JMM, and identify the performance impact of different features in the new JMM.

Since the old JMM specification given in the JLS is abstract and rule-based, we refer to the operation style formal specification developed in [13]. For $JMM_{new}$, Doug Lea describes instruction re-orderings, multiprocessor barrier instructions,
and atomic operations in his cookbook [6]. Some other research papers also give the allowed reordering among operations in $JMM_{new}$ [18].

Besides using different JMMs, we choose different hardware memory models to compare the JMMs against various relaxed multiprocessor platforms. The following hardware memory models are selected (listed in order of relaxedness): Sequential Consistency (SC), Total Store Order (TSO), Partial Store Order (PSO), Weak Order (WO), and Release Consistency (RC). We need to compare the relaxed memory models with the JMMs one by one, and consider the re-orderings allowed by these models among various types of operations. These operations include shared variable read/write, lock/unlock, volatile variable read/write and final fields (only for the $JMM_{new}$). If the underlying hardware memory model allows more behaviors than the JMM, memory barrier instructions are inserted through JVM to guarantee the JMM is not violated.

Memory barriers are inserted at different places. For clarity, we employ the following notations to organize memory barriers into groups. 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 $Wr^\uparrow$, write operations must be completed before $x$ starts. $Rd^\uparrow$ and $Wr^\uparrow$ can be combined to $RW^\uparrow$, which requires both read and write operations to complete. On the other hand, if a requirement of $Rd_\downarrow$ is associated with operation $x$, the 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 $RW_\downarrow \equiv Rd_\downarrow \wedge Wr_\downarrow$. 

30
4.1 Barriers for normal reads/writes

In both $JMM_{old}$ and $JMM_{new}$, there are no restrictions among shared variable operations. Therefore, reads/writes to shared variables can be arbitrarily reordered with other shared variable reads/writes within a thread if the accesses are not otherwise dependent with respect to basic Java semantics (as specified in the JLS). For example, we cannot reorder a read with a subsequent write to the same location, but we can reorder a read and a write to two distinct locations. Consequently, if a pair of operations is allowed to be reordered, they can be completed out-of-order thereby achieving the effect of bypassing.

Obviously, the allowed behaviors among shared variable reads/writes are more than those allowed by any hardware memory models. Therefore no memory barriers need to be inserted between shared variable read/write instructions on multiprocessor platforms.

For shared variable reads/writes, the situation is quite simple in the absence of lock/unlock and volatile variables. In fact, for multithreaded Java programs, lock/unlock and volatile variables have special purposes. So the JMM gives the semantics of these operations and enforce access restrictions of them. The Table 4.1 shows the main rules of $JMM_{new}$ for lock/unlock and volatile reads/writes [6]. The cells with "No" indicate that you cannot reorder instructions with particular sequences of operations. The cells for Shared Variable Reads are the same as for Shared Variable Writes, those for Volatile Reads are the same as Lock, and those for Volatile Writes are same as Unlock, so they are collapsed together here. From the table, we can see there is no restriction between shared variable reads/writes.
<table>
<thead>
<tr>
<th>Can Reorder</th>
<th>2nd operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st operation</td>
<td>Normal Read</td>
</tr>
<tr>
<td>Normal Read</td>
<td>Normal Write</td>
</tr>
<tr>
<td>Normal Write</td>
<td></td>
</tr>
<tr>
<td>Volatile Read</td>
<td>No</td>
</tr>
<tr>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td>Volatile Write</td>
<td>No</td>
</tr>
<tr>
<td>Unlock</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Re-orderings between memory operations for $JMM_{new}$

Other cells are explained in the following sections.

### 4.2 Barriers for Lock and Unlock

Lock and unlock are synchronization operations which are different from normal read and write operations. Thus we need to consider them specially. A lock is essentially an atomic read-and-write operation. Only when the lock is completed successfully, the following operations can execute. So any instruction after a lock possesses a control dependency on the lock, and hence can’t bypass the lock. Thus we need not insert any memory barriers after a lock operation. This is applicable to any hardware and software memory models, and a lock operation is never associated with $Rd_1$ or $Wr_1$ or $RW_1$. However, we need consider inserting memory barriers before a lock since operations before a lock can be completed after the lock under
The unlock operation is not the same as the lock. It is an atomic write operation to shared memory address. There is no control dependency on it. Thus, operations after unlock may bypass unlock and operations before unlock may be bypassed by unlock. Therefore, we need insert memory barriers before and after unlock.

First, let us consider the memory barriers to be inserted under these various hardware memory models to satisfy $JMM_{old}$. $JMM_{old}$ is originally described in [4]. But it is abstract and rule-based, which is not suitable for formal verification. In [13], an equivalent formal executable specification of $JMM_{old}$ is developed, which is used by us to obtain the memory barriers required under $JMM_{old}$. The results are summarized in Table 4.2. To explain how the results are derived, let us see how the actions of lock and unlock are described in [13], illustrated in Figure 4.1. lock, means a lock operation in thread $i$. $j$ refers to shared variables in the program and there are totally $m$ shared variables. $rd_{qt,j}$ is a read queue and contains values of the variable $v_j$ as obtained (from master copy) by read actions in thread $i$, but for which the corresponding load actions (to update the local copy) are not yet to be performed. Similarly, queue $wr_{qt,j}$ contains values of the variable $v_j$ as obtained (from local copy) by store actions in thread $i$, but for which the corresponding

<table>
<thead>
<tr>
<th>Operation</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>No</td>
<td>No</td>
<td>$W_r\uparrow$</td>
<td>No</td>
<td>$R_d\uparrow$</td>
</tr>
<tr>
<td>Unlock</td>
<td>No</td>
<td>No</td>
<td>$W_r\uparrow \land W_r\downarrow$</td>
<td>No</td>
<td>$W_r\downarrow$</td>
</tr>
</tbody>
</table>

Table 4.2: Memory Barriers Required for Lock and Unlock Satisfying $JMM_{old}$
Action  \textbf{lock}: \\
\((\forall k \neq i \text{lock}_{\text{cnt}}_k = 0) \land \forall 1 \leq j \leq m \ (\text{empty}(\text{rd}_{q_{i,j}}) \land \neg \text{dirty}_{i,j}) \rightarrow \text{lock}_{\text{cnt}}_i := \text{lock}_{\text{cnt}}_i + 1; \text{ for } j := 1 \text{ to } m \text{ do } \text{stale}_{i,j} := \text{true} \)

Action  \textbf{unlock}: \\
\text{lock}_{\text{cnt}}_i > 0 \land \forall 1 \leq j \leq m \ (\text{empty}(\text{wr}_{q_{i,j}}) \land \neg \text{dirty}_{i,j}) \rightarrow \text{lock}_{\text{cnt}}_i := \text{lock}_{\text{cnt}}_i - 1

Figure 4.1: Actions of  \textbf{lock} and  \textbf{unlock} in  \textit{JMM}_{\text{old}}

\textbf{write} actions (to update the local master copy) are yet to be performed. Here let’s discuss  \textit{empty}(\text{rd}_{q_{i,j}}) and  \textit{empty}(\text{rd}_{q_{i,j}}) in action  \textbf{lock} and  \textbf{unlock} first.  \textbf{empty} here means to finish all the memory operations in corresponding queues. In action  \textbf{lock}, queue  \text{rd}_{q_{i,j}} needs to be emptied before the  \textbf{lock} can be performed. Similarly, in action  \textbf{unlock}, queue  \text{wr}_{q_{i,j}} needs to be emptied before the  \textbf{unlock} can be performed. The component  \text{dirty}_{i,j} is a bit indicating whether the local copy of  \textit{v}_j is dirty, that is, there is an assignment to  \textit{v}_j by thread  \textit{i} which is not yet visible to other threads. Here we require no variables can be dirty, which means all assignments are visible to other threads. Therefore, in  \textit{JMM}_{\text{old}} we only need to consider read operations for  \textbf{lock} and write operations for  \textbf{unlock}.

We now consider a particular hardware memory model, say PSO. PSO allows reads and writes to bypass previous writes, and no other bypassing is allowed. Note that lock is an atomic read-and-write operation and only memory barriers insertion before a lock are needed. In addition, only read operations are required to be considered here. Since read operations are not allowed to be bypassed by other memory operations in PSO, no memory barriers need to be inserted before a lock in this situation. However, lock can’t be reordered with other lock/unlock operations, so a  \textit{Wr}^\top is required to ensure this. For unlock, it is an atomic write
operation and only write operations are required to be considered here. Therefore, in PSO write operations before an unlock can be bypassed by the unlock. Similarly, write operations after an unlock can bypass the unlock. These violate the program order restrictions in $JMM_{old}$ [13]. Thus $W_{r\uparrow}$ and $W_{r\downarrow}$ need to be inserted before and after the unlock respectively to ensure the $JMM_{old}$ is not violated.

Now let us consider the memory barriers which need to be inserted before lock and before/after unlock under various relaxed models so that $JMM_{new}$ is satisfied. Table 4.1 presents the program order restrictions imposed by $JMM_{new}$. Here we are only concerned with lock/unlock and normal read/write. From the table, we can see that lock can be reordered with respect to previous normal read/write but not with following normal read/write. While unlock can be reordered with respect to the following normal read/write but not with previous normal read/write. Since any operation after a lock can not bypass the lock, no memory barriers are required after a lock. But for PSO, write can bypass previous write so a $W_{r\uparrow}$ memory barrier needs to be associated with lock. For unlock, we only need to insert memory barriers before the unlock to prevent it bypassing previous normal read/write. For TSO, only read can bypass previous write so no memory barriers are required for unlock. But for PSO, write can also bypass previous write so a $W_{r\downarrow}$ memory barrier needs

<table>
<thead>
<tr>
<th>Operation</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>No</td>
<td>No</td>
<td>$W_{r\uparrow}$</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Unlock</td>
<td>No</td>
<td>No</td>
<td>$W_{r\uparrow}$</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.3: Memory Barriers Required for Lock and Unlock Satisfying $JMM_{new}$
to be associated with unlock. For WO and RC, unlock can be regarded as guarded actions. For WO, read/write can not bypass or be bypassed by unlock. For RC, unlock can be bypassed by following read/write, which is in accordance with the requirement of $JMM_{new}$. Therefore, no memory barriers are required for both WO and RC. Thus we can summarize the results in Table 4.3.

### 4.3 Barriers for volatile reads/writes

If a variable is defined as volatile, then operations to the variable will directly access the main memory. For $JMM_{old}$, reads/writes of volatile variables are not allowed to be reordered among themselves. But they may be reordered with respect to normal variables. For example, in the following pseudo code

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read volatile v</td>
<td>write u, 1</td>
</tr>
<tr>
<td>read u</td>
<td>write u, 2</td>
</tr>
<tr>
<td></td>
<td>write volatile v, 1</td>
</tr>
</tbody>
</table>

it is possible to read $v==1$ and $u==1$ in the first thread. Actually, this is a weakness of the volatile variable semantics [25]. To comply with the $JMM_{old}$, memory barriers need to be inserted before volatile reads/writes, the scheme of which is described in Table 4.4.

To explain how we obtain the results, consider a particular hardware memory model, say PSO. PSO allows reads and writes to bypass previous writes, and no other bypassing is allowed. $JMM_{old}$ does not allow volatile reads/writes to reorder with respect to other volatile reads/writes. However, from the hardware level, we
Table 4.4: Memory Barriers Required for Volatile Variable Satisfying $JMM_{old}$

can not distinguish volatile reads/writes from normal reads/writes. So we first put a memory barrier $W_r\uparrow$ before a volatile read to prevent the volatile read from reordering with previous writes (both normal and volatile). No other memory barriers are required for the volatile read because no other reordering is allowed by the hardware memory model. While for the volatile write, a $W_r\uparrow$ is needed to prevent the volatile write from reordering with previous writes. Moreover, the following reads/writes may also reorder with this volatile write from the hardware view. But no memory barriers are needed because if the following reads/writes are volatile, there are $W_r\uparrow$ before them and reordering is avoided, and for the normal reads/writes, the reordering is allowed by $JMM_{old}$. The requirements of memory barriers for other hardware memory models are derived in the same way.

In $JMM_{new}$, the restrictions among volatile variables and with normal variables are described in Table 4.1. The allowed reorderings are similar to those of $JMM_{old}$. But we need to be aware of two points that are different from $JMM_{old}$. As shown in Table 4.1, the cell between volatile read and normal read/write and the one between normal read/write and volatile write are filled with "No". Thus volatile read can not be reordered with following normal reads/writes and volatile write can not be reordered with the previous normal reads/writes. The scheme of memory barriers
Table 4.5: Memory Barriers Required for Volatile Variable Satisfying $JMM_{new}$

<table>
<thead>
<tr>
<th>Operation</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volatile Read</td>
<td>No</td>
<td>$W_r^\uparrow$</td>
<td>$W_r^\uparrow$</td>
<td>$RW^\uparrow \land RW^\downarrow$</td>
<td>$RW^\uparrow \land RW^\downarrow$</td>
</tr>
<tr>
<td>Volatile Write</td>
<td>No</td>
<td>No</td>
<td>$W_r^\uparrow$</td>
<td>$RW^\uparrow$</td>
<td>$RW^\uparrow$</td>
</tr>
</tbody>
</table>

is indicated in Table 4.5. The results are obtained the same way as $JMM_{old}$ except that two re-orderings described above are not allowed in $JMM_{new}$. The memory barrier $RW^\downarrow$ for WO and RC shows the difference from Table 4.4.

Since $JMM_{new}$ imposes more constraints than $JMM_{old}$, a few more memory barriers are required to obey $JMM_{new}$ for some hardware memory models. This leads to some performance difference across the two software memory models in benchmarks involving large number of volatile variable accesses.

### 4.4 Barriers for final fields

Final fields in Java programs are initialized once and never changed, and should be treated specially. In $JMM_{old}$ there are no special semantics for final fields. However, $JMM_{new}$ provides special treatment for final fields as described in Chapter 2. The semantics requires that the final fields must be used correctly to provide a guarantee of immutability. This can be achieved by ensuring all writes in a constructor to be visible when final fields are initialized. Final fields are generally set in the constructor, so the effect can be obtained by inserting a barrier at the end of the constructor. Thus, a memory barrier $W_r^\uparrow$ is required before the constructor finishes.
Chapter 5

Experimental Setup

It is very difficult to compare the performance impact of the old JMM and the new JMM on real multiprocessor platforms because the results are greatly influenced by the system and it is impossible to reproduce identical situations at different time and the statistics are hard to collect. Therefore we decided to use multiprocessor simulators. There are many kinds of simulators from instruction level to system level. Since we want to study the effect of the old JMM and the new JMM in disabling the re-orderings allowed by different hardware memory models from commercial multiprocessor perspective, it is better for us to use a system-level simulator that can simulate a complete multiprocessor platform. Thus we choose the Simics system-level, functional simulator to simulate a multiprocessor target system [20]. Simics is a system-level architectural simulator developed by Virtutech AB and supports various processors like SPARC, Alpha, x86 etc. It is a platform for full system simulation that can run actual firmware and completely unmodified kernel and driver code. Furthermore, it provides a set of application programming
interfaces (API) that allow users to write new components, add new commands, or write control and analysis routines [20].

In our experiment, the processor is simulated as Sun Microsystems’ SPARC V9 architecture. The target platform is a four-processor shared memory system running Linux. In order to obtain the 5 different hardware memory models, we need to use the feature of the Simics out-of-order processor model. In this model, multiple instructions can be active at the same time, and several instructions can commit in the same cycle. And memory operations can be executed out of order. There is also a consistency controller to enforce the architecturally defined consistency model [21]. Thus we can simulate multiprocessor with different hardware memory models by configuring the consistency controller.

Upon the simulated platform, we use Kaffe as the Java Virtual Machine (JVM) because Kaffe is an open source JVM and has been ported to various platforms. It is possible for us to change the source codes to implement the hardware and software memory models. In addition, we need Java threads to be scheduled to different processors. This requires special thread library from the operating system and support from the JVM. Kaffe has an option to choose thread library and can make use of Pthreads library supported in Linux.

The benchmark used in our experiment is Java Grande benchmark suit. This benchmark suit has a multithreaded version, which is designed for parallel execution on shared memory multiprocessors. We choose five benchmarks of different types from the multithreaded benchmark suit, which have different number of volatile variables and locks/unlocks. We can see how those different types of variables
affect the performance under the two JMM specifications.

5.1 Simulator

We use Simics to simulate our shared memory multiprocessor platform. Simics is an efficient, instrumented, system level instruction set simulator, allowing simulation of multiprocessor. It supports a wide range of target systems as well as host systems and provides a lot of freedom for customization. We can easily specify the number of simulated processors and add other modules (e.g., caches and memory) to the system.

5.1.1 Processor

We simulate a shared memory multiprocessor (SMP) consisting of four SUN Ultra-SPARC II and MESI cache coherence protocol. The processors are configured as 4-way superscalar out-of-order execution engines. We use separate 256KB instruction and data caches: each is 2-way set associative with 32-byte line size.

The simulated processor is UltraSPARC II running in out-of-order mode. Simics provides two basic execution modes: an in-order execution mode and an out-of-order mode. The in-order execution mode is the default mode and quite simple. In this mode, instructions are scheduled sequentially in program order. In other words, other instructions can not execute until a previous instruction has completed, even if it takes many simulated cycles to execute. For example, a memory read operation that misses in a cache stalls the issuing CPU. The out-of-order execution mode has the feature of a modern pipelined out-of-order processor. This mode can produce
multiple outstanding memory requests that do not necessarily occur in program order. This means that the order of issuing instructions is not the same as the order of completing instructions. In Simics, this is achieved by breaking instructions into several phases that can be scheduled independently. Clearly, we must use out-of-order execution mode in our experiments so that we can simulate multiprocessor platform with different hardware memory models.

The Simics out-of-order processor model can be run in two different modes, Parameterized mode and Fully Specified mode, depending on what the user wants to model and in what level of detail. The parameterized mode is intended to simulate a system where having an out-of-order processor is important, but the exact details of micro-architecture are not important. This mode provides three parameters for user to specify the number of transactions that can be outstanding: the number of instructions that can be fetched in every cycle, the number of instructions that can be committed in every cycle and the size of the out-of-order window. In the full specified mode, the user has full control over the timing in the processor through the Micro Architecture Interface (MAI). In our experiment, we only need out-of-order processors but not the details of how to schedule instructions. Therefore parameterized mode can meet our requirements.

5.1.2 Consistency Controller

Every out-of-order processor must have a consistency controller that needs to be connected between Simics processors and the first memory hierarchy. The consistency controller is a memory module to ensure that the architecturally defined
consistency model is not violated. The consistency controller can be constrained through the following attributes (setting an attribute to 0 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 Sequential Consistency (SC). For TSO writes can be reordered with following reads. To obtain this hardware memory model, 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. For WO and RC, it is not sufficient to just set the four attributes to zero. We need further to identify the synchronization operations. However, it is not easy to achieve this in Simics because there are no corresponding instructions from the hardware level. But in the Java bytecode instruction set, there are two specific opcodes for synchronization, MONITORENTER and MONITOREXIT for lock and unlock respectively. Thus it is much easier to identify the synchronization operations in JVM, which will be described in the following section.
However, there is another problem in the implementation. Indeed, the Simics Consistency Controller does not support PSO in the default mode. So we need modify the default Consistency Controller to implement PSO. 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 can’t bypass any previous instruction in the original implementation. We allowed the store to go ahead even if there are uncommitted earlier instructions. We did not face any problems due to the removal of this restriction of Simics. That is, we hardly ever faced a situation where an uncommitted earlier instruction raised exception; if such a situation did happen, we aborted that simulation run and restarted simulation. The PSO has been verified in our implementation.

5.1.3 Cache

Cache is the first memory hierarchy in our system. In our simulator, we only use one level cache, so we employ the generic-cache module provided by Simics. This cache is an example memory hierarchy modeling a shared data and instruction cache. It supports a simple MESI snooping protocol if an SMP system is modeled. It can also be extended to multi-level caches using the next_cache attribute if necessary. The cache size, number of lines and associativity etc. can be specified by setting the provided attributes.
5.1.4 Main Memory

The main memory in Simics is also implemented as a module. In our simulator, the memory is shared among all the processors. So the memory module must be connected to all the processors and caches. The entire memory hierarchy in our simulator is displayed in Figure 5.1. All the processors are connected with consistency controllers first and the consistency controllers must be associated with the first memory hierarchy. Here the first memory hierarchy is cache and the consistency controllers are connected to their corresponding caches. Finally, all the caches are attached to the shared main memory.
5.1.5 Operating System

Since Simics is a full-system simulator, an operating system is required for our simulated system. Our target system is a Symmetric Multi-Processor (SMP) system with four processors. Thus we need an operating system supporting SMP. A Linux SMP kernel is definitely the best choice as Linux supports both SPARC and SMP and there is no license problem for it. In fact, in Simics it is not necessary to install an operating system from scratch because Simics can boot up from a disk image with the required operating system. Fortunately, Simics provides a sparc-linux kernel-2.4.14 disk image that supports SMP, which greatly speeds up our process.

5.1.6 Configuration and Checkpoint

The detail configurations of the target systems are described in a file written in a special configuration language. The file consists of a collection of modules and their attributes. Modules can be connected to other modules by setting attributes in the configuration file. The simulated machine boots up according to the content of the configuration file.

The simulated system can be interrupted and saved at any time while running. The saved checkpoint includes all simulated state and the current configuration. On the other hand, the saved checkpoint can be loaded in Simics and the simulated system can be recuperated to the state where it was saved. Thus we can obtain identical states any times we want. This is very crucial for our experiments.


5.2 Java Virtual Machine

The choice of JVM is very important to our experiment because we need to change the JVM to implement different memory models and other requirements. Kaffe is a free virtual machine that runs Java code and supports a wide range of platforms. Furthermore, there are several choices for the implementation of thread in Kaffe. Altogether, Kaffe can satisfy the needs in our experiment and is an excellent JVM for us.

As described above, to implement the WO and RC hardware memory models, we need to identify synchronization operations. It is much easier to achieve this in JVM because there are special opcodes for synchronization in Java bytecode instruction set. Opcode MONITORENTER and MONITOREXIT correspond to lock and unlock operation. For WO, memory operations before a lock can’t be reordered with the lock operation. And memory operations after a lock can’t be reordered with the lock operation either. Similarly, unlock operation can’t be reordered with previous operations and following operations. Thus we need to put memory barriers before and after lock/unlock. Since a lock is essentially an atomic read-and-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, it is similar except that we only need to insert one memory barrier after a lock and one before an unlock. Due to the
same reason, memory barrier after a lock isn’t necessary. Thus in Kaffe, only one memory barrier is inserted before the implementation of MONITOREXIT for RC.

Since our simulated platform is a four-processor SMP Linux, some measures must be taken for multithreaded programs to make best use of multiprocessors. Generally there are three parallelization methods: POSIX Threads (Pthreads), Message Passing Libraries and Multiple Processes. Since both Message Passing Libraries and Multiple Processes usually do not share memory and communicate either by means of Inter-Process Communications (IPC) or a messaging API, they are not specific to SMP. Only Pthreads provide us with multiple threads sharing memory. Kaffe provides several methods for the implementation of multiple Java threads including kernel-level and application-level threads. However, threads of application-level do not take advantage of the kernel threading and the thread packages keep the treading in a single process, hence do not take advantage of SMP. Consequently, to use multiprocessors of an SMP, we must use a kernel Pthreads library.

5.3 Java Native Interface

The software memory models are implemented in the Java source level and the memory barriers for different memory models are inserted in the Java programs. The reason is because there is no such semantics like volatile variables and final fields in the hardware level, and it is comparatively harder to achieve this in the JVM. In the Java source codes, it is very easy to identify volatile variables, synchronization methods and final fields. The difficulty is not where to insert mem-
ory barriers but how to do so. SPARC processors have specific memory barriers to prevent CPU from reordering memory accesses across the barrier instructions. However, it is not possible to insert such memory barrier instructions directly in Java programs because Java is independent of hardware platforms. GCC permits programmers to add architecture-dependent assembly instructions to C and C++ programs. Thus it is possible for us to write a C program containing memory barrier instructions.

To use C programs in Java, we need to make use of the Java Native Interface (JNI). The JNI allows Java code that runs within a JVM to operate with applications and libraries written in other languages, such as C, C++, and assembly. Writing native methods for Java programs is a multi-step process.

1. Write the Java program that declares the native method.

2. Compile the Java program into class that contains the declaration for the native method.

3. Generate a header file for the native method using javah provided by the JVM (in Kaffe it is kaffeh).

4. Write the implementation of the native method in the desired language. Here is C with inline assembly.

5. Compile the header and implementation files into a shared library file.

After these five steps, the native method written in the Java program can be invoked in any Java programs. Then the memory barrier instructions can be inserted into any place in a Java program.
5.4 Benchmarks

The benchmark used by us is from Java Grande Forum Benchmark Suite, which is a suite of benchmarks to measure different execution environments of Java against each other and native code implementations. The five multithreaded benchmarks selected from the benchmark suite are: Sync, LU, SOR, Series and Ray. Those benchmarks are selected from different categories and their sizes are reduced to fit our system. These benchmarks are designed to test the performance of real multithreaded applications running under a Java environment. The performance is measured by running the benchmark for a specific time and recording the number of operations executed in that time. Sync measures the performance of synchronized methods and synchronized blocks. LU, SOR and Series are medium-sized kernels. In particular, LU solves a 40x40 linear system using LU factorization followed by a triangular solve. The factorization is computed using multithreads in parallel while the remainder is computed in serial. It is a Java version of the well known Linpack benchmark. SOR performs 100 iterations of successive over-relaxation on a 50 × 50 grid. This benchmark is inherently serial and the algorithm has been modified to allow parallelization. The Series benchmark computes the first 30 Fourier coefficients of the function \( f(x) = (x + 1)^x \) on the interval 0...2. This benchmark heavily exercises transcendental and trigonometric functions. Ray is a large application benchmark and measures the performance of 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 and writes, accounting for 2-15% of all the operations.
5.5 Validation

In our experimental setup, we made some changes to the simulator, the Java Virtual Machine (JVM) and the benchmarks. These changes may invalidate our simulated system. So we need to make sure that our simulation model is implemented correctly and it is an accurate representation of the real system.

First, let’s see the change to the simulator. We removed the restriction that a store instruction cannot bypass any previous instruction. This restriction guarantees that no exception happens in the simulator. However, it is so strict that we can’t implement PSO with it. So we had to allow the store to go ahead even if there are uncommitted earlier instructions. In our experiment, we did not face any problems due to the removal of this restriction.

Second, the changes to the JVM and the benchmarks are both for insertions of memory barriers. Memory barrier insertions are guided by the hardware and software memory models as described in Chapter 4. These memory barriers only restrict the execution orders of memory operations. Under the guidance of memory models, the program can still produce correct results.

Moreover, we run a unmodified Linux SMP kernel and successfully built the Kaffe JVM on it. Therefore, we can ensure the simulator runs correctly. And the execution results and memory traffic were also analyzed to validate the correctness. For all the benchmarks, they print some information about the execution after every run. The information can be used to make sure the runs are correct. In a lower level, we can trace memory traffic in the simulator. The execution can be broken at any time and we can check the memory operations. From the analysis, we found
the runs are all correctly produced.

Validation is very important to our experiment. We tried our best to validate our simulation model. In order to guarantee the validation, we examined all the places that may cause problems. And we also analyzed the memory traffic and execution outcomes in different levels.
Chapter 6

Experimental Results

This chapter presents the results of our experiments. From the results, we can compare the performance impact of the old JMM and the new JMM on out-of-order multiprocessor platform with different hardware memory models. All the five multithreaded Java Grande benchmarks are adapted to observe the specifications provided in the old JMM and the new JMM respectively. Then they are executed on the simulated system that is configured as Sequential Consistency (SC), Total Store Order (TSO), Partial Store Order (PSO), Weak Ordering (WO) and Release Consistency (RC) hardware memory models. The performance is measured by the number of cycles needed for a benchmark under certain software and hardware memory model. Since every benchmark has different numbers of volatile variables, synchronization operations and final fields, we can analyze their influence to the entire performance. Those numbers also affect the number of memory barriers inserted in the benchmarks. We first present the numbers of required memory barriers for both the old and the new JMM under those relaxed hardware memory
### Table 5: Characteristics of benchmarks used

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Volatile Read</th>
<th>Volatile Write</th>
<th>Constructors with Final Field Writes</th>
<th>Lock</th>
<th>Unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>8300</td>
<td>936</td>
<td>52</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SOR</td>
<td>51604</td>
<td>480</td>
<td>20</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SERIES</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SYNC</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>RAYTRACER</td>
<td>48</td>
<td>20</td>
<td>768</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 6.1: Characteristics of benchmarks used

models.

## 6.1 Memory Barriers

The number of memory barriers is a crucial factor to our experiment. It greatly influences the entire performance of the benchmarks and reflects the requirements of the software and hardware memory models. The memory barriers are due to volatile variable accesses, synchronization operations and final fields. Table 6.1 shows the number of volatile reads/writes, synchronization operations and final field writes for our benchmarks. 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. In order to observe the effect of the synchronization operations to the performance, we also choose two benchmarks without any volatile read/write. Those memory barriers are inserted in the places according to the schemes described in Chapter 4 and they ensure that Java programs comply with $J\text{MM}_{old}$ and $J\text{MM}_{new}$. Memory barriers affect the performance significantly
because the overheads are not just the cycles executing memory barrier instructions but also include waiting cycles to finish other operations. The waiting cycles account for the major overheads as there may be many memory operations pending to be completed.

Table 6.2 shows the number of memory barriers for $JMM_{old}$ and $JMM_{new}$ under relaxed hardware memory models. Since those memory barriers are introduced by different reasons, we also include separate numbers in every situation. In some cases, the numbers of memory barriers are the same, but they are from different source, and thus the total cycles required will not be equal.

Since SC is stricter than both of the JMMs, no memory barrier is required for this hardware memory model. From the table we can see that LU, SOR and Ray need much more memory barriers than Series and Sync. This is because LU, SOR and Ray all have a large number of volatile reads and writes, and for both $JMM_{old}$ and $JMM_{new}$ memory barriers are required under relaxed hardware memory models. Since $JMM_{new}$ imposes more restrictions on volatile variables, generally $JMM_{new}$ needs more barriers than $JMM_{old}$ for these three benchmarks under certain hardware memory model. Moreover, for $JMM_{old}$ with these three benchmarks, we can observe that the hardware memory models PSO, WO and RC need more barriers than TSO. The reason is that TSO need 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 $JMM_{old}$. Similarly, for $JMM_{new}$ with these three benchmarks, PSO introduces more memory barrier than TSO, and WO and RC introduce more than PSO. This
<table>
<thead>
<tr>
<th></th>
<th>SOR</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>2004</td>
<td>2812</td>
<td>2808</td>
<td>2812</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>2004</td>
<td>2820</td>
<td>2808</td>
<td>2820</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>2004</td>
<td>2810</td>
<td>4813</td>
<td>4813</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>2006</td>
<td>2816</td>
<td>4815</td>
<td>4815</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LU</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>3979</td>
<td>4924</td>
<td>4920</td>
<td>4824</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>3979</td>
<td>4932</td>
<td>4920</td>
<td>4832</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>3979</td>
<td>4922</td>
<td>6124</td>
<td>6124</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>3985</td>
<td>4932</td>
<td>6130</td>
<td>6130</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Series</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Sync</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ray</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>35</td>
<td>84</td>
<td>49</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>35</td>
<td>100</td>
<td>49</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NEW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>volatile read/write</td>
<td>35</td>
<td>92</td>
<td>73</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>lock/unlock</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>final field write</td>
<td>863</td>
<td>863</td>
<td>863</td>
<td>863</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>898</td>
<td>963</td>
<td>936</td>
<td>963</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Number of Memory Barriers inserted in different memory models
is because TSO needs memory barriers before volatile read operations, and PSO need barriers before both volatile read and write operations, and WO and RC need barriers before volatile read and write operations and after volatile read operations.

The other two benchmarks Series and Sync have no volatile variables (showed in Table 6.1). The memory barriers are due to synchronization operations and final fields. Since Series and Sync both do not have many synchronization operations and final fields, the necessary memory barriers are very few. For synchronization operations, \( JMM_{old} \) need more memory barriers than \( JMM_{new} \). Thus for these two benchmarks, more memory barriers are inserted for \( JMM_{old} \) than \( JMM_{new} \). However, for \( JMM_{new} \) memory barriers are inserted in the end of constructor with final field writes. Therefore Sync requires more memory barriers under TSO and PSO for \( JMM_{new} \) than \( JMM_{old} \).

Among the benchmarks, only Ray has substantial number of constructors with final field writes (showed in Table 6.1). Hence only in Ray the number of memory barriers due to final fields is observable, which causes \( JMM_{new} \) has much more memory barriers than \( JMM_{old} \). And since final fields are treated in the same way, there is no great difference among the hardware memory models.

### 6.2 Total Cycles

The total cycles measure the overall performance of the benchmarks under a specific JMM and a hardware memory model. Two factors affect the total cycles: the inserted memory barriers due to JMMs and hardware memory models. The memory barriers cause some overhead to the performance according to the number
of the inserted memory barriers. Hardware memory models also have a significant influence to the performance. The numbers of total cycles are shown in the tables 6.2 to 6.6. These numbers are obtained by running each benchmark with four threads on the simulator with different hardware memory models. In order to observe the impact of the JMMs, we also obtain the total cycles of all the benchmarks without the restrictions of JMM under these hardware memory models. Thus for each benchmark we show the total cycles across the hardware memory models under three conditions: (a) no JMM is enforced, (b) $JMM_{old}$ is enforced, and (c) $JMM_{new}$ is enforced.

For SC no memory barriers are inserted in all conditions and only one number is obtained. SC is the strictest hardware memory model and no reordering is allowed among memory operations. Thus for each benchmark the total cycle in this situation is greater than all other hardware memory models.

Table 6.2 to total 6.6 below show the total cycles of the five benchmarks in different memory models. From the numbers we can see that the hardware memory models have a crucial impact to the overall performance because for all the three situations, the more relaxed the hardware memory models are, the fewer total cycles the benchmarks need. Thus in order to get a better performance while running Java multithreaded programs on multiprocessor platforms, it is important to choose a more relaxed hardware memory model.

Figure 6.1 to figure 6.5 display the performance difference of $JMM_{old}$ and $JMM_{new}$ for these five benchmarks. The percentages are calculated by the difference from $JMM_{old}$ to $JMM_{new}$ relative to $JMM_{old}$. Positive percentages denote
<table>
<thead>
<tr>
<th>SOR</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>917936137</td>
<td>805044089</td>
<td>796483143</td>
<td>737270330</td>
<td>732185230</td>
</tr>
<tr>
<td>OLD</td>
<td>917936137</td>
<td>815760725</td>
<td>813172621</td>
<td>750390974</td>
<td>748304439</td>
</tr>
<tr>
<td>NEW</td>
<td>917936137</td>
<td>815767831</td>
<td>813157673</td>
<td>758128916</td>
<td>756108943</td>
</tr>
</tbody>
</table>

Table 6.3: Total Cycles for SOR in different memory models

<table>
<thead>
<tr>
<th>LU</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>889305456</td>
<td>761524728</td>
<td>750106721</td>
<td>709623362</td>
<td>706913310</td>
</tr>
<tr>
<td>OLD</td>
<td>889305456</td>
<td>788256682</td>
<td>781451934</td>
<td>735899282</td>
<td>733335441</td>
</tr>
<tr>
<td>NEW</td>
<td>889305456</td>
<td>788289275</td>
<td>781456496</td>
<td>742066910</td>
<td>739577827</td>
</tr>
</tbody>
</table>

Table 6.4: Total Cycles for LU in different memory models

<table>
<thead>
<tr>
<th>SERIES</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>789634551</td>
<td>619083358</td>
<td>616091141</td>
<td>554711801</td>
<td>550539432</td>
</tr>
<tr>
<td>OLD</td>
<td>789634551</td>
<td>619084257</td>
<td>616158562</td>
<td>554713616</td>
<td>550604174</td>
</tr>
<tr>
<td>NEW</td>
<td>789634551</td>
<td>619083783</td>
<td>616130521</td>
<td>554712495</td>
<td>550541524</td>
</tr>
</tbody>
</table>

Table 6.5: Total Cycles for SERIES in different memory models

<table>
<thead>
<tr>
<th>SYNC</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>1221890359</td>
<td>858705676</td>
<td>849215204</td>
<td>754785342</td>
<td>753762523</td>
</tr>
<tr>
<td>OLD</td>
<td>1221890359</td>
<td>858706139</td>
<td>849261810</td>
<td>754785746</td>
<td>753806984</td>
</tr>
<tr>
<td>NEW</td>
<td>1221890359</td>
<td>858713605</td>
<td>849243153</td>
<td>754792310</td>
<td>753767148</td>
</tr>
</tbody>
</table>

Table 6.6: Total Cycles for SYNC in different memory models
<table>
<thead>
<tr>
<th>RAY</th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
<th>WO</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>1068641872</td>
<td>894826339</td>
<td>884314029</td>
<td>839672514</td>
<td>828621775</td>
</tr>
<tr>
<td>OLD</td>
<td>1068641872</td>
<td>894983094</td>
<td>884805297</td>
<td>839852743</td>
<td>829056537</td>
</tr>
<tr>
<td>NEW</td>
<td>1068641872</td>
<td>898245756</td>
<td>888585746</td>
<td>843751963</td>
<td>832967457</td>
</tr>
</tbody>
</table>

Table 6.7: Total Cycles for RAY in different memory models

performance improvement. While negative percentages mean performance deterioration. The figures show performance doesn’t change the same way under those benchmarks. The performance is different for different benchmarks. This is probably because every benchmark has different number of volatile variables, synchronization operations and final field writes. Moreover, $JMM_{old}$ and $JMM_{new}$ need to insert different numbers of memory barriers under different hardware memory models. These compositive effects decide the total cycles required for the benchmarks. However, we can also draw some conclusion. Generally 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 for volatile variables, especially for $JMM_{new}$. That is why the benchmarks with significant volatile variables have much worse performance under $JMM_{new}$ than $JMM_{old}$.

Figure 6.6 to figure 6.10 illustrate the performance difference between SC and other relaxed memory models for both $JMM_{old}$ and $JMM_{new}$. All the five benchmarks show that the hardware memory models have significant impact on the overall performance, the more relaxed the hardware memory model, the better the performance. These results are consistent with those results of [2]. From those fig-
Figure 6.1: Performance difference of $JMM_{old}$ and $JMM_{new}$ for SOR

Figure 6.2: Performance difference of $JMM_{old}$ and $JMM_{new}$ for LU
Figure 6.3: Performance difference of $JMM_{old}$ and $JMM_{new}$ for SERIES

Figure 6.4: Performance difference of $JMM_{old}$ and $JMM_{new}$ for SYNC
Figure 6.5: Performance difference of $JMM_{old}$ and $JMM_{new}$ for RAY

Under certain hardware memory model, the total cycles are determined by the number of memory barriers. More memory barriers cause more overhead to the benchmark and more total cycles are required for that benchmark. However, two special cases need to be noticed. The first one is the LU benchmark under PSO. In this case the memory barriers are the same but $JMM_{new}$ need a few more cycles to complete.

Figure 6.6: Performance difference of SC and Relaxed memory models for SOR
Figure 6.7: Performance difference of SC and Relaxed memory models for LU

Figure 6.8: Performance difference of SC and Relaxed memory models for SERIES

Figure 6.9: Performance difference of SC and Relaxed memory models for SYNC
Figure 6.10: Performance difference of SC and Relaxed memory models for RAY
more cycles than $JMM_{old}$. But after investigating how the memory barriers are introduced, we found those memory barriers are not from the same source. Volatile variables bring in the same number of memory barriers under certain software memory model. However, for the synchronization operations $JMM_{old}$ introduce more memory barriers than $JMM_{new}$, and for the final fields memory barriers are only required for $JMM_{new}$. All these make up of the memory barriers required for this benchmark. Since they are from different source, the overheads brought by them are not equal. Thus the total cycles for $JMM_{old}$ and $JMM_{new}$ are not identical although the number of memory barriers is the same. Another case is the Series benchmark under TSO and WO. For both $JMM_{old}$ and $JMM_{new}$ no memory barriers are necessary under these two hardware memory models. But it is impossible to get equal cycles for the two JMMs because of the non-determinism in scheduling threads. Thus it is reasonable to report an average number of several executions. In this case we can not claim the performance under one JMM is better than that under another JMM.
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we study the performance impact of Java Memory Model on out-of-order multiprocessor. Hardware memory model describes the behaviors allowed by multiprocessor implementations while Java Memory Model (JMM) describes behaviors allowed by Java multithreading implementations. The existing JMM ($JMM_{old}$) and the newly proposed JMM ($JMM_{new}$) are used in this study to show how the choices of JMM can affect the performance of multiprocessor platforms. To ensure that the execution on the multiprocessor with some hardware memory model does not violate the JMM, we add memory barriers to enforce ordering. A multiprocessor simulator is used to execute the multithreaded Java Grande Benchmarks under different software memory models and hardware consistency models.

The results show that $JMM_{new}$ imposes more restrictions than $JMM_{old}$ with regard to the volatile variable accesses. This will reduce the performance if there are significant number of volatile variable accesses under $JMM_{new}$ but it ensures the security of multithreaded program. Overall, the $JMM_{new}$ can achieve almost
the same performance as the $JMM_{old}$ and more importantly it guarantees that the in incompletely synchronized programs will not create security problems. In addition, the $JMM_{new}$ makes the implementation of JVM much easier.

With the popularity of out-of-order multiprocessors, more and more commercial and scientific multiprocessor platforms are put to use. It has a significant meaning to study the impact of JMM on out-of-order multiprocessors because Java is becoming more and more popular and a new JMM is proposed to replace the old one. It can be a guide for the revision and implementation of the new JMM.

### 7.2 Future Work

In our study, we get the overall performance impacts of $JMM_{old}$ and $JMM_{new}$ under different hardware memory models. The impact is due to a combination of different reasons. Therefore we get different impacts from different benchmarks. The future work can be done to analyze the effect of every individual reason. Thus we will have more understanding of the impact of $JMM_{old}$ and $JMM_{new}$. This may also be used as reference for revising JMM to improve the performance.

In the implementation of JMM, we insert the memory barriers directly in the source level, which is easy to implement but will bring some more overheads. Future work can be done to insert memory barriers in JVM or hardware level. This will be more precise and it is also possible to obtain the numbers of cycles due to different reasons.

The implementation of the new JMM may also affect the performance itself. Currently, there are not so many implementations. But with the freezing of the
new JMM, there will be much more different implementations. Comparing the performance difference of different implementations may be a challenging research topic.

The platform will have a significant influence to this experiment. Therefore some work may be done to improve the multiprocessor platform. One serious problem met in this study is the low efficiency of the simulator.
Bibliography


