



136  
701  
THS

1  
2002

This is to certify that the  
thesis entitled


INSERTING FENCES TO GUARANTEE SEQUENTIAL CONSISTENCY

presented by

Xing Fang

has been accepted towards fulfillment  
of the requirements for

Master's degree in Computer Science  
& Engineering

  
Major professor

Date 7/30/02



INSERTING FENCES TO GUARANTEE SEQUENTIAL CONSISTENCY

By

Xing Fang

A THESIS

Submitted to  
Michigan State University  
in partial fulfillment of the requirements  
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2002

## ABSTRACT

### INSERTING FENCES TO GUARANTEE SEQUENTIAL CONSISTENCY

By

Xing Fang

While sequential consistency is arguably the most intuitive and natural memory consistency model for programmers, many shared memory multiprocessors follow a relaxed consistency model. Relaxed consistency models allow reordering of reads and writes, enabling a variety of hardware level optimizations. This boosts system performance, but at the price of difficult programming and porting. In this thesis we present a compiler that achieves the best of both worlds: performance and ease of programming. The compiler provides a sequentially consistent view of the underlying architecture to the programmer by automatically mapping the program with sequentially consistent semantics to hardware supporting relaxed consistency. This is done by inserting memory fence instructions, where necessary, to force the program execution to be sequentially consistent. A simple thread-escape analysis is first performed on the programs, and the result is used to direct fence insertion algorithms in the later passes of the compiler. We present different fence insertion optimization algorithms developed and implemented in Jikes<sup>TM</sup>Reserch Virtual Machine.

To my parents.

# TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Memory Consistency Models</b>	<b>5</b>
2.1	Sequential Consistency . . . . .	6
2.2	Relaxed Memory Consistency . . . . .	7
2.3	The Best of Both Worlds . . . . .	8
<b>3</b>	<b>The Compiler</b>	<b>10</b>
3.1	Escape Analysis . . . . .	11
3.2	Delay Set Analysis . . . . .	12
3.3	Fence Instruction Insertion . . . . .	13
3.4	Structure of the Jikes RVM Optimizing Compiler . . . . .	13
<b>4</b>	<b>Fence Instruction Insertion</b>	<b>17</b>
4.1	Fence Instruction Semantics . . . . .	17
4.2	Delay Set Reduction . . . . .	18
4.3	Inserting Fences to Enforce Delays . . . . .	19
4.4	Fence Insertion Algorithms in Our System . . . . .	22

4.4.1	Naive Fence Insertion . . . . .	22
4.4.2	Local Optimization . . . . .	23
4.4.3	Global Optimization . . . . .	25
4.4.4	Exploiting Synchronized Blocks . . . . .	26
4.5	Fence Insertion Example with Our Algorithms . . . . .	27
<b>5</b>	<b>Performance Test Results and Analysis</b>	<b>32</b>
5.1	The Test Environment . . . . .	32
5.2	Performance Results with the Multiprocessor Machine . . . . .	34
5.3	Performance Results on the Uniprocessor Machine . . . . .	40
5.4	Counting the Fences . . . . .	42
<b>6</b>	<b>Conclusions and Future Work</b>	<b>43</b>
6.1	Conclusions . . . . .	43
6.2	Future Work . . . . .	44



# LIST OF TABLES

5.1	Performance on the multiprocessor machine, with simple escape analysis	34
5.2	Performance on the multiprocessor machine, manual escape analysis .	37
5.3	Performance on the multiprocessor machine, original escape analysis .	39
5.4	Performance on the uniprocessor machine, with simple escape analysis	40
5.5	Performance on the uniprocessor machine, manual escape analysis . .	41
5.6	Performance on the uniprocessor machine, original escape analysis . .	41

# LIST OF FIGURES

1.1	Example of Counter-intuitive Outcome with a Relaxed Memory Model	2
1.2	Possible Sequentially Consistent Results . . . . .	2
1.3	The Compiler . . . . .	4
2.1	Abstraction of the Memory Subsystem Under Sequential Consistency	7
2.2	Sequential Consistency vs. Relaxed Consistency . . . . .	8
3.1	The structure of Jikes RVM Optimizing Compiler . . . . .	14
4.1	Greedy Heuristic to Optimize Fence Insertion . . . . .	21
4.2	The Naive Fence Insertion Algorithm . . . . .	22
4.3	The Local Optimization Algorithm . . . . .	24
4.4	The Global Fence Insertion Algorithm . . . . .	24
4.5	Exploiting Synchronized Blocks . . . . .	26
4.6	An Example Control Flow Graph . . . . .	27
4.7	Result of Fence Insertion with Naive Insertion Algorithm . . . . .	28
4.8	Fence Insertion with Local Insertion Algorithm: Original State . . . . .	29
4.9	Result of Fence Insertion with Local Insertion Algorithm . . . . .	29

4.10	Dominance w.r.t. a Node: Set of Locations that can Enforce this Delay	30
4.11	Set of Delays Enforcible by a Potential Insertion Location . . . . .	30
4.12	Fences Inserted to Enforce Inter-Block Delays, w/o Global Optimization	31
4.13	Fences Inserted to Enforce Inter-Block Delays, w/ Global Optimization	31
5.1	Performance Results with Simple Escape Analysis . . . . .	36
5.2	Performance Results with Manual Escape Analysis . . . . .	38
5.3	Static Count of Syncs Inserted in Hot Methods of _201_Compress . .	42
6.1	Example of Delay Reduction . . . . .	45

# Chapter 1

## Introduction

Shared memory multiprocessors are becoming widely accepted in many areas of computing today. Programmers using such machines generally expect the behavior of the memory to be similar to a uniprocessor running concurrent threads of a single program. This memory model for multiprocessors, called Sequential Consistency[9], is an intuitive extension of the uniprocessor memory model. It is arguably the most natural memory consistency model to programmers—they assume sequential consistency even if they don't know exactly what it is.

Many shared memory multiprocessors follow a relaxed consistency model. Relaxed consistency models allow reordering of reads and writes, enabling a variety of hardware level optimizations, such as speculative execution and data prefetching. This boosts system performance, but at the same time it makes programming and porting difficult because the programmer is exposed to the various instruction reordering and atomicity constraints of memory operations. Results of execution may be counter-intuitive, as shown by the following example.

Initially,  $x = 0, y = 0, X = 0, Y = 0$   
 Thread 1                      Thread 2  
*s11*:  $X = x$                       *s21*:  $Y = y$   
*s12*:  $y = 1$                       *s22*:  $x = 1$   
 Intuitively impossible outcome:  $X = 1, Y = 1$

Figure 1.1: Example of Counter-intuitive Outcome with a Relaxed Memory Model

Figure 1.1 shows a parallel program, with two threads running on different processors, processor 1 and processor 2. Originally all the variables are zero. After the parallel execution we print out the value of X and Y. Under the sequential consistency model the result of execution should appear to be equivalent to one of the results shown in Figure 1.2, which are obtained by executing the two threads interleavably.

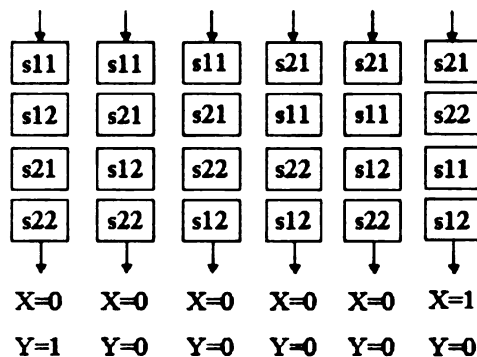


Figure 1.2: Possible Sequentially Consistent Results

With a relaxed memory model we can reorder some of the instructions. We observe that there is no dependence between *s21* and *s22*, so reordering them is legal. A possible execution order after the reordering is *s22* → *s11* → *s12* → *s21*, which yields the result X and Y both equaling 1.

This result is not equal to any of the results in Figure 1.2 and it is counter-intuitive. If X is 1 then *s22* must have executed before *s11*. Note that *s21* appears before *s22*

in thread 2 so intuitively we would expect  $s_{21}$  to execute before  $s_{22}$ . For the same reason we expect  $s_{12}$  to execute after  $s_{11}$ , so now  $s_{21}$  must have executed before  $s_{12}$ . Then Y should be 0, instead of 1.

The magnitude of the difficulties introduced by relaxed memory models have led some to argue that future systems should implement sequential consistency as their hardware memory consistency model because the performance boost of relaxed memory consistency models does not compensate for the burden placed on system software programmers[8]. For example, they argue that in Figure 1.1, the hardware should guarantee the ordering between  $s_{21}$  and  $s_{22}$ , as well as the ordering between  $s_{11}$  and  $s_{12}$ , in order to make the results sequentially consistent, and intuitive.

But actually this requirement is too strong. Not all the program orders (i.e., the orderings among the instructions, specified by the source program) have to be enforced to guarantee sequential consistency. Only part of the orders among the shared variable accesses need to be honored to achieve that. These program orders are called *delays* and the procedure to find the delays is called *Delay Set Analysis*[15]. In this example the delays are  $s_{21} \rightarrow s_{22}$  and  $s_{11} \rightarrow s_{12}$ , but in general not all the program orders are delays.

If a compiler can find a proper delay set  $\mathbf{S}$  for the parallel program and control the underlying hardware to enforce  $\mathbf{S}$ , then the programmer could treat the compiler and architecture as a whole system and regard it as sequentially consistent. Because  $\mathbf{S}$  is generally smaller than the original set of program orders, this system can still profit from the performance advantage of the relaxed memory model.

In this thesis we present implementation of such a system(Figure 1.3). The com-

piler is constructed on top of a multiprocessor system with a relaxed memory model. It performs delay set analysis and inserts fence instructions to enforce the delays, guaranteeing a sequentially consistent view to the programmer, regardless of the underlying memory model.

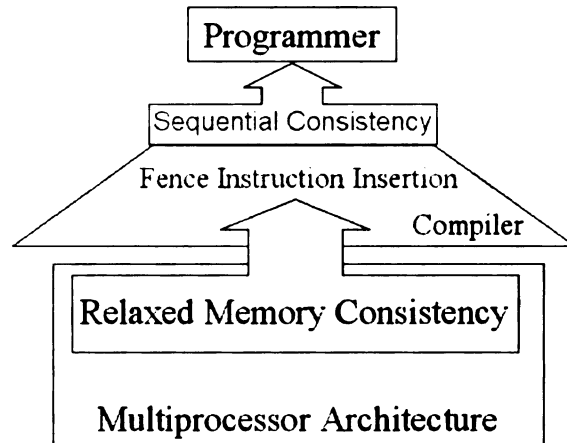


Figure 1.3: The Compiler

The remainder of the thesis is organized as follows. Section 2 introduces memory consistency models of multiprocessors and Section 3 describes the components of our compiler. We focus on the fence insertion optimization techniques in Section 4. Then in Section 5 we present our test results and analyze them. In Section 6 we conclude and look at possible work to be done in the future.

## Chapter 2

# Memory Consistency Models

Programmers always have some basic expectations about the results of their program. For example, in the uniprocessor case, if they issue a read of a memory location  $m$  immediately after a write of  $m$ , they will expect the value returned by the read to be the value just written into  $m$ . This simple memory semantics, in which a read of a variable returns the value of the most recent write to the variable, is most intuitive. Generally programmers also assume that all memory operations in a program are executed in *program order*, where *program order* is the order of the operations specified by the source program.

A memory consistency model for a shared address space specifies constraints on the order in which memory operations must **appear** to be performed with respect to one another[10].

Note that the only thing we care about a program is its observable behavior. In other words the program is looked upon as a black box and when we talk about memory consistency models, only the result of execution matters.



Popular memory consistency models include the intuitive sequential consistency model, and various relaxed consistency models.

## 2.1 Sequential Consistency

### **Definition 2.1.1 *Sequential Consistency***

*A multiprocessor system is sequentially consistent if the result of the execution of any program is the same as if all operations were executed in some global order and the operation of each parallel component appear in this sequence in the order specified by its program[12].*

Figure 2.1 depicts the abstraction of memory provided to programmers by a sequentially consistent system[1]. Every process **appears** to issue and complete memory operations one at a time and **atomically** in program order, i.e., a memory operation doesn't appear to be issued until the previous one from the same process has completed. In addition, the common memory appears to service the requests one at a time in an interleaved manner according to an arbitrary schedule. Memory operations appear *atomic* in this interleaved order. It should appear to all the processes **as if** one operation in the consistent interleaved order executes and completes before the next one begins[6].

Note that in reality the memory operations does not have to happen in the manner specified above. Only the result of execution is important. The system is sequentially consistent as long as the **result** is the same as one of the runs conforming to the specification in the previous paragraph. The model completely hides the underlying

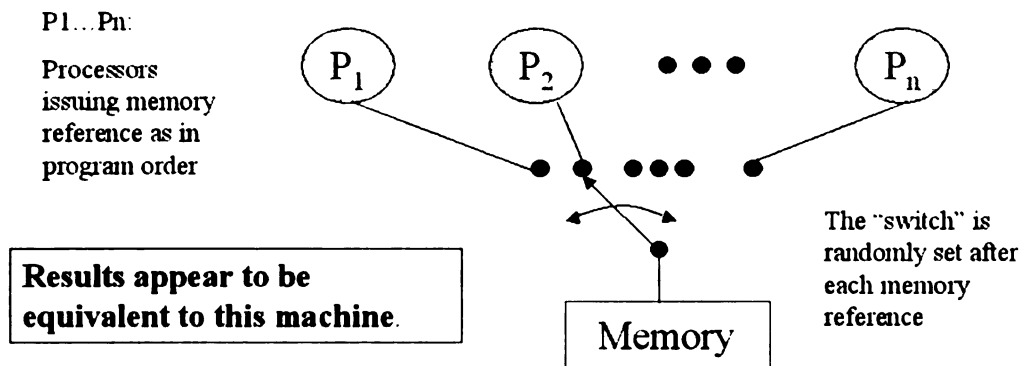


Figure 2.1: Abstraction of the Memory Subsystem Under Sequential Consistency

concurrency in the memory system hardware from the programmer.

Sequential consistency is intuitive but it greatly restricts the use of many performance optimizations commonly used by uniprocessor hardware and compiler designers. As a result various relaxed memory consistency models are proposed and implemented in today's commercial multiprocessor systems.

## 2.2 Relaxed Memory Consistency

Relaxed memory consistency models either relax the *program order requirement*, or the *write atomicity requirement*, or both, in order to enable various system optimizations.

Popular relaxed memory consistency models include processor consistency, weak ordering, and release consistency. They allow various reorderings to happen between memory accesses to different locations, according to their respective ordering constraints.

The Java memory model is another relaxed consistency model. It is different from the others in that it is a software model, specified in [7] and presumably supported by the virtual machines. It is intended to be efficient and easy to use but is actually poorly specified and difficult to understand[14].

The major problem with relaxed memory models is that they are not very intuitive, thus very difficult to use. A concrete example was shown in Figure 1.1, where a counter-intuitive outcome happens as the result of the reordering of two memory accesses. Theoretically an experienced programmer should be able to understand and/or deduce the outcome of a program running on a system with relaxed memory model, nevertheless programming on such machines is difficult and error-prone.

## 2.3 The Best of Both Worlds

Figure 2.3 compares the sequential consistency model against relaxed memory models. It seems that we are in the dilemma of having to choose from one end of the road, where programming is easy but the system is slow, or the other.

	<b>Sequential Consistency</b>	<b>Relaxed Consistency</b>
Pros	Simple and Intuitive	Boosted Performance
Cons	Inefficient	Counter-Intuitive Difficult to port

Figure 2.2: Sequential Consistency vs. Relaxed Consistency

Our compiler strives to achieve the best of both worlds, providing the ease of programming as well as enhanced performance, by inserting fences to guarantee sequential consistency on top of an architecture with a relaxed memory consistency

model. Reorderings are allowed when they are in agreement with the sequential consistency model, thus providing improved performance.

# Chapter 3

## The Compiler

The compiler is built on top of a relaxed memory model. It identifies the delays and inserts fence instructions to enforce them, providing a sequentially consistent view to the programmer. The underlying memory model could be a hardware model, such as weak ordering, or a software model, such as the Java memory model.

Our current compiler is based on the Jikes Research Virtual Machine(RVM)[3], from the IBM T.J.Watson Research Center. It runs on IBM AIX/PowerPC and Linux/IA-32 platforms.

The compiler does three analyses to achieve its goal: Escape Analysis, Delay Set Analysis, and Fence Instruction Insertion. We will discuss them one by one. Then we will take a look at the structure of the Jikes RVM optimizing compiler and see where our analyses reside.

## 3.1 Escape Analysis

Escape analysis is a technique that identifies the variables in a thread that are possibly accessible from another thread. This analysis is needed because delays only happen among shared variables.

We currently implemented two escape analysis algorithms in our compiler. One of them is called simple escape analysis. It works on a method and identifies those objects that can be accessed from outside the method. It is an iterative process. Initially the class variables, method parameters and return variables are marked as escaping because all of them can be accessed from outside the method. Later we inspect each instruction in the method and if a variable is accessible from an escaping variable then it will also be marked escaping. This process goes on until no more variables are marked escaping. The result of this analysis is fairly conservative, because escaping from a method (accessible from outside the method) is much more common than escaping from a thread (accessible from another thread).

To get an estimate of our compiler's performance, we employed another escape analysis called manual escape analysis. Here a human being inspects the source code of a program, does escape analysis by hand and makes the compiler aware of the escaping variables by marking them in the source code. The compiler then catches the markings and records the information. This approach is very optimistic because no compiler can ever reach the preciseness of the human brain. The manual escape analysis is only used to get an estimate of the upper bound of our system's performance.

## 3.2 Delay Set Analysis

The second phase of our compiler is Delay Set Analysis[15], which determines the minimal set of delays we need to enforce.

Let  $\mathbf{P}$  be the order enforced by the source program (i.e., program ordering) between operations. Throughout this discussion, operations are assumed to be atomic.  $\mathbf{P}$  is the *transitive closure* of the graph which contains the control flow edges of all the control flow graphs of each thread in the parallel program. *A node in the control flow graph represents an operation.* Two nodes  $m$  and  $n$  will be  $m\mathbf{P}n$  if there is a path between  $m$  and  $n$ . Let  $\mathbf{C}$  be a *conflict* relation on variable accesses. The *conflict* relation consists of the set of all pairs  $(v_i, v_j)$ , where  $v_i$  and  $v_j$  are operations containing conflicting accesses. Two memory references *conflict* if they access the same memory location in different threads that might execute concurrently, and at least one of them is a write[11].

A delay relation  $\mathbf{D}$  between two operations  $u$  and  $v$  forces  $v$  to wait until  $u$  completes execution. A *critical cycle* is a cycle of  $\mathbf{P} \cup \mathbf{C}$  that has no chords<sup>1</sup> in  $\mathbf{P}$ . The delay relation  $\mathbf{D}$  enforces sequential consistency if all  $\mathbf{P}$  edges in the critical cycles appear in  $\mathbf{D}$ . If  $\mathbf{D}$  consists of all the  $\mathbf{P}$  edges in the critical cycles, then  $\mathbf{D}$  is a minimal delay relation that enforces sequential consistency in any execution of the program.

Currently we haven't developed a real delay set analysis. Instead, if shared memory access  $u$  can reach shared access  $v$  then we assume there is a delay  $u \rightarrow v$  that

---

<sup>1</sup>For two nonadjacent nodes  $u$  and  $v$  in a cycle, a chord is an edge  $(u, v)$ .

we need to enforce. Basically we are now using the set of all the program orderings among the shared accesses as the delay set. This is guaranteed to be safe but it's very conservative.

This conservativeness adversely influences the performance of our whole system, and also makes our fence insertion optimization techniques less effective, as reflected in our test results.

### **3.3 Fence Instruction Insertion**

The last phase of our compiler concerns fence instruction insertion and optimization. It tries to determine the optimal set of locations in the compiled code to insert memory fence instructions. We devised and implemented three fence insertion/optimization algorithms and we will talk about them in detail in the next chapter.

### **3.4 Structure of the Jikes RVM Optimizing Compiler**

We implemented our compiler in the Jikes Research Virtual Machine(RVM) from the IBM T.J.Watson Research Center. It is a research Java virtual machine written almost entirely in Java.

Jikes RVM executes Java bytecode by compiling them to machine instructions at run time. It has three compilers: the baseline compiler, the optimizing compiler and the adaptive compiler. Our system was implemented in the optimizing compiler.



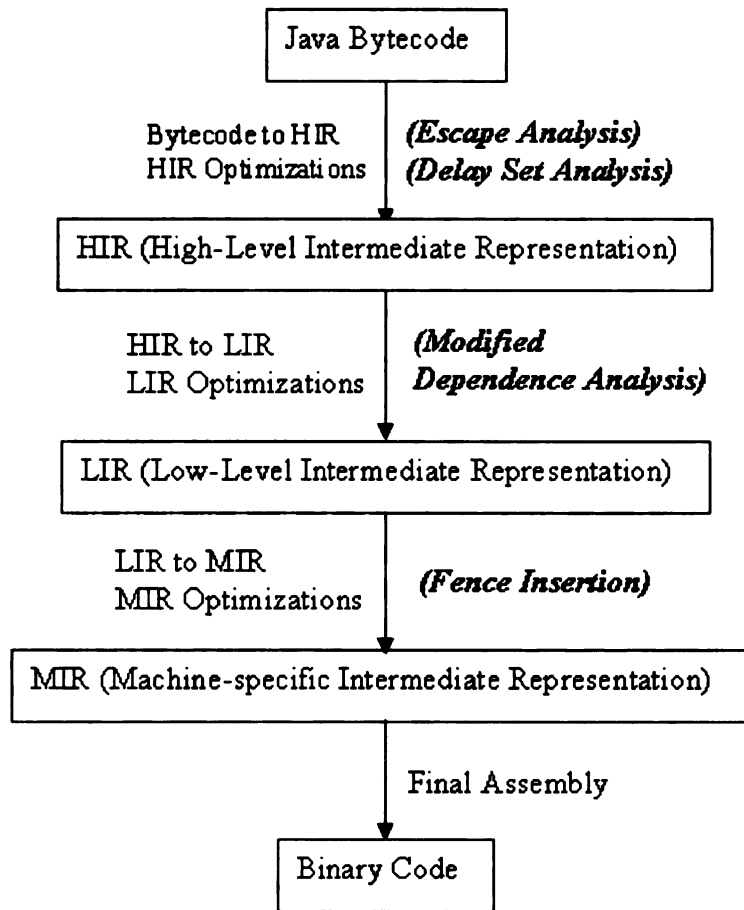


Figure 3.1: The structure of Jikes RVM Optimizing Compiler

Figure 3.1 shows the structure of the Jikes RVM optimizing compiler.

The unit of compilation in the optimizing compiler is a method. The compiler begins by translating Java bytecode to a *High-level Intermediate Representation (HIR)*, which is a register-based intermediate representation. In this process instructions are grouped into extended basic blocks, and method exception tables are constructed. Later the *HIR* is translated into *Low-level Intermediate Representation (LIR)*, expanding the *HIR* instructions into operations that are specific to the RVM's object layout and parameter passing conventions. Also a *dependence graph* is constructed

for later use. Instruction selection is performed next using the dependence graph, and the *LIR* is translated into *Machine-specific Intermediate Representation(MIR)* in this stage. Register allocation is performed, the *prologue* and the *epilogue* are added and executable code is emitted[3].

Escape analysis is concerned about identifying variables that are possibly shared among multiple threads. This should be done on a fairly high level in the compilation hierarchy, because not only is the information about sharedness available at the high level but also the program size is significantly smaller there , in order to reduce the complexity of analyses. As a result we implemented escape analysis in the HIR level. The same reasoning holds true for delay set analysis so it is implemented in the HIR level too.

Fence insertion has to be performed at a very low level and we implemented it after register allocation in our system. Fence instructions of different architectures has various forms and semantics, and only at the lowest level can we discover the specific architecture and insert fences correctly and efficiently.

The dependence graph is used for instruction selection. Each node of the graph is an LIR instruction and each edge corresponds to a dependence constraint between a pair of instructions, preventing code reordering during the instruction selection phase.

We modified the dependence graph construction code in the RVM, adding the delays as dependences. The delay set is already available by the time the dependence graph is constructed, and the delays have to be enforced for the system to provide sequential consistency. To do that we can not allow instruction reordering for two instructions that has a delay between them. Adding delays as dependences ensures

the *compiler* will honor the delays at *compile* time. Fences are later inserted to force the *processor* to guarantee the delays at *runtime*.

# Chapter 4

## Fence Instruction Insertion

In this chapter we discuss the theory of fence insertion optimization and present the three insertion/optimization algorithms implemented in our compiler.

### 4.1 Fence Instruction Semantics

Fence instructions are provided in modern processors as a mechanism for overriding their default relaxations. These instructions can be inserted in the machine code to enforce ordering among instructions.

The fence instructions of commercial architectures have various names and semantics: store barrier in the SPARC V8 architecture; read-read, read-write, write-read, and write-write fences(MEMBAR) in the SPARC V9 architecture; memory barrier(MB) and write memory barrier(WMB) in Alpha; sync in MIPS and the PowerPC architecture; memory fence(mf) in the Intel IA-64 architecture; and load fence(lfence), store fence(sfence) and memory fence(mfence) in the Intel Pentium 4 processor.

Because the semantics of a fence differ from architecture to architecture, we assume that a *fence* (or a synchronization instruction) has the following semantics[10]:

**Definition 4.1.1 Fence**

*A fence instruction imposes ordering between memory operations in such a way that when a fence instruction is executed by a processor, all previous memory operations of the processor are guaranteed to have completed. Furthermore, no memory operation of the processor that follow the fence instruction in the program is issued until the fence completes execution.*

If the architecture supports weaker fences(e.g. read-read,read-write,write-read, and write-write fences of the SPARC V9 architecture)than the above definition, then more efficient executions can be attained by modifying the fence insertion algorithms to take advantage of the weaker fences.

Our current compiler is based on the IBM PowerPC architecture, and *sync* instructions are inserted into the compiled code to enforce delays.

## 4.2 Delay Set Reduction

Delay Set Analysis gives us a minimal set of program orderings needed to guarantee sequential consistency. However, properties of the particular memory models are not considered in delay set analysis. This information can be exploited to further reduce the delay set. Each memory model specifies some ordering constraints that are forced by the model, and we do not need to enforce the delays that are enforced by these constraints.

Let  $\mathbf{D}$  be the delays found by the delay set analysis and  $\mathbf{D}_o$  be the delays enforced by the ordering constraints of the memory model(i.e., if  $u\mathbf{D}_ov$ , then  $u \rightarrow v$  match one of the constraint patterns and is preserved by the architecture). The following is a minimal delay relation that enforces correctness together with  $\mathbf{D}_o$  [15]:

$$\mathbf{D}_m = ((\mathbf{D} \cup \mathbf{D}_o)^+)^{\text{tr}} - \mathbf{D}_o$$

here  $^+$  and  $^{\text{tr}}$  denote transitive closure and transitive reduction operations respectively. Thus, only the delays in  $\mathbf{D}_m$  need to be implemented with fences or other special instructions, depending on the consistency model[10].

### 4.3 Inserting Fences to Enforce Delays

Fence insertion involves inserting memory fence instructions in the compiled code to enforce a computed delay set. Fence instructions are costly and the ultimate objective of the fence insertion algorithms is to minimize the number of memory fences executed by a program. However, reducing the total static number of memory fences inserted must be helpful and would serve as a good heuristic of this objective.

In our compiler fences are added as separate instructions. They are inserted **before** a node of the control flow graph to enforce one or more delays. To make it easier to describe we say the fence is inserted **at** the node and the node is called a *fence insertion location*. A node  $y$  can be marked as a fence insertion location to enforce a delay  $u\mathbf{D}_mv$  if  $y$  always executes after  $u$  and before  $v$  whenever  $u$  and  $v$

execute.

A conservative condition for finding a fence insertion location  $y$  that enforces a delay  $u\mathbf{D}_m v$  is: *If every path from  $u$  to  $v$  in the control flow graph of a thread goes through  $y$ , then  $y$  executes whenever  $u$  and  $v$  execute.*

To find fence insertion locations, we use the notion of *dominators with respect to a node*[10]. A node  $n$  *dominates* a node  $m$  ( $n \mathbf{dom} m$ ) if every control flow path from the program entry node to  $m$  goes through  $n$ [10]. The (classical) dominators of a node  $m$  are the *dominators with respect to the program entry node* of the control flow graph.

**Definition 4.3.1 Dominators with respect to a Node[10]**

*A node  $s$  dominates a node  $v$  with respect to a node  $u$  if every control flow path from  $u$  to  $v$  goes through  $s$ . This relation is denoted by  $s \mathbf{dom}_u v$ , and the set of such dominators  $s$  are denoted by  $\mathbf{dom}_u[v]$ .*

We use the iterative algorithm for classical dominators [4, 2] to find dominators with respect to a node  $u$  by treating  $u$  as the program entry node.

Minimizing the number of fence insertion locations is NP-hard because its decision version is an NP-complete problem[10, 12, 13]. Instead of checking each subset of the nodes in  $\bigcup_{u \in \mathbf{D}_m} \mathbf{dom}_u[v]$  to determine whether the fences in the subset enforces all the delays  $\mathbf{D}_m$ , an approximation algorithm can be used to minimize the number of fences inserted. It is a slight modification of the greedy heuristic developed to solve the optimization version of the **MINIMUM COVER** problem [5].

The algorithm listed below in Figure 4.1 is taken from [12]. It is the basis of our discussion of fence insertion optimization.

```

1   $U \leftarrow \bigcup_{(u,v) \in D_m} [v]$ 
2   $C \leftarrow \emptyset$ 
3  for each  $n \in U$ 
4       $C_n \leftarrow \emptyset$ 
5      for each  $u \in D_m$ 
6          if  $n \in \text{dom}_u[v]$  then
7               $C_n \leftarrow C_n \cup \{(u, v)\}$ 
8          end if
9      end for
10      $C \leftarrow C \cup \{C_n\}$ 
11 end for
12  $X \leftarrow D_m$ 
13  $M \leftarrow \emptyset$ 
14 while  $X \neq \emptyset$ 
15     Select a  $C_n \in C$  that maximize  $|C_n \cap X|$ .
16      $X \leftarrow X - C_n$ 
17      $M \leftarrow M \cup \{n\}$ 
18 end while

```

Figure 4.1: Greedy Heuristic to Optimize Fence Insertion

The algorithm first gets the *dominance w.r.t. a node* information and determines, for each possible fence insertion location, the set of delays that a potential fence can enforce. Then it greedily selects the location that enforces the largest number of delays as the next fence insertion location. Those delays that are enforceable by the selected fence are removed from the delay set and this selection process goes on, until all the delays are enforced.

Some other concerns exist that influence the choice of fence insertion locations. Since the goal of the relaxed memory consistency model is to make memory operations to different locations overlapped (pipelined) or reordered in order to hide the memory latency, we want to insert a fence as close to node  $v$  as possible in order to maximize the reordering and overlapping by processors if  $u \rightarrow v$  is a delay. Also, it is more desirable to insert a fence at a memory-barrier node  $n \in \text{dom}_u[v]$  that is located in



```

SharedAccessPool =  $\emptyset$ 
For each instruction  $u$  in the method
  if  $u$  is a shared memory access then
    put  $u$  into SharedAccessPool
    for each instruction  $v$  in SharedAccessPool
      if there is a delay  $v \rightarrow u$  then
        insert a fence instruction right before  $u$ , if there isn't one there
      end if
      if there is a delay  $u \rightarrow v$  then
        insert a fence instruction right before  $v$ , if there isn't one there
      end if
    end for
  end if
end for

```

Figure 4.2: The Naive Fence Insertion Algorithm

a less frequently executed path[10].

## 4.4 Fence Insertion Algorithms in Our System

We describe below the three different fence insertion algorithms implemented in our system.

### 4.4.1 Naive Fence Insertion

The first insertion algorithm we implement is the *naive* insertion algorithm, which does not optimize fence insertion. The algorithm is shown in Figure 4.2.

We check every pair of shared variable accesses in the method for possible delays between them. Here we need to check possible delays  $v \rightarrow u$  and  $u \rightarrow v$  because of the possible presence of loops in the method. Even if  $u$  appears in the static program after  $v$ , there is possibility that a program order  $v \rightarrow u$  exists. Also for the same

reason, we check  $(u,u)$  for delay for each shared variable access  $u$ .

In this algorithm a fence is inserted for each delay. The only exception is when there has been a fence before the shared variable, in which case an additional consecutive fence is redundant. Note that with our conservative delay set analysis, we have a delay between almost every pair of shared variables. As a result effectively every shared variable has a fence inserted before it. We can see this is very conservative and there must be room for optimization.

The other two algorithms try to optimize fence insertion. For this purpose the delay set is divided into intra-block delays (delays between instructions in the same extended basic block) and inter-block delays (delays between instructions in different basic blocks). Inter-block delays starting from the same block and end at another same block are combined into a single inter-block delay, between blocks (as compared to the original concept of delay between instructions).

#### 4.4.2 Local Optimization

We devised the *local* optimization algorithm to optimize the fences being inserted to enforce intra-block delays. The algorithm doesn't consider global control flow information nor seeks to optimize insertion of fences that enforce inter-block delays. That is why it is called local. Figure 4.3 is a description of the algorithm.

First, for each extended basic block that has a inter-block delay edge arriving into it, the algorithm inserts a fence at the entry of the block. This fence would enforce all the delays that come from outside the block because it dominates all the instructions

```

For each extended basic block  $B$  in the method
  if there exists an inter-block delay into  $B$  then
    insert a fence instruction at the start of block  $B$ 
  end if
  Get the set  $S$  of true register dependences in  $B$ ;
  Get the transitive closure of  $S$ , and remove it from the delay set  $D$ ;
   $SharedAccessPool = \emptyset$ 
  For each instruction  $u$  in  $B$ , start from the first instruction
  and iterate in program order
    if  $u$  is a shared memory access then
      for each instruction  $v$  in  $SharedAccessPool$ 
        if there is a delay  $v \rightarrow u$  then
          insert a fence instruction right before  $u$ ,
          (if there isn't one there)
           $SharedAccessPool = \emptyset$ 
        end if
      end for
      put  $u$  into  $SharedAccessPool$ 
    end if
  end for
end for

```

Figure 4.3: The Local Optimization Algorithm

```

For each extended basic block  $B$  in the method
  Collect those extended basic block  $C$  that has delay  $(B \rightarrow C)$  and
  put them into a set  $S$ .
  Set up a data flow framework to compute  $\mathbf{dom}_{\mathbf{B}}[S]$ 
end for
Apply the algorithm in Figure 4.1
Insert a fence at the start of each basic block chosen by the algorithm.

```

Figure 4.4: The Global Fence Insertion Algorithm

in the block: the fence is on every possible control flow path coming into the block. This leaves us with only the intra-block delays to worry about.

As we mentioned, the ordering constraints of the memory model could be exploited to reduce the delay set. In this algorithm we are exploiting the possible true register dependence among the instructions, which is honored by most processors (including PowerPC) when they do reordering. If there is a delay  $u \rightarrow v$  and there is a (transitive) true register dependence between  $u$  and  $v$ , then the delay will be enforced by the processor architecture and no fences need to be inserted for it.

As the last step of the local optimization, we start from the entry point of each extended basic block towards the end, checking each shared memory access  $u$  for possible unenforced delays with the form  $v \rightarrow u$ , where  $v$  is another shared memory access in the same block. A set *SharedAccessPool* is constructed to store the candidates of  $v$ . *SharedAccessPool* is initially set to  $\emptyset$ ; when we come across a shared access  $u$  it is added into *SharedAccessPool*. If a delay with the form  $v \rightarrow u$  is discovered, where  $v \in \textit{SharedAccessPool}$ , we insert a fence right before  $u$  and *SharedAccessPool* is reset to  $\emptyset$  because the newly inserted fence would take care of all other possible delays from instructions prior to  $u$ .

### 4.4.3 Global Optimization

We implemented the *global* optimization algorithm to exploit control flow information among the basic blocks and optimize fences being inserted to enforce inter-block delays. At the same time the algorithm also employs the techniques used in local

optimization to optimize fences inserted to enforce intra-block delays.

Figure 4.4 shows the part of the global optimization algorithm that optimizes fences inserted to enforce inter-block delays. Basically it finds out the *dominance w.r.t. a node* info and applies the algorithm in Figure 4.1.

#### 4.4.4 Exploiting Synchronized Blocks

Synchronized blocks are implemented in the Jikes<sup>TM</sup>RVM with a *sync* instruction at the end of the block. We exploited this *sync* instruction in the global optimization algorithm by removing the delays that has been enforced by this SYNC from the delay set. Figure 4.5 shows an example of such delay removal.

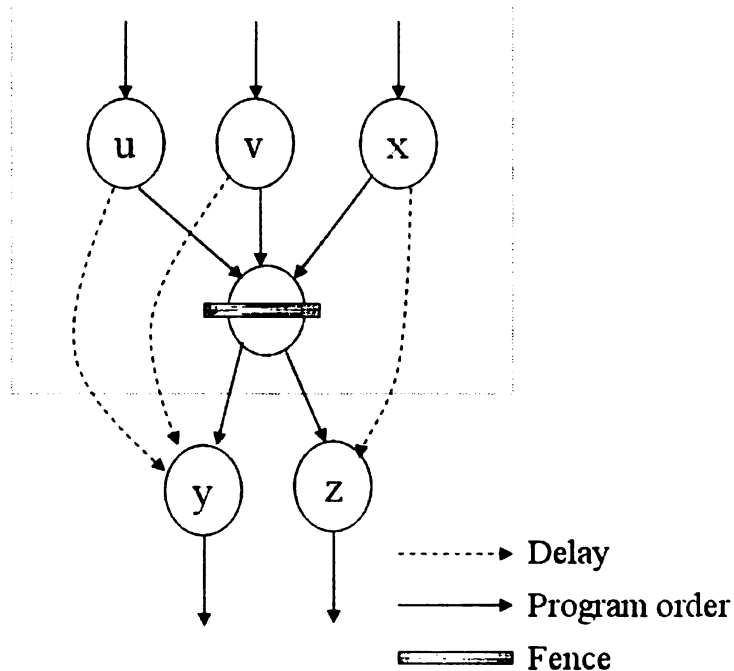


Figure 4.5: Exploiting Synchronized Blocks

The *sync* instruction (black bar in the middle of Figure 4.5) post-dominates in-

structions  $u, v$  and  $x$ , so delays  $u \rightarrow y$ ,  $v \rightarrow y$ , and  $x \rightarrow z$  has been enforced by the *sync* and can be removed from consideration in the fence insertion algorithms.

## 4.5 Fence Insertion Example with Our Algorithms

The following is an example of fence insertion with the three fence insertion algorithms. Figure 4.6 shows a simple control flow graph and the contents of one of its basic blocks, block  $c$ . The dark arrows are the delays we need to enforce. The black lines are the program orders.

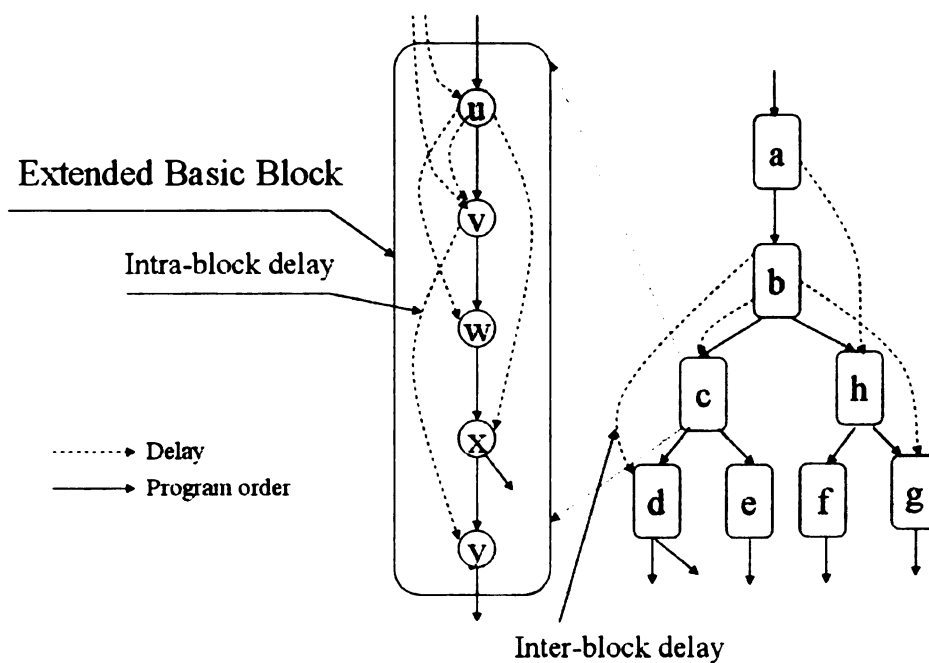


Figure 4.6: An Example Control Flow Graph

We first focus on fence insertion inside the extended basic block  $c$ . With the naive insertion algorithm we check each pair of shared memory accesses  $u$  and  $v$  in the method body and see whether there is delay between them. If there is a delay  $u \rightarrow v$

we insert fence before  $v$ . So in this case there would be five fences for this block  $c$ , as shown in Figure 4.7. Later we will see that at least three of these five fences are redundant, as we apply the insertion optimization techniques.

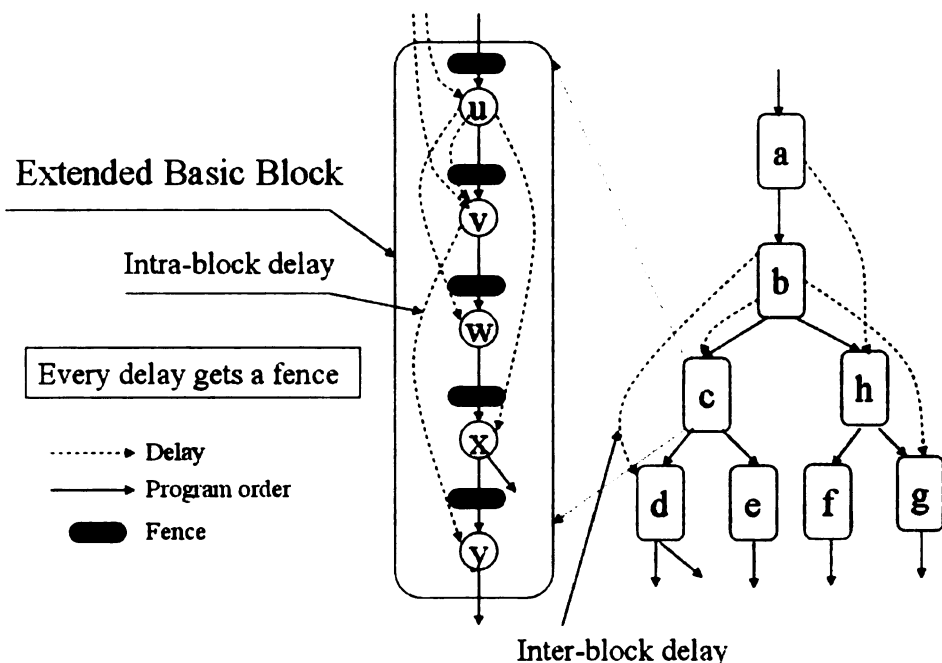


Figure 4.7: Result of Fence Insertion with Naive Insertion Algorithm

The local optimization technique first notices the inter-block delay  $b \rightarrow c$ , and inserts a fence at the start of block  $c$ . Then it identifies the true register dependences inside the block  $c$  (Figure 4.8), and gets the transitive closure of it (Figure 4.9). Note that the delay  $v \rightarrow y$  has been enforced by the hardware because  $(v, x)$  will not be reordered and  $(x, y)$  will not be ordered.

Now we start from the entry of the block  $c$  and check whether a fence is needed to enforce some delays. When we come to  $v$  we see the need for one so we add a fence there.

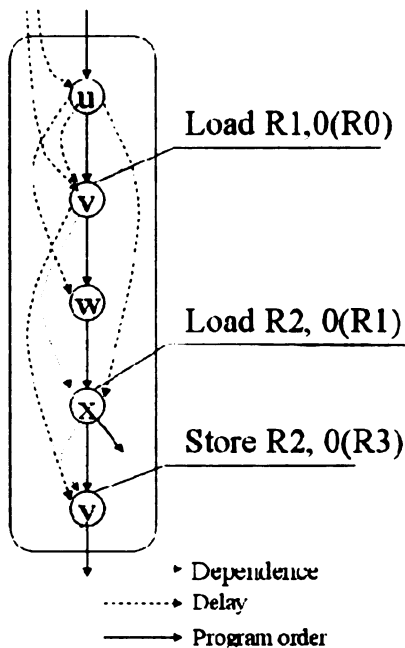


Figure 4.8: Fence Insertion with Local Insertion Algorithm: Original State

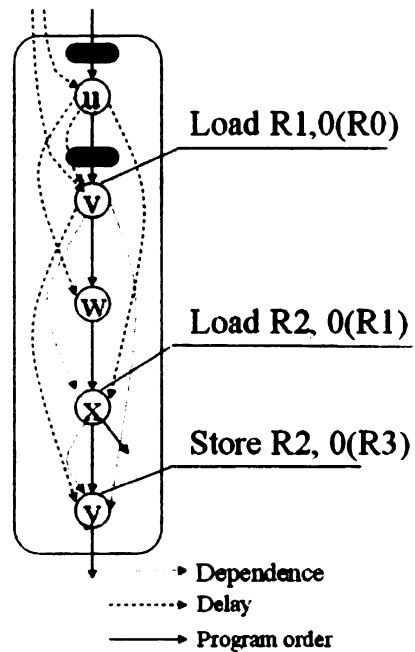


Figure 4.9: Result of Fence Insertion with Local Insertion Algorithm

The delays  $u \rightarrow w$  and  $u \rightarrow x$  are not directly enforced by a fence (there are no fences before  $w$  or  $x$ ). However  $u \rightarrow w$  has been enforced because the only path from  $u$  to  $w$  is from  $u$  to  $v$  to  $w$ . The fence before  $v$  is on the only path from  $u$  to  $w$ , which means the delay  $u \rightarrow w$  is enforced by the fence at  $v$ . This is an exploitation of the dominance relationship within the block. Delay  $u \rightarrow x$  is enforced too because the same reasoning holds true for the path from  $u$  to  $x$ .

We saw that with the naive fence insertion we need 5 fences in the block; now we only need 2. And sometimes the one at the entry of the basic block can be removed as well, thanks to the global optimization.

Now we apply the global optimization algorithm to optimize fence insertion to enforce inter-block delays. First we compute, with a data flow framework, the poten-



$b \rightarrow c$ : enforceable by $c$ $b \rightarrow d$ : enforceable by $c, d$ $b \rightarrow g$ : enforceable by $h, g$ $a \rightarrow h$ : enforceable by $b, h$
---

Figure 4.10: Dominance w.r.t. a Node: Set of Locations that can Enforce this Delay

$c$ enforces $b \rightarrow c, b \rightarrow d$ $d$ enforces $b \rightarrow d$ $g$ enforces $b \rightarrow g$ $b$ enforces $a \rightarrow h$ $h$ enforces $b \rightarrow g, a \rightarrow h$
--

Figure 4.11: Set of Delays Enforceable by a Potential Insertion Location

tial fences that can enforce a particular delay  $u \rightarrow v$ . It is  $\mathbf{dom}_u[v]$ . The results are shown in Figure 4.10.

Then we convert this information into the set of delays that each potential inserted fence can enforce(Figure 4.11).

With this information, we apply the greedy heuristic in Figure 4.1. Basically we try to find fence insertion locations that enforces as many delays as possible. In this example the block  $c$  is chosen first because a fence inserted at the start of  $c$  would be able to enforce 2 delays; then  $h$  is chosen. After that the four inter-block delays have been enforced by the two fences in Figure 4.13, instead of requiring four fences as in Figure 4.12.

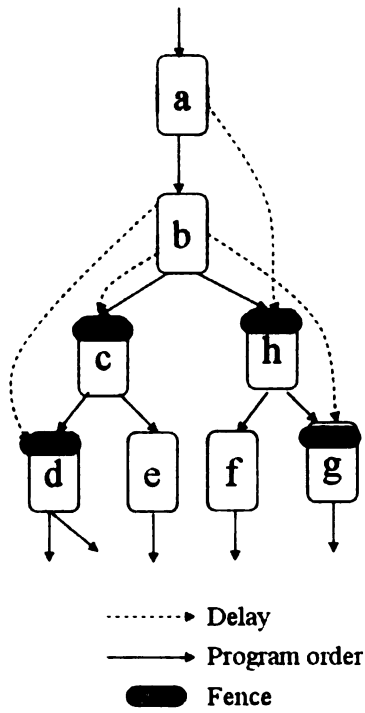


Figure 4.12: Fences Inserted to Enforce Inter-Block Delays, w/o Global Optimization

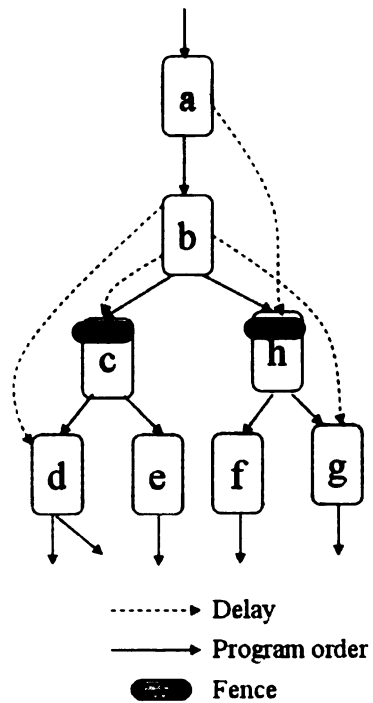


Figure 4.13: Fences Inserted to Enforce Inter-Block Delays, w/ Global Optimization

# Chapter 5

## Performance Test Results and Analysis

We have run some tests to get a quantitative estimate of the performance of our compiler. Our goal here is not to provide higher performance but ease of programming with comparable performance.

### 5.1 The Test Environment

Tests were run on two machines: a single processor IBM RS/6000 PowerPC in Michigan State University, with 512MB of physical memory, running AIX4.3.3; And an IBM SP machine with 8GB of physical memory, using 4 375MHz processors, located in the University of Illinois.

The benchmarks we used include SpecJvm98(.201\_compress, .202\_jess, .209\_db, .222\_mpegaudio, .227\_mtrt and .228\_jack), four multi-threaded benchmark programs

(elevator, philo, sor and tsp) from ETH, Zurich, and some programs (Crypt, LUfact, MolDyn, MonteCarlo, SOR, Series and SparseMatmult) from the Java Grande Forum Benchmark suite version 1.0. All the programs in SepcJvm98 are single-threaded, except `_227_mtrt`; the rest of the benchmark suite are multi-threaded.

We ran the test programs with our simple escape analysis, the manual escape analysis, and the original escape analysis that comes with Jikes RVM. We performed our fence instruction insertion and optimizations for each escape analysis on the two machines respectively. The programs were run for four times and the average execution time was recorded.

## 5.2 Performance Results with the Multiprocessor Machine

The following tables show the results we got on the multiprocessor IBM SP machine. The "Original" columns show the average execution time of the programs, using the default Jikes RVM Java memory model. The "Naive" columns show the execution time with the naive fence insertion algorithm. The "Local" and "Global" columns show the result with the local/global fence optimization algorithms, respectively. The fence insertion algorithms were described in Chapter 4. The unit of time is seconds and the slowdown numbers are shown in parentheses.

Table 5.1: Performance on the multiprocessor machine, with simple escape analysis

Benchmark	Original	Naive	Local	Global
_201_compress	16.971	351.477(20.710)	261.606(15.415)	240.346(14.162)
_202_jess	9.890	45.533(4.604)	36.062(3.646)	31.201(3.155)
_209_db	30.940	62.042(2.005)	48.738(1.575)	41.795(1.351)
_222_mpegaudio	13.476	321.295(23.841)	219.742(16.306)	212.066(15.736)
_227_mtrt	4.851	32.883(6.779)	28.165(5.806)	27.008(5.568)
_228_jack	19.552	133.980(6.852)	132.437(6.773)	130.859(6.693)
elevator	22.508	22.510(1.000)	22.509(1.000)	22.509(1.000)
philo	15.383	15.826(1.029)	15.808(1.028)	15.872(1.032)
sor	1.743	25.048(14.373)	12.206(7.004)	11.504(6.601)
tsp	2.519	29.740(11.806)	21.684(8.608)	17.073(6.778)
Crypt	23.550	42.822(1.818)	33.486(1.422)	33.066(1.404)
LUFact	3.185	31.712(9.957)	31.925(10.024)	31.778(9.978)
MolDyn	71.413	958.442(13.421)	470.258(6.585)	470.219(6.584)
MonteCarlo	13.842	37.115(2.681)	34.198(2.471)	33.954(2.453)
SOR	4.106	41.647(10.143)	41.962(10.220)	35.803(8.720)
Series	140.749	411.365(2.923)	376.404(2.674)	376.247(2.673)
SparseMatmult	3.827	43.293(11.313)	34.889(9.117)	35.061(9.162)

Table 5.1 shows the performance data with the simple escape analysis that we

described in Section 3.1.

The average slowdown with simple escape analysis and the global fence insertion algorithm is 6.062. This is actually not very bad because currently our emphasis is on the ability to provide sequential consistency, not on system performance. The escape analysis we have is very conservative and we even don't have a real delay set analysis yet.

Also with Table 5.1, we can see the average slowdown with the naive fence insertion algorithm is 9.036 and with the local insertion optimization algorithm, the average is 6.451. Comparing to the slowdown with simple escape analysis and global fence insertion optimization, this shows that our fence insertion optimization algorithms are effective, although the effect is not so much as we expected. The reason is again, we currently don't have delay set analysis. With a delay between almost every pair of shared accesses, the optimization techniques, especially the global optimization, can really do little. This result can be viewed as a lower bound of system performance.

Figures are easier to understand so we made a graphical representation of the data in Table 5.1, as shown in Figure 5.1. We can easily see the trend we mentioned from the figure.

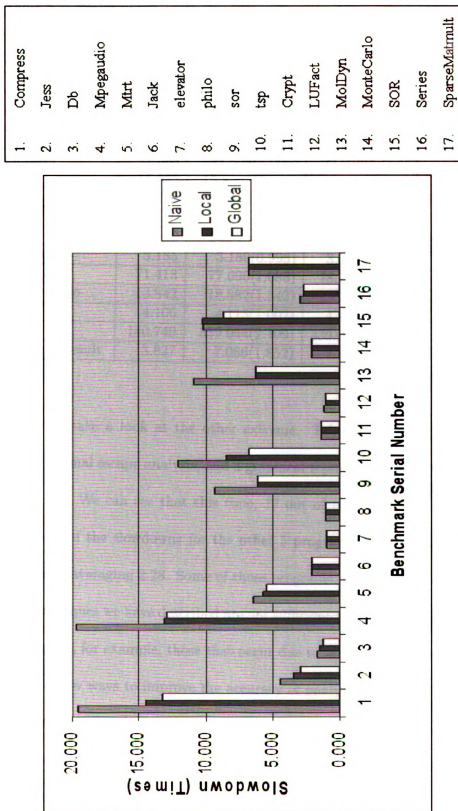


Figure 5.1: Performance Results with Simple Escape Analysis

Table 5.2: Performance on the multiprocessor machine, manual escape analysis

Benchmark	Original	Naive	Local	Global
_201_compress	16.971	18.512(1.091)	18.504(1.090)	18.451(1.087)
_202_jess	9.890	9.930(1.004)	9.945(1.006)	9.962(1.007)
_209_db	30.940	30.603(0.989)	30.664(0.991)	30.748(0.994)
_222_mpegaudio	13.476	31.894(2.367)	30.599(2.271)	30.572(2.269)
_227_mtrt	4.851	4.881454(1.006)	4.869(1.004)	4.884(1.007)
_228_jack	19.552	19.88054(1.017)	19.936(1.020)	19.965(1.021)
elevator	22.508	22.508(1.000)	22.508(1.000)	22.508(1.000)
philo	15.383	15.8367(1.029)	15.852(1.030)	15.899(1.034)
sor	1.743	1.834(1.052)	1.824(1.046)	1.756(1.008)
tsp	2.519	10.491(4.165)	10.244(4.066)	8.059(3.199)
Crypt	23.550	24.820(1.054)	24.896(1.057)	24.885(1.057)
LUFact	3.185	3.180(0.998)	3.172(0.996)	3.172(0.996)
MolDyn	71.413	77.696(1.088)	78.710(1.102)	78.206(1.095)
MonteCarlo	13.842	18.582(1.342)	18.750(1.354)	18.653(1.348)
SOR	4.106	4.113(1.002)	4.109(1.001)	4.114(1.002)
Series	140.749	339.044(2.409)	339.085(2.409)	338.728(2.407)
SparseMatmult	3.827	7.086(1.852)	7.391(1.931)	7.38(1.928)

Now let's take a look at the other extreme. Table 5.2 shows the performance data with manual escape analysis, and a graphical representation of this data is given in Figure 5.2. We can see that this time, 12 out of the 17 benchmarks showed no slowdowns, and the slowdowns for the other 5 programs are much less than in the previous case, averaging 2.28. Some of those programs performed poorly because the analysis techniques we have described are not sufficient to eliminate false dependences in the program, for example, those that occur due to indirect array indexing. We are looking into new ways to improve the accuracy of analysis.

This result can be viewed as a rough upper bound of possible system performance. It shows that it is feasible to provide sequential consistency without much performance degradations.



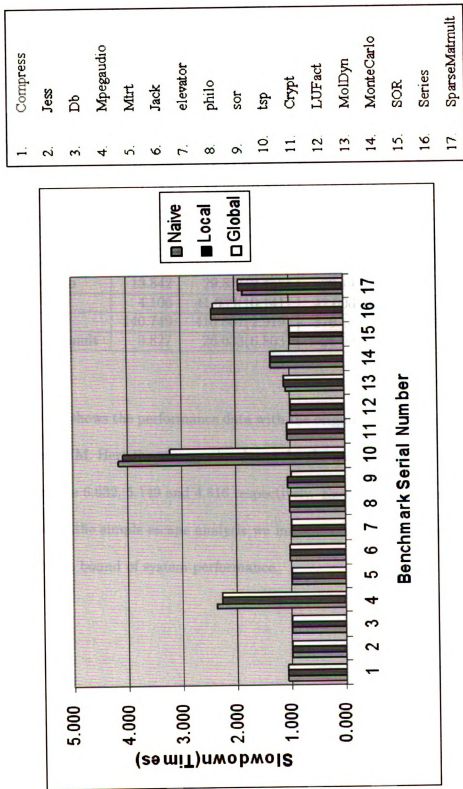


Figure 5.2: Performance Results with Manual Escape Analysis

Table 5.3: Performance on the multiprocessor machine, original escape analysis

Benchmark	Original	Naive	Local	Global
_201_compress	16.971	331.796(19.551)	246.462(14.523)	225.185(13.269)
_202_jess	9.890	43.634(4.412)	34.513(3.490)	29.520(2.985)
_209_db	30.940	50.916(1.646)	45.363(1.466)	39.565(1.279)
_222_mpegaudio	13.476	264.573(19.632)	176.170(13.072)	174.469(12.946)
_227_mtrt	4.851	31.252(6.443)	27.663(5.703)	26.529(5.469)
_228_jack	19.552	42.387(2.168)	41.911(2.144)	41.084(2.101)
elevator	22.508	22.509(1.000)	22.509(1.000)	22.509(1.000)
philo	15.383	15.827(1.029)	15.831(1.029)	15.782(1.026)
sor	1.743	16.250(9.325)	10.643(6.107)	10.607(6.087)
tsp	2.519	30.368(12.056)	21.297(8.454)	16.960(6.733)
Crypt	23.550	32.805(1.393)	32.722(1.389)	32.742(1.390)
LUFact	3.185	3.586(1.126)	3.524(1.106)	3.512(1.103)
MolDyn	71.413	781.062(10.937)	446.444(6.252)	444.847(6.229)
MonteCarlo	13.842	29.871(2.158)	29.141(2.105)	28.763(2.078)
SOR	4.106	41.639(10.141)	42.006(10.230)	35.751(8.707)
Series	140.749	410.861(2.919)	376.404(2.674)	376.216(2.673)
SparseMatmult	3.827	26.033(6.803)	26.008(6.796)	26.027(6.801)

Table 5.3 shows the performance data with the original escape analysis that comes with Jikes RVM. Here the average slowdowns for the naive, local and global insertion algorithms are 6.632, 5.149 and 4.816 respectively. Except this, the trend here is the same as with the simple escape analysis we implemented. This result could serve as a better lower bound of system performance.

## 5.3 Performance Results on the Uniprocessor Machine

The following tables show the the results we got on the uniprocessor PowerPC RS/6000 machine. They confirmed the same trend as on the multiprocessor machine.

The meanings of the columns are the same as in the previous section.

Table 5.4: Performance on the uniprocessor machine, with simple escape analysis

Benchmark	Original	Naive	Local	Global
_201_compress	25.781	440.999(17.106)	344.546(13.365)	321.451(12.469)
_202_jess	15.005	61.033(4.067)	54.734(3.648)	44.886(2.991)
_209_db	62.941	93.383(1.484)	84.486(1.342)	73.750(1.172)
_222_mpegaudio	16.582	283.844(17.118)	273.584(16.499)	271.338(16.364)
_227_mtrt	10.276	71.649(6.973)	62.919(6.123)	60.606(5.898)
_228_jack	27.338	60.752(2.222)	59.227(2.166)	56.621(2.071)
elevator	22.504	22.506(1.000)	22.505(1.000)	22.505(1.000)
philo	12.789	13.686(1.070)	13.337(1.043)	13.299(1.040)
sor	6.494	138.609(21.343)	65.861(10.141)	61.563(9.480)
tsp	8.050	168.844(20.974)	123.362(15.324)	97.862(12.157)
Crypt	104.921	215.429(2.053)	161.664(1.541)	159.471(1.520)
LUFact	16.103	183.542(11.398)	184.260(11.442)	183.518(11.396)
SOR	20.507	241.764(11.789)	240.816(11.743)	204.544(9.974)
Series	614.199	2158.710(3.515)	1964.300(3.198)	1958.620(3.189)
SparseMatmult	52.248	284.035(5.436)	235.728(4.512)	236.473(4.526)

Table 5.5: Performance on the uniprocessor machine, manual escape analysis

Benchmark	Original	Naive	Local	Global
_201_compress	25.781	55.117(2.138)	55.191(2.141)	56.086(2.176)
_202_jess	15.005	16.747(1.116)	16.816(1.121)	15.964(1.064)
_209_db	62.941	63.083(1.002)	62.977(1.001)	63.036(1.002)
_222_mpegaudio	16.582	47.624(2.872)	46.330(2.794)	45.766(2.760)
_227_mtrt	10.276	10.353(1.008)	10.098(0.983)	10.102(0.983)
_228_jack	27.338	27.818(1.018)	28.666(1.049)	29.105(1.065)
elevator	22.504	22.505(1.000)	22.541(1.002)	22.506(1.000)
philo	12.789	13.404(1.048)	13.263(1.037)	13.296(1.040)
sor	6.494	6.645(1.023)	6.621(1.020)	7.893(1.215)
tsp	8.050	55.171(6.854)	54.720(6.797)	41.692(5.179)
Crypt	104.921	105.320(1.004)	104.912(1.000)	106.316(1.013)
LUFact	16.103	16.295(1.012)	16.218(1.007)	16.384(1.017)
SOR	20.507	20.043(0.977)	20.180(0.984)	20.112(0.981)
Series	614.199	1745.110(2.841)	1742.900(2.838)	1742.550(2.837)
SparseMatmult	52.248	74.806(1.432)	72.445(1.387)	73.622(1.409)

Table 5.6: Performance on the uniprocessor machine, original escape analysis

Benchmark	Original	Naive	Local	Global
_201_compress	25.781	479.131(18.585)	358.700(13.914)	326.662(12.671)
_202_jess	15.005	61.345(4.088)	48.361(3.223)	41.325(2.754)
_209_db	62.941	76.787(1.220)	68.524(1.089)	60.474(0.961)
_222_mpegaudio	16.582	375.009(22.616)	249.463(15.044)	247.352(14.917)
_227_mtrt	10.276	69.155(6.730)	60.113(5.850)	57.700(5.615)
_228_jack	27.338	55.256(2.021)	55.554(2.032)	52.326(1.914)
elevator	22.504	22.550(1.002)	22.505(1.000)	22.504(1.000)
philo	12.789	13.274(1.038)	13.403(1.048)	13.264(1.037)
sor	6.494	87.563(13.483)	56.429(8.689)	56.200(8.654)
tsp	8.050	168.399(20.919)	121.971(15.152)	99.456(12.355)
Crypt	104.921	159.187(1.517)	159.110(1.516)	158.870(1.514)
LUFact	16.103	17.336(1.077)	17.201(1.068)	16.738(1.039)
SOR	20.507	240.596(11.732)	241.041(11.754)	203.852(9.941)
Series	614.199	2163.570(3.523)	1963.540(3.197)	1962.470(3.195)
SparseMatmult	52.248	215.444(4.124)	200.990(3.847)	204.008(3.905)

## 5.4 Counting the Fences

Method Name	Percentage in sequential time*	Naive (Manual/Simple)	Local (Manual/Simple)	Global (Manual/Simple)
Compressor. compress()	41.03%	0/51	0/36	0/28
Decompressor. Decompress()	25.07%	0/58	0/44	0/41
Compressor. Output()	8.18%	37/67	37/67	37/67
Decompressor. Getcode()	4.09%	16/58	16/56	16/56

\*: Profiled on SPARC Machine

Figure 5.3: Static Count of Syncs Inserted in Hot Methods of `_201_Compress`

To try to interpret the performance results we got the hot methods of the benchmarks by profiling and counted the static number of fences inserted into the hot methods. Figure 5.3 shows the hot methods in `_201_compress` and the number of fences inserted in them, with each of the escape analyses and fence insertion algorithms. The profiling was done on a SPARC machine, but the hot methods should be the same as on a PowerPC. We could see that with the manual escape analysis we didn't insert any fence instructions in the two hottest methods, while we inserted quite a lot of fences with the simple escape analysis. This explains the difference in the slowdown with the two escape analyses. Also we could see that with simple escape analysis, the number of fences we inserted did decrease as we employ more aggressive insertion optimization techniques. This explains the difference of slowdown in Figure 5.1, among columns denoting different fence insertion algorithms.

# Chapter 6

## Conclusions and Future Work

### 6.1 Conclusions

In this thesis, we described the design and implementation of a compiler that inserts fences to guarantee sequential consistency. We devised three fence instruction insertion algorithms and ran tests to get the two boundaries of system performance. We proved that it is feasible to provide sequential consistency on top of a relaxed memory model by inserting memory fences during compilation.

We are devising some reasonably accurate escape analyses and delay set analyses to make the compiler more realistic. When these analyses are deployed, we expect to get modest slowdowns somewhere in between the two extremes (closer to the result with manual escape analysis).

## 6.2 Future Work

We are still looking into new opportunities to optimize fence instruction insertion.

Currently we only implemented the compiler on the PowerPC architecture, which has only one memory fence instruction (*sync*). For other parallel architectures with relaxed memory models, there may exist several fence instructions with slightly different semantics. We can exploit these different kinds of fences and fine-tune our fence insertion algorithms. We are planning to port our implementation onto the Linux/Intel Pentium 4 platform, where there are three different fence instructions: load fence, store fence, and memory fence.

Another opportunity for optimization is to reduce the delay set before enforcing them with fences. Some delays may actually be redundant; They will automatically be enforced if some other set of fences are enforced. An example of this is shown in Figure 6.1.

Here delay  $u \rightarrow v$  is redundant because if the delays  $a \rightarrow b$  and  $e \rightarrow f$  are enforced then delay  $u \rightarrow v$  will automatically be enforced. So we can remove  $u \rightarrow v$  from the delay set before the fence insertion.

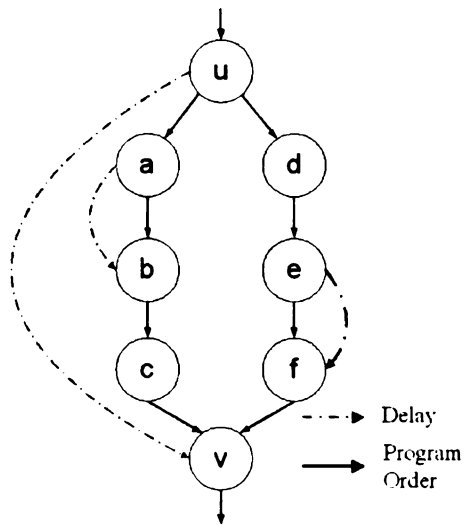


Figure 6.1: Example of Delay Reduction



# BIBLIOGRAPHY

- [1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, and et. al. The jalapeo virtual machine. *IBM System Journal*, 39(1), February 2000.
- [4] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, 1998.
- [5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [6] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann, 1999.
- [7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
- [8] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.
- [9] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [10] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1999. Technical Report UIUCDCS-R-99-2112.
- [11] Jaejin Lee. Hiding the java memory model with compilers. Technical Report MSU-CSE-00-29, Department of Computer Science and Engineering, Michigan State University, December 2000.
- [12] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings of The 2000 International Conference on Parallel Architectures and Compilation Techniques*, October 2000.
- [13] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers: Special Issue on Parallel Architectures and Compilation Techniques*, 50(8):824–833, August 2001.

- [14] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.
- [15] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 02328 7935