Relaxed Consistency and Synchronization in Parallel Processors

Richard N. Zucker
Department of Computer Science and Engineering
University of Washington

Technical Report No. 92-12-05
December 1992

Relaxed Consistency and Synchronization in Parallel Processors

by

Richard N. Zucker

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1992

Approved by_____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_____

Date_____

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Relaxed Consistency and Synchronization in Parallel Processors

by Richard N. Zucker

Chairperson of the Supervisory Committee: Professor Jean-Loup Baer
Department of Computer Science
and Engineering

Parallel programs often do not obtain close to linear speed-up when compared to a sequential version of the program running on a uniprocessor. There are many reasons that linear speed-up is not obtained. Two important ones are the overhead of synchronization and memory latency.

Synchronization, the coordination of the work done by different processors, is an overhead that does not exist in uniprocessor programs. Therefore, excessive time spent performing synchronization leads to a loss of performance. Many previous studies to evaluate this overhead have used artificial benchmarks with high levels of lock contention. In this dissertation I study both the effects of synchronization on the performance of real parallel programs and the impact of the efficiency of the implementation of the synchronization algorithm. The results show that the frequency of synchronization is the most significant factor leading to performance loss. When synchronization occurs sufficiently often, the implementation algorithm has a non-negligible effect.

Memory latency, the length of time from when a request to memory is initiated until it completes, is a major problem in multiprocessors. Many hardware and software enhancements have been proposed to deal with the problem. One of the ideas is relaxed models of memory consistency. Relaxed models, such as *weak ordering* or *release consistency*, replace *sequential consistency*, the usual intuitive model of how the memory of the system is implemented. With this change in the memory model, many architectural features can now be used that are not allowed under sequential consistency, but at the cost of imposing constraints to the programmer of parallel systems. In this dissertation I consider many of these architectural features such as bypassing, lock-up free caches and a software controlled cache coherence scheme I propose. I attempt to determine the performance benefits of using such features and which features provide the most benefit. The results show that relaxed consistency can provide significant performance gains for some programs and architectures. The choice of a given relaxed model does not significantly affect the gains. Software controlled cache coherence, a scheme that requires a smaller hardware investment, can provide equivalent performance in some cases and competitive performance in others.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Firstly, I would like to thank my advisor, Jean-Loup Baer, for his assistance, guidance and encouragement in my work. Without him I would never have completed my work at the University of Washington. I would also like to thank the current and past members of my committee, Susan Eggers, Hank Levy, Larry Ruzzo, Larry Snyder and Arun Somani, for their input.

DARPA and NASA provided me with a fellowship which greatly facilitated my research. They also arranged for my internship with the Massively Parallel Computing Intitiative (MPCI) Project at Lawrence Livermore National Labs, which allowed me to use the Cerberus simulator, a vital tool in my research, for which I would like to thank Eugene Brooks and the MPCI Project.

There are many current and former fellow graduate students at Washington who have provided support and technical assistance to me during my time in graduate school including: Gail Alverson, Craig Anderson, Rob Bedichek, Dave Bradlee, Tien-Fu Chen, Terry Farrah, Ed Felten, Simon Kahan, Eric Koldinger, Brian Lockyear, Cathy McCann, Rajendra Raj, Wendy Thrash, Raj Vaswani and Wen-Hann Wang. I also am grateful to graduate students from elsewhere, Sarita Adve of Wisconsin and Kourosh Gharachorloo of Stanford, for clarifying my understanding about relaxed consistency and as well as for their comments about my papers.

My friend Diane Gorenberg provided significant assistance with the writing style of Chapter 4.

Finally, I would like to thank my parents and family for their support and encouragement throughout my time in graduate school and before.

Chapter 1

Introduction

There is always a demand from users for more computing power than currently exists and currently the most practical way to obtain the greatest computing power is to use parallel processors. The use of parallel machines has become far more prevalent in the past few years and will continue to expand in the future. However, obtaining the utmost useful computing from multiprocessors is not easy. Many programs do not get anywhere close to a linear speed-up in execution time when the performance is compared to that on a single processor on the parallel machine, and the speed-up compared to a program optimized for a uniprocessor will generally be even lower than that. What are the factors that impede the performance gains?

The first factor, one that cannot be corrected, is that some problems simply are not parallelizable and gain nothing from being run on a parallel processor. A second factor is that there is a huge software base already in existence, which is designed to run on uniprocessors, and in which users have a large amount of money invested. Users do not want to discard these programs and write new ones from scratch. Rather, they would like to have the programs parallelized automatically, or be able to modify them only slightly so that they can be run in parallel. However, the basic algorithm of the program, designed for a uniprocessor, may not be the best one for a multiprocessor. Also, current automatic parallelization tools are not very advanced and fail to do a good job

of parallelizing many programs. Parallelizing a program manually can be very difficult. Writing parallel programs is an inherently more difficult task than writing sequential ones, and writing them for performance even more so (there are occasions where people have written parallel versions of sequential programs, only to find that they ran slower that the sequential one [71]).

Another major loss of performance for parallel programs is the effect of synchronization. It is an inherent property of parallel programs that they must spend some of their time coordinating their work, something which need not be done in the uniprocessor case. The time spent in this coordination, called synchronization, is time lost in comparison to the uniprocessor version of the program. Therefore, efficient synchronization, and minimal use of it, is important when trying to get the most out of a multiprocessor. Otherwise the synchronization overhead destroys the potential benefit of running the program in parallel.

A fourth factor is that parallelism adds another level of difficulty to the design of efficient operating systems and other run-time system support. Schedulers need to allocate processors fairly. They must give a program a reasonable number of processors. If too few are allocated, a program may not run efficiently. If too many are allocated, then processors, a valuable resource, may not be used efficiently. The scheduler must also consider processor affinity. If a thread has been running on a certain processor for a while, its footprint is in the cache and TLB, and moving the thread will cause it to run more slowly for a while due to the large number of cache and TLB misses. These are issues that scheduler designers are still dealing with and trying to balance in designing scheduling algorithms for multiuser multiprocessor machines. If they are not dealt with well, then there can be a significant degradation of performance. Also, whether threads of control are implemented at the user or kernel level is very important and involves certain tradeoffs that can adversely impact the user program, part of which depends upon how the user wrote the program. However, it does seem that a good balance may have been found in deciding at what level to implement threads [7].

Finally, the memory latency of parallel processors is higher than that of uniprocessors and hence, also leads to performance loss. There are various causes of this higher latency. There are cache coherence effects which introduce additional traffic on the interconnection network. The architecture's physical organization may have a greater processor-memory distance due to the constraints of organizing large numbers of processors and memory modules, so the base memory latency (in the case of no contention on the interconnect) will often be higher. This is especially an issue in the case of an architecture with distributed shared memory. In such a non-uniform memory access (NUMA) organization, data placement becomes a very important part of writing the program. If it is not done well, then too many accesses are to memory that is far away, and the high latency will drastically affect the program's performance.

I have studied two of the factors mentioned above that affect parallel program performance on shared-memory multiprocessors, namely synchronization and memory latency, and ways to deal with them. I have done this in the context of two different architectures for shared-memory multiprocessors. The first is a shared-bus multiprocessor. In such a machine main memory modules and processors are connected to a single bus. Each processor has a private cache for data and instructions. If data that is present in two (or more) processors' cache is written by one of the processors, the other processor's copy will be out of date. This cache coherence problem, can be solved in a number of different ways. In the shared-bus case cache coherence is maintained by using snoopy cache protocols [12]. When a processor initiates a cache action that can affect or be affected by other processors' cached values, it broadcasts the action on the bus so that the other processors invalidate or update their own copy if they have one, or can supply the updated copy if it is dirty in their cache. Each cache continually "snoops" on the bus to see if it needs to take any actions.

Cache coherence can be maintained via other types of protocols, one of which uses directories [29]. In such a scheme each line of memory has a bit vector associated with it, with one bit for each processor in the system. For each processor that has

a cached copy of that line, the corresponding bit is set. When a processor wants to write the line, an invalidation message must be sent to each processor with a copy of the line (there are a number of variations on this scheme for dealing with its excessive memory requirements [11, 31, 25, 58, 81]). Directories are often used when there is no single broadcast medium like a bus. The other architecture that I will examine uses a directory scheme to maintain cache coherence since it uses a multistage interconnection network (MIN) arranged in an Omega network to interconnect the processors to the main memory.

Both snoopy and directory protocols for cache coherence are considered to be hardware enforced cache coherence (although LimitLESS [31] can be considered a mixture). However, cache coherence can also be maintained in software. That is beyond the scope of this introduction and will be considered in Chapter 5.

This dissertation is organized as follows. In Chapter 2 I introduce how synchronization is used in parallel programs, concentrating on locks, and discuss hardware and software issues to improve the efficiency of locking. Then I present some studies on locking in parallel programs. In Chapter 3 I introduce and review one technique for alleviating the performance impact of high memory latency, namely relaxed models of memory consistency. In Chapter 4 I will present several studies I have done on the performance of relaxed memory models. These studies cover a wide range of architectural features and their implications. Chapter 5 is devoted to software controlled cache coherence (SCCC). I will explain why SCCC relates to relaxed models of memory consistency and present some simulations of SCCC. In Chapter 6 I will present general conclusions.

Chapter 2

Locking Patterns

2.1 Introduction

There are many paradigms for programming shared-memory multiprocessors. These include Single Program, Multiple Data (SPMD), where each processor is executing the same program, and threads, where each processor may be executing different routines, but all in the same shared address space. Regardless of the paradigm being used, the various processes or threads must synchronize. Constructs typically used for synchronization include semaphores, monitors, which can be built on top of semaphores and only allow one processor to execute certain routines at a given time and tuple-space (from Linda [27]) to atomically access shared data. At the lowest level though, all synchronization methods have a common mechanism: locks.

Locks are one of the most important parallel programming constructs. They allow the creation of critical sections, so that only a single process can access a set of data at a given time. Critical sections allow a process to modify data without another process reading the data at that time, and hence reading data in an inconsistent state, and stops another process from trying to modify the data at the same time the first one is modifying it, and hence causing the final state to be inconsistent. Also, other synchronization constructs such as barriers and synchronizing read and writes are often implemented using locks,

making locks a crucial element for most synchronization.

There are many ways to implement locks and many proposed lock algorithms. However, there is no single best algorithm. Many algorithms depend upon special instructions, which some architectures do not implement. Also, the cache coherence mechanism, interconnection network and memory latency affect the performance of lock algorithms. Finally, how a program uses locks will affect the performance of a lock algorithm, thereby favoring one algorithm over another. In this chapter I am going to give an overview of different locking techniques and present a trace-driven study of two of them on a shared-bus multiprocessor and how the techniques, and the act of locking affect program behavior, especially processor utilization.

2.2 Locking Techniques

Locking or mutual exclusion can be done totally in software. For example, Dekker's Algorithm [40] requires only spin waiting, memory interlock and a sequentially consistent[1] system. However, the code for Dekker's Algorithm for more than two processors is complicated and tricky to write correctly. To facilitate implementing locks most shared-memory multiprocessors supply some sort of hardware instruction (e.g. atomic exchange, Compare & Swap) which can be used to write cleaner and more efficient lock algorithms. The parallel programming libraries of such machines usually provide routines for locking and unlocking that use these instructions. However, the way in which these instructions are used can vary and affect the performance of the program.

Using an operation called Test&Set I will present a number of different algorithms for locking and discuss the ramifications of each algorithm. I will assume that Test&Set is implemented using an atomic exchange operation, which is considered to be an instruction that indivisibly exchanges a register with a memory location. A lock is represented in memory as a zero (free) or one (held or busy). To acquire a lock, a processor puts a

[1] This term will be explained in detail in Chapter 3. At this point, it is sufficient to consider that the system is serializable [46].

one in a register and then exchanges that register with the lock's location in memory. If the value brought into the register, the one that had been in the memory location, is a zero, then this processor now holds the lock. If the value is a one, then some other processor already holds the lock and the processor trying to acquire the lock must try again at some later time. To release the lock, the processor holding it must write a zero into the lock's memory location.[2] The key difference in most of the locking schemes is when to attempt an atomic update and when to retry if necessary. I will consider this in the context of a shared-bus multiprocessor with hardware enforced cache coherence using an invalidate based protocol, although many of the concepts apply to other protocols and other types of shared-memory multiprocessors. Much of this review is from work by Anderson [8, 9, 10].

```
repeat
until (test-and-set( lock ) == FREE);
```

Figure 2.1: Locking using Test&Set

The simplest way to implement a lock routine is to use a loop that continually executes a Test&Set until the lock is acquired, as shown in Figure 2.1. The problem with this scheme is that if several processors are waiting for the lock, then continually executing Test&Sets will generate a great deal of bus traffic. This is because when a processor tries to do an atomic exchange, it must first invalidate the other cached copies. So, it will always be requesting the bus to fetch a new copy of the line containing the lock since its own copy will have been invalidated by the other processors (if locks are not cached the same problem with bus traffic will exist with just one waiting processor since the exchange will be done directly at the memory module). This increased level of bus traffic will slow down the processors that are computing (as opposed to synchronizing), including the processor holding the lock. Clearly we want to reduce the bus traffic

[2] On some systems it is necessary that the lock be released by using an exchange of zero rather than a normal store, or the value does not get updated properly.

introduced by spin-waiting. A solution is to implement a Test&Test&Set sequence.

```
repeat
    while (lock == BUSY);
until (test-and-set( lock ) == FREE);
```

Figure 2.2: Locking using Test&Test&Set

Test&Test&Set was introduced by Segall and Rudolph [86]. In this scheme (see Figure 2.2) the processor reads the lock variable until it observes that the lock is free. Then it does a Test&Set and attempts to acquire the lock. If that fails (another waiting processor acquired the lock before it could), the processor reverts to simply reading the lock variable until the lock is released again. The advantage of this algorithm is that the processor is only reading the variable while waiting; it is not writing it. So it can read the copy that is in its cache, which does not cause any bus traffic, and hence does not adversely impact other processors. This provides a great performance gain over Test&Set.

Under light loads Test&Test&Set works well, but it can cause excessive bus traffic at times. Consider a situation where there are n processors trying to acquire a lock and they are all spinning on a cached copy of the lock and hence, not causing any bus traffic. When the lock is released, each spinning processor will have its cached copy invalidated. Therefore each one will then attempt to fetch a new copy of the lock since each one is continually reading the lock. Since the loop in Figure 2.2 is very small, there will be n almost simultaneous bus requests.[3] Once a processor has fetched a new copy of the line, it will complete its check of the lock and see that the lock is available. So the processor will attempt a Test&Set. Because of the small size of the loop, there will again be n almost simultaneous bus requests. But this time it has more serious repercussions. Each of these requests is for exclusive access, in order that the requesting processor

[3]If read snarfing [72] is used many of those requests will be satisfied without causing bus transactions (if the bus uses split transactions, read snarfing may provide less benefit than otherwise).

can do an atomic exchange in its cache. The first processor to do the exchange will acquire the lock and enter the critical section. The next processor to do the exchange will see that the lock is not free and start a read-only loop. The exchange by the third processor to do the exchange will invalidate the copy of the lock in the cache of the spinning processor. So that processor must request another copy from memory and then both of those processors will spin. The next processor to do a Test&Set will invalidate the copies of TWO spinning processors, both of which now must fetch new copies of the lock from memory. And so on for the other processors. How much this behavior impacts performance depends upon the type of bus, the arbitration protocol and some nondeterministic elements such as bus usage by asynchronous I/O and other programs running on other processors. In the worst case it can take $O(n^2)$ time for the system to stabilize [10].

Anderson [10] considers a number of ways to implement locks that do not exhibit under high loads the deleterious behavior of Test&Test&Set as do Mellor-Crummey and Scott [76, 77]. However, I will review queuing locks from [56] which also deal with this problem, since I implement them in my simulator (see Section 2.3). In a simulation environment queuing locks will provide equivalent performance to MCS locks [76] (which are implemented in a very similar fashion) and are more easily implemented than the solutions in [10].

In queuing locks a processor wanting to acquire a lock performs a single atomic exchange operation to get the address of a memory location, say M, and a special value, say A. It also stores an address N and a value B for the next processor. It then spins by reading the value stored in memory location M until that value is different from A. It does its spinning by reading the value in the cache, therefore causing no bus activity. When the value is different from A, it knows that it has acquired the lock. When it releases the lock, it will change the value B in location N to some new value C thus passing the lock to another waiting processor, if any. Each processor spins on a different memory location, and therefore there is no problem with contention since the lock is

handed off to a specific processor. A similar scheme is described in Anderson [10], but its implementation requires a ReadandIncrement instruction, or a small critical section. Queuing locks requires only the atomic exchange.

An example of queuing locks is given in Figures 2.3 - 2.8. In Figure 2.3 is the state of the lock structure after being initialized. In Figure 2.4 a processor has set up its own data structure for an attempt to acquire the lock and in Figure 2.5 it has done an atomic exchange on the lock structure and sees that it now has the lock. In Figure 2.6 a second processor is about to attempt to acquire the lock and in Figure 2.7 it has done its atomic exchange. When the processor sees that it does not have the lock, it will spin-wait. In Figure 2.8 the first processor has released the lock and has changed the value in Cell 1 to something other than $V1$, and the second processor now has the lock.

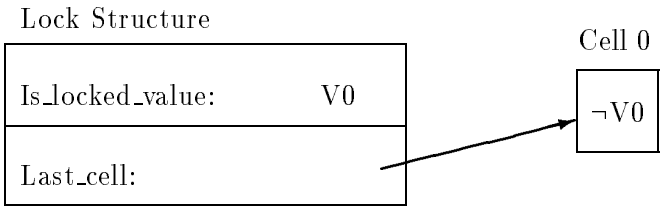


Figure 2.3: Queuing Locks - Initial

Queuing locks clearly has higher overhead than the simpler lock algorithms due to the more complex lock structure and the need to exchange two values atomically. But the overhead is low enough that in the case of low lock contention it should have little impact, and in the case of high contention the superiority of the algorithm should be more important.

Comparison of the lock algorithms I have reviewed and others has been done in Anderson [10], Graunke and Thakkar[56] and Mellor-Crummey and Scott [77]. However, those studies used artificial benchmarks, sometimes with artificially high levels of lock contention. The questions then are two-fold: what are the locking patterns of real parallel programs, and how do more advanced locking algorithms work in the case of

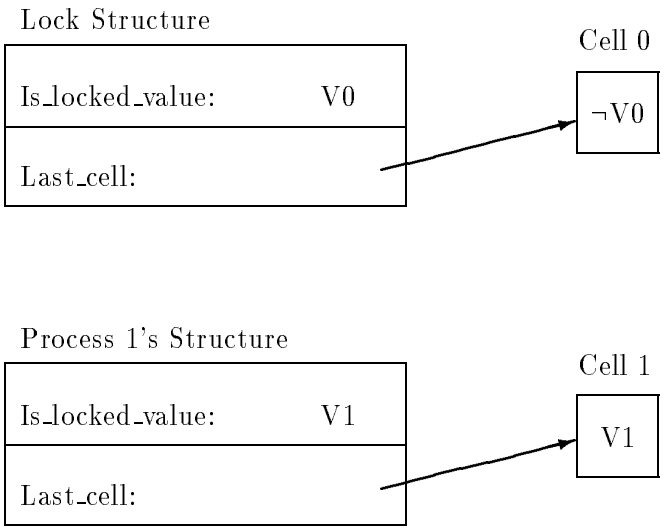


Figure 2.4: Queuing Locks - First processor about to enqueue

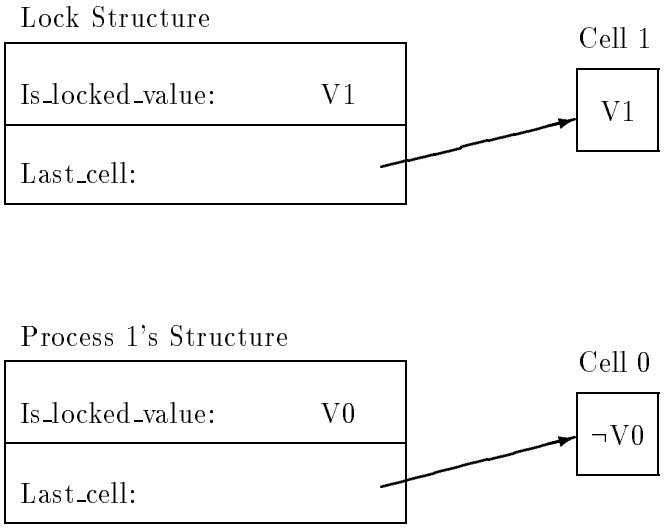


Figure 2.5: Queuing Locks - First processor enqueued, has lock

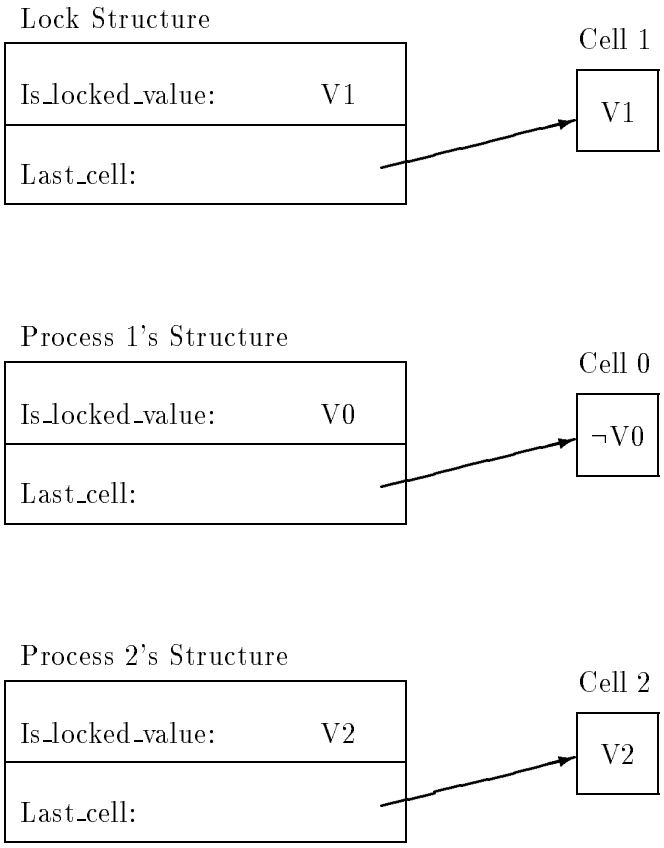


Figure 2.6: Queuing Locks - 2nd processor about to enqueue

real programs? This is what I have examined in a trace-driven study.

2.3 Methodology

2.3.1 Traces

My studies were done using trace-driven simulation. This performance evaluation tool allows me to contrast a number of different architectural variations as well as to assess the effect of changes in system parameters. A very important aspect of the trace-driven simulation, for the purposes of this study, is that I am able to analyze the "ideal" behavior of the traced programs, i.e., I can determine how long any section of the program would

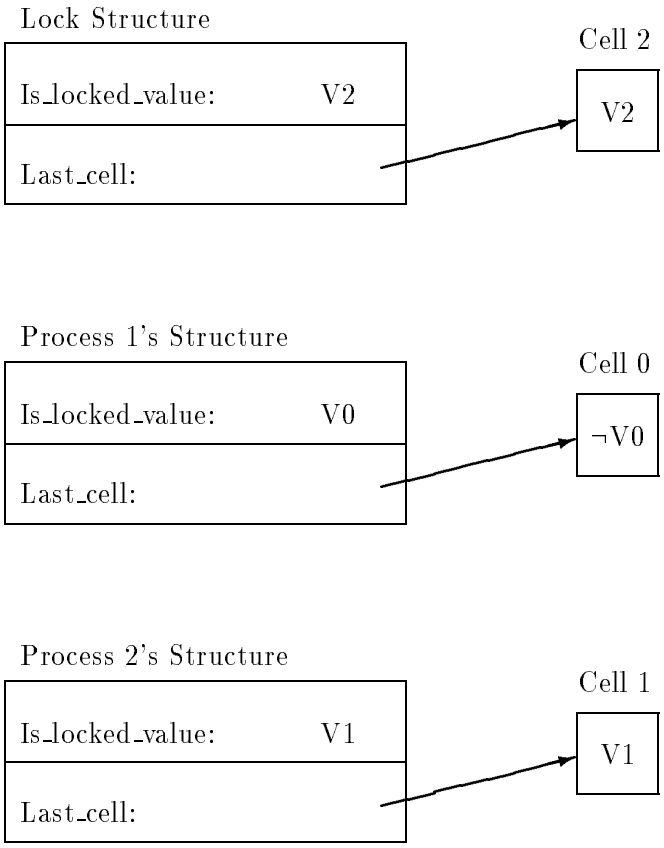


Figure 2.7: Queuing Locks - 2nd processor enqueued

take given no interference from other programs or stalling due to cache misses.

I used traces of programs running on a Sequent Symmetry Model B with 20 Intel 80386 processors. These traces were collected using the MPTrace system [43]. MPTrace is an in-line tracing technique. It only saves the entry address of each basic block and memory references within that block that cannot be statically reconstructed. In a post-processing phase the trace is expanded to give the full memory reference trace. This includes the number of cycles needed to execute each instruction, assuming no wait states. All times given in the statistics are expressed in units of these cycles.

MPTrace provides us with a per processor trace file of all memory references. All lock-spinning is removed from the trace file. Only the actual lock operation is left, and

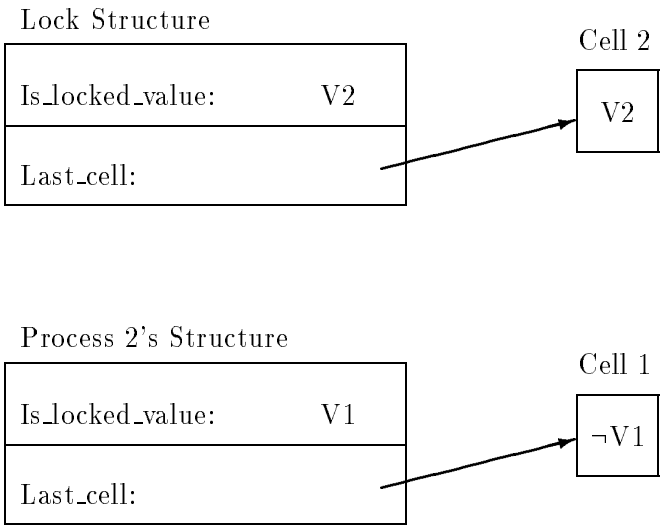


Figure 2.8: Queuing Locks - 1st processor has released lock, 2nd has acquired it

the lock operation must be simulated (see Section 2.3.4).

2.3.2 Model architecture

I simulated a bus-based architecture similar to the Sequent Symmetry Model B (cf. Figure 2.9) from which the traces were collected. Each processor has a two way set-associative 64 Kbyte cache. The line size is 16 bytes. The caches are write-back with LRU replacement. The Illinois protocol is used for hardware enforced cache coherence [12]. The cache-bus interface includes a four element buffer. All memory requests, write-backs, cache-cache transfers, and coherence actions initiated by the processor must pass through this buffer. If a dirty line is in the buffer to be written-back, it is visible to the cache coherence mechanism.

The bus modeled is a 64 bits wide (data and address) split transaction bus. Arbitration is round-robin. A split transaction occurs only on memory requests. While the read/write is performed in the memory module or buffered in the memory controller, the bus is not held so that it may be used by other devices. This implies that a request

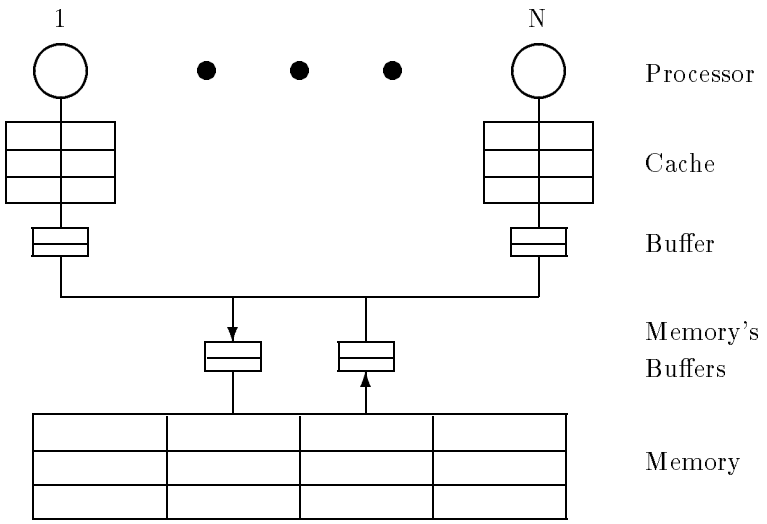


Figure 2.9: Model Architecture

may arrive at the memory while a previous request is being processed. Hence my model incorporates a two element buffer at the memory input. This also means that the bus may be busy when a memory access completes. Therefore I have incorporated a two element buffer at the memory output.

The memory has an access time of three cycles. Assuming no contention in the buffers or on the bus, a cache read miss causes the processor to stall for six cycles: One cycle to send the request to memory, three cycles to access memory, and two cycles to send the 16 byte line back since the bus is eight bytes wide. The caches use a write allocate strategy on a write miss and consequently a write miss also causes a six cycle stall.

2.3.3 Benchmarks

The programs that I simulated include VLSI CAD tools and scientific programs written in either C or C++. The C++ programs (the first three in Table 2.1), were written using the Presto [19] programming environment. Presto consists mainly of a number of C++ classes which provide for synchronization and user level threads. The scheduling and context switching of the threads are executed at the user level. Thus, the instructions

that perform the thread management are in the trace. In the C traces (the last three in Table 2.1) these system functions are not included.

The three Presto programs are Grav, Pdsa, and FullConn. Grav implements the Barnes and Hut clustering algorithm for simulating the time evolution of large numbers of stars interacting under gravity [47]. The program trace ran for three timesteps of evolution for a system of 2000 stars. Pdsa [89] does topological optimization using simulated annealing. FullConn is a run of a Synapse [92] distributed simulation of a fully-connected processor network.

The three C programs are Pverify, Qsort, and Topopt. Pverify [45] is a combinational logic verification program which compares two different circuit implementations to determine whether they are functionally (Boolean) equivalent. The circuits used for the trace were combinational benchmarks for evaluating test generation algorithms. Topopt [45] does topological compaction of MOS circuits using dynamic windowing and partitioning techniques. It is based upon a simulated annealing algorithm for its topological optimizations. Its input was a technology independent multi-level logic circuit. Qsort [63] is a quicksort program run on 1,000,000 random integers. This is not the best benchmark since sorting is more likely to be done as a subroutine of a program and therefore is not typical of an entire program. In addition, this program was written for research purposes and only sorts integers, which again may not be typical of real programs. Nonetheless, it provides some useful insight as long as one keeps these limitations in mind.

Tables 2.1 and 2.2 list "ideal" (in the sense given in Section 2.3.1) statistics about these traces. The programs were run on a system with either 9, 10, or 12 processors being active. Due to the allocation scheme used in Presto most data in the Presto programs is allocated as shared even when it need not be. This is reflected in the numbers given; i.e., a very high proportion of shared data. By contrast, in the C programs only a little more than a third of the data references are to shared data. The column "Work Cycles" refers to the number of cycles that the traced instructions would take to execute assuming the "ideal" conditions: no cache misses or other stalls. Grav and Qsort have been simulated

with significantly longer traces than those shown in Table 2.1 with no change in the basic results presented in Section 2.4.

Table 2.1: Benchmark Ideal Statistics
Cycles and references are averages per processor and are in 1000's.

Program	# of	Work	References		
	Proc.	Cycles	All	Data	Shared
Grav	10	2,841	1,185	423	377
Pdsa	12	2,458	1,206	431	410
FullConn	12	3,848	967	346	332
Pverify	12	5,544	2,431	682	254
Qsort	12	2,825	1,177	252	142
Topopt	9	10,182	4,135	1,113	413

Table 2.2: Benchmark's Ideal Lock Statistics
Lock pairs and nested locks are averages per processor.
The average held and total held are in cycles.

Program	Lock	Nested	Avg.	Total	% of
	Pairs	Locks	Held	Held	Time
Grav	6389	2579	200	1,131	39.8
Pdsa	3110	1467	190	510	20.7
FullConn	652	134	334	210	5.5
Pverify	555	0	3642	2,021	36.5
Qsort	212	0	52	11	0.3
Topopt	0	0	N/A	0	0.0

In Table 2.2 the "ideal" data on locking patterns as taken from the traces is given. "Total Held" refers to the total number of cycles during the run during which a lock was held. The "% of Time" column is the percent of the run-time for which a lock was held ("Total Held" divided by "Work Cycles"). The column "Nested locks" refers to when a lock is locked while another lock, an outer lock, is already held by the same processor. These nested locks occur only in Presto programs when threads are removed from the

run queue. The outer lock is the scheduler lock and the inner lock is the thread queue lock. The inner one is sometimes held when the outer one is not held. However, this does not often happen. So, as far as lock contention is concerned, the inner lock is not usually a source of contention since it is rarely acquired without the outer lock having been acquired first. Therefore, for Grav, Pdsa and FullConn I need only consider the total number of locks minus the number of nested locks.

Most of the time, the locks are held for only a few hundred cycles. An exception is Pverify where the locks are held for a very long time.

2.3.4 Locking Implementations

There are two main goals of the study. First to see how effective the more advanced lock implementations really are, and secondly, to examine the pattern of locking in real programs. So, an advanced locking algorithm which will hopefully have minimal impact on the locking pattern and the processors doing useful work is needed, as is a more mundane algorithm. Therefore, I did my studies using two locking implementations: queuing locks and Test&Test&Set (see Figure 2.2).

For the queuing lock implementation on my simulator, when a processor wants to acquire a lock, a memory access is made. When the result of that access returns to the processor, the processor checks to see whether or not it has acquired the lock. If it has, it enters the critical section. Otherwise the processor stalls. When the lock is released, the processor releasing the lock does a memory access. Also, a cache to cache transfer is done if another processor is waiting for the lock.

The simulator scheme is not a true representation of queuing locks. In an exact queuing lock implementation, there would be an additional memory access in the phase when a processor gets on the queue for the lock. In addition, in the Illinois protocol that I am using, there would be an additional memory access after the release of the lock if a processor is waiting and there would be no cache to cache transfer. I used the slightly

more efficient scheme to minimize the implementation constraints.[4] With the results that I have generated, I believe that the two missing bus transactions have no impact on the validity of my results as applied to queuing locks.

2.4 Results

2.4.1 Queuing Lock Implementation

In this section I present the results that I found when I examined the behavior of the benchmarks using my approximation to queuing locks. I am mostly interested in the number of lock transfers and the number of processors waiting at the time of the transfer and the impact of these numbers on the degradation in processor utilization. I would also like to see which "ideal" statistics (e.g., number of lock pairs, how long locks are held) are good predictors of lock contention.

The basic statistics that I collected from the simulation of the benchmarks are summarized in Table 2.3. The run-time for Topopt is somewhat skewed compared to the numbers in Table 2.1 because there is one processor whose trace has a much higher average cycle per instruction (CPI) although it has the same length in references. For a given processor, its utilization is calculated as the number of work cycles for that processor divided by the total number of cycles until that processor completed simulating its trace. The processor utilization given in the tables is the average of each processor's utilization.

As shown in Table 2.3, the programs with the largest number of lock acquisitions (cf. Table 2.2), Grav and Pdsa, have the lowest processor utilization and the highest percentage of stalls due to waiting for a lock. It is of course not a surprise that the program with the most locks shows this behavior. What is interesting is that although the locks are held for almost the shortest period of time, on the average, they still end up causing by far the most contention. Furthermore, the amount of time the program executes in "locked mode" is not a significant factor either since Pverify's percentage

[4] Idiosyncrasies of the simulator made it difficult to simulate queuing locks more completely.

Table 2.3: Benchmark Runtime Statistics: Queuing Lock Implementation
The stall causes are the percent of stalls caused by that event.

Program	run-time (cycles)	Processor Utilization (%)	Stall Causes	
			cache miss	lock wait
Grav	9,228,727	32.6	3.2	96.5
Pdsa	7,105,257	40.3	10.2	89.5
FullConn	4,407,243	95.5	86.9	10.2
Pverify	5,997,346	96.1	100.0	0.0
Qsort	4,307,966	67.8	99.7	0.3
Topopt	13,818,998	99.3	100.0	0.0

of time in that mode is much greater than Pdsa's and Pverify's processor utilization is greatly superior to Pdsa's.

Some more details on the lock contention are shown in Table 2.4, including the number of lock transfers, i.e., the number of times a lock is released by a processor and acquired by another waiting processor. The level of contention for the lock is reflected in the number of waiters. This number is the average of the number of processors still waiting for the lock after it has been released by one processor and acquired by the first waiter. For Grav and Pdsa this number is slightly over half the number of processors. This is extremely heavy contention since, by comparison, a barrier would yield a number less than half the number of processors. By contrast, Pverify almost never has two processors wanting the lock simultaneously and FullConn does not have this happen a significant number of times.

In summary, the best predictor for programs with high lock contention that can be found through the "ideal" analysis is the number of lock acquisitions. Grav and Pdsa have the most lock acquisitions by a factor of greater than four and a half and have the worst behavior. This is what we would expect. As the number of lock acquisitions increases, there is a greater chance of there being contention for the lock. This is true even though Grav and Pdsa on the average hold locks for some of the shortest periods of

Table 2.4: Lock Contention Statistics: Queuing Lock Implementation
Lock holding times are averages in cycles.

Program	Time held	Transfer Lock Stats		
		Number	Waiters at Transfer	Time held
Grav	211	28,725	5.19	336
Pdsa	203	16,977	6.18	356
FullConn	389	344	0.40	844
Pverify	3766	28	0.00	41
Qsort	120	180	0.89	174

time. The percentage of time that locks are held is not a predictor of locking behavior. For example, in the "ideal" analysis of Pverify, locks are held for a percentage of time almost as long as for Grav. However, the effect of lock contention is minimal on the run-time of Pverify.

Some of the results might be influenced by the Presto programming environment. The two programs with the most lock contention are Presto programs. It is clear from FullConn that writing a program in Presto does not automatically mean that there will be a lot of lock contention. However, even if programming in Presto introduces many stall cycles due to lock contention, it still is a useful tool. The Presto programming environment is based on user-level threads instead of system level threads and thereby the losses due to lock contention might be recouped since there are no traps to the kernel [7]. A version of the Grav and Pdsa programs written directly in C, and trying to attain the same level of parallelism, would have this overhead to contend with. However, the lock contention, while present at the system level, would not show up in the application level traces. It is also interesting to note that the Presto program with the best lock behavior, FullConn, was written by someone familiar with the inner workings of Presto as part of his Ph.D. dissertation [91]. Grav and Pdsa, with their poorer behavior, were written as part of a ten week seminar.

2.4.2 Importance of the Lock Implementation

Tables 2.5 and 2.6 show the same statistics as Tables 2.3 and 2.4, but with locks implemented using the Test&Test&Set primitive. The differences between the two sets of tables indicate the importance of an efficient lock implementation.

Table 2.5: Benchmark Runtime Statistics: Test&Test&Set
The stall causes are the percent of stalls caused by that event.

Program	run-time (cycles)	Processor Utilization (%)	Stall Causes	
			cache miss	lock wait
Grav	9,970,129	30.7	3.6	96.4
Pdsa	7,680,362	37.9	9.8	90.2
FullConn	4,416,720	94.6	88.0	12.0
Pverify	5,996,557	96.1	99.1	0.9
Qsort	4,310,056	67.6	99.4	0.6

Table 2.6: Lock Contention Statistics: Test&Test&Set
Lock holding times are averages in cycles.

Program	Time held	Transfer Lock Stats		
		Number	Waiters at Transfer	Time held
Grav	217	28,742	5.16	343
Pdsa	208	16,882	6.21	363
FullConn	409	338	0.30	978
Pverify	3767	36	0.03	48
Qsort	130	166	0.61	181

Naturally, there won't be any major difference in the behavior and run-time of programs with low lock acquisitions, i.e., the last three programs. The interesting figures are for the two programs that exhibit high lock contention. As can be seen, Grav takes 8.0% longer when using Test&Test&Set than when using queuing locks and Pdsa takes

8.1% longer. In looking at Table 2.6 we see that locks in those two programs are not held significantly longer than when using queuing locks nor are there more processors waiting at the transferring locks. The run-time increase is mainly due to three factors. The first is the time needed to transfer the lock. When there are many processors waiting for a lock, it takes approximately 21-25 cycles for any processor to get the lock versus 1.2-1.5 cycles for the queuing lock scheme that I simulated. Multiplying the difference by the number of lock transfers gives us an idea of the magnitude of the increase due to this factor. In Grav this results in 78% of the increase in run-time and in Pdsa 77%. The second factor is the length of time that locks are held. Even though in the Test&Test&Set implementation transferring locks are held only five to six cycles longer, this ends up being an important difference. In the case of a transferring lock, this cost of five to six cycles is paid by a waiting processor for each processor that precedes it in acquiring the lock. So, for the two programs under consideration, this extra cost contributes approximately thirty cycles to each time that a processor waits for a transferring lock. This causes 17% of the increased run-time for both Grav and Pdsa. The third factor is that with a high number of waiting processors, there is a concomitant flurry of Test-and-Sets that causes increased bus contention. This flurry occurs after a processor has acquired the lock and does not appear to affect that processor's behavior since the lock holding times do not significantly change. A plausible explanation is that the processor with the lock must already have the working set for the critical section in its cache (for a detailed description of what happens when the lock is released in Test&Test&Set see Anderson [10]). However, the increased bus contention does have an overall impact. The bus utilization for Grav doubled when using Test&Test&Set and for Pdsa increased 40%, and this slows down even those processors that do not want the lock. We can surmise that the remainder of the increase in execution time, 5% for Grav and 6% for Pdsa, is caused by this factor.

In summary, an efficient lock implementation can make a noticeable difference in the execution times of programs with high lock contention. It appears that the decrease in

execution time is mostly due to the reduction in time for lock acquisition *per se*, and secondarily to a decrease in lock holding time and to lower bus contention, allowing better progress of those processors that are not waiting for the lock. However, if there is high lock contention during much of the program's execution, then the serialization effect of critical sections will be a much greater factor in the performance of the program than the lock algorithm will be and the programmer should pay greater attention to improving the synchronization pattern of the program.

2.4.3 Conclusions

Previous studies of locking algorithms have focused on artificial benchmarks. Although these benchmarks have been useful in the development of new and more efficient algorithms for locking, they do not necessarily show how much performance is lost by real programs with less efficient locking schemes. In this study I used real parallel programs for my comparisons. These programs show that although there is a noticeable performance improvement from using more advanced lock algorithms, the gain is not as great as might be expected. This is because when the level of lock contention is high enough to produce a benefit for a scheme like queuing locks, the sequential nature of the critical section has a greater impact on the program's run-time (cf. Amdahl's Law). This effect overwhelms most of the benefit that the better lock algorithm provides. The synchronization pattern of the program is what really needs careful attention.

In my study, the two high contention programs, Pdsa and Grav, had high contention locks that were not held for very long, roughly 350 cycles. However, Qsort's locks were held for 52 cycles on the average in the ideal case and 130 in practice, and in some cases for as few as 12 cycles. But the lack of lock contention in that program is too low to allow us to reach any conclusion about lock algorithms. It does demonstrate that some programs can be written with locks held for extremely short periods of time. However, if an application is written such that the inherently sequential part of the program does not dominate and there is a high level of contention for a lock, then it may be the case

that advanced lock algorithms will be crucial for that program's performance. However, my benchmarks did not include any programs with such behavior. An example of a program like that would be one where fine-grained tasks are frequently added to and removed from a shared work queue. If the enqueue and dequeue operations were merely the modification of pointers to add and remove queue elements, then we would have a program where the lock algorithm might produce a significant performance gain.

Chapter 3

Relaxed Models of Memory Consistency

Memory latency, which can be defined as the time from when a processor requests a word of memory that is not in its cache until that word arrives at the processor, is becoming an ever greater performance bottleneck in modern computer systems. The disparity between processor and memory speed (cycle time) is increasing, and main memory cannot keep up. Moreoever, memory latency is an even greater problem for multiprocessors and this effect stems from several causes. If a bus is used for the interconnection, the multiple processors cause increased contention for the bus. This severely limits the scalability of bus-based systems. If buses are not used, the distance from the processor to memory will increase for large-scale multiprocessors due to the interconnection network. This is especially a problem in systems using a multistage interconnection network (MIN), where the transit time to memory increases as more processors are added. Even if distributed shared-memory on a mesh-type interconnection is used, then some memory will be further away as more processors are added. If shared data is being cached, as is most likely the case in a high performance machine, then there is the issue of maintaining cache coherence. Regardless of whether coherence is enforced by the hardware or the software, maintaining coherence results in additional messages on the interconnect. There are

messages to invalidate cache lines and the invalidations will result in more cache misses (although not every line invalidated will end up causing a miss). If an update protocol is used instead, there may be fewer messages, but the messages will be longer because they must contain the new data. These additional messages do not exist in the uniprocessor version of the program and cause higher levels of contention on the interconnect, and therefore higher latency. So, it is important to find ways to deal with memory latency.

Various ideas have been proposed, and some used, to deal with the cost of memory latency on cache misses in multiprocessors. The two main techniques are (i) to reduce the number of cache misses, and (ii) to reduce the penalty incurred when there is a miss. Examples of the former include making caches larger, prefetching into the cache [14, 55, 66], and write generate [94]. Examples of the latter technique include two level caches [15], hierarchies of caches [35], cache-only memory architectures [60] and memory in the switches of a multistage interconnection network (MIN) [79]. Also, if a distributed shared-memory system is used so that a portion of the memory is local to each processor (as opposed to a dance hall structure), the cost of some of the misses will be reduced [6, 83]. Prefetching data into the cache can be said to fall into this category as well as the first category. If a line has been prefetched but has not yet arrived from memory when one of the words is referenced, at least the full latency will not be observed. Another example of reducing the penalty on a cache miss is to do other useful work while waiting for a memory request. For example, if there is cheap context switching, then when there is a cache miss the processor can switch to another context in just a few cycles [6, 93].

Despite all these techniques, the degrading effects of high memory latency can be quite significant. Additional solutions to the problem are still needed. A promising way to deal with high memory latency is to have a system that implements a relaxed model of memory consistency.[1] When using a relaxed model of memory consistency, the processor need not stall every time there is a miss. In addition, relaxed models of consistency permit advanced architectural features to be used which cannot be used in

[1] I will use the terms relaxed model(s) of memory consistency, relaxed memory models, relaxed models of consistency and relaxed models to refer to the same concept.

normal multiprocessors and which may improve performance.

In the remainder of this thesis I will discuss and present results relating to relaxed models of consistency. I will explain in detail what they are and what their architectural and software ramifications are. Then I will present some performance studies showing the effects of using relaxed models.

It should be noted that most of the techniques mentioned in the above paragraphs are not exclusive. They may be combined for additional benefits [57].

3.1 Why Relaxed Consistency

There a number of architectural features that can be used in a uniprocessor, but which cannot be used without caution in a normal multiprocessor. The use of such features relates to the memory model that the machine implements. A memory model is simply a set of guarantees that the hardware makes to the software about how memory accesses will be done, or in other words, what the programmer can assume about memory operations. Normally the programmer assumes, without even realizing it, a model like sequential consistency (SC) [70]:

> [A 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.[2]

Thus, in a sequentially consistent system the end result of the program must be as if each shared access has completed before the next shared access is started and these accesses are executed in program order. This is what one would naturally expect. If this is not the case, then certain algorithms, such as Dekker's Algorithm for mutual exclusion [40] may not work correctly.

[2] In this and subsequent definitions operation means a shared memory access.

Initially flag = 0	
$\underline{P_1}$	$\underline{P_2}$
WriteBuffer()	repeat
flag = 1	until (flag == 1)
	ReadBuffer()

Figure 3.1: Violating sequential consistency - Bounded Buffers

An example of the perils of non-sequential consistency that is simpler than Dekker's Algorithm can be seen in the producer-consumer code in Figure 3.1. If the system on which this code is executing is sequentially consistent, then after P_2 reads $flag$ as having a value of one, then it will always be reading the new values in the buffer. However, consider a non-sequentially consistent system with multiple memory modules. If the buffer is in a memory module which is a hot-spot, then the write of the buffer may be delayed. But if $flag$ is in a memory module that is not busy, then the write of $flag$ will be done sooner. P_2 will read flag as having a value of one before the writes into the buffer have completed and it will read the old values from the buffer that are in its cache.

Initially X = Y = 0	
$\underline{P_1}$	$\underline{P_2}$
X = 1	Y = 1
if (Y == 0) kill P_2	if (X == 0) kill P_1

Figure 3.2: Violating sequential consistency

A more subtle example of the perils from non-sequentially consistent systems is in Figure 3.2 [4]. In a sequentially consistent system the end result of this code fragment is that either process P_1 or P_2 will be killed or neither will be, but not both. However,

if sequential consistency is violated, then both could be killed. This could happen if the writes to X and Y are put into write buffers which can then be bypassed by reads. So, even if a process' own data dependencies are observed, as they were in both cases, sequential consistency is still vital for the working of algorithms with interprocess data dependencies [68].

It should be noted that the definition of sequential consistency says that the end result of the execution has to be the same as if SC's constraints were obeyed. That means that operations can be executed out of order if the result does not change. Guaranteeing an SC execution for a uniprocessor in the presence of out of order instruction completion is very simple. Before a context switch can be done, all outstanding shared memory references must complete. As long as the process' own dependencies are observed, the result will be an SC execution. Memory accesses may be done out of order because there is no other executing process that is affected by these operations at that time and problems such as those shown in Figures 3.1 and 3.2 cannot occur. Therefore, in a uniprocessor loads can bypass stores in a write buffer and the cache can be lockup-free [67], and the system can still guarantee that all executions are sequentially consistent. However, in a multiprocessor, since there are processes executing in parallel, accesses may be seen out of order if care is not taken. Various ideas have been proposed to enhance the ability of multiprocessors to guarantee sequential consistency while allowing greater flexibility in the ordering of memory accesses and dealing with memory latency [3, 51], but there could be greater performance gains if the constraints of sequential consistency were loosened.

One way to loosen the constraints of sequential consistency is to change the memory model. As shown in the examples in Figures 3.1 and 3.2, the reason that SC is important is that any normal read or write can be used for synchronization (that is, communicating state information between different processes). When synchronizing processes, it is usually necessary that SC be maintained. But if SC is needed because of synchronization, why not just enforce it at synchronization points? Most memory accesses are normal loads and stores, which can be done in an efficient fashion. A logical conclusion is to

single out the scarcer synchronization operations and execute them in an SC way, but keep the common case fast. And this is exactly what Dubois, Scheurich and Briggs suggested with *weak ordering*.

3.2 Weak Ordering

Weak ordering (WO) was the first proposed relaxed memory model. It is defined [41]:

> In a multiprocessor system, storage accesses are weakly ordered if:
>
> 1. accesses to global synchronizing variables are strongly ordered and if
> 2. no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed and if
> 3. no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

For our purposes, strongly ordered and sequentially consistent accesses can be considered as synonymous (distinctions between the two are examined in Adve and Hill [3]) and strong ordering will not be referred to again. Performed is defined formally by Dubois et al. [41] and in Appendix A. Basically it means that a STORE is performed when the value stored by the processor executing the instruction can be seen by all other processors, and a LOAD is performed when the value to be returned by a LOAD has been set and cannot be changed. The definition of weak ordering implicitly assumes that uniprocessor data and control dependencies are obeyed. Unless noted otherwise, this is also assumed in the definitions of all other relaxed models.

If hardware obeys this definition of weak ordering, then what was said at the end of Section 3.1 is exactly true. Synchronization operations are sequentially consistent with one another, but when executing the normal accesses between such points in the program, the constraints on the hardware are relaxed. In terms of implementations, this means that between synchronization points accesses can be done in any order that obeys the uniprocessor dependencies. But when a synchronization point is reached, all

outstanding accesses must complete before the synchronization access is started. Then the synchronization must complete totally before any new shared accesses are started.

If this new model of weak ordering is used, then programmers are not using their intuitive model of sequential consistency. Changing memory models makes the job of writing parallel programs, an already difficult task, even more difficult. However, it has been shown that hardware can appear sequentially consistent to programmers with minimal constraint on their programming style [4], thereby allowing the programmer to continue using the familiar model of sequential consistency, while the hardware implements a more efficient, relaxed model of consistency. The significant restrictions in this case are that the programs must contain no data races and that all synchronization operations be visible to the hardware (so that the hardware knows when synchronization is being done, and hence when to enforce rules two and three of the definition). The former is not much of a restriction since a data race is usually an error on a sequentially consistent machine as well. The latter condition is not a significant restriction since most multiprocessors already provide hardware support for synchronization (e.g., Test-and-Set), and these primitives are what the programmer usually uses for synchronization (as opposed to normal loads and stores as in Dekker's Algorithm). If weak ordering were used in the example given in Figure 3.2, then X and Y would need to be marked as special locations used for synchronization or the operations to read and write them would need to be marked as special synchronization operations. As said above, most synchronization in real programs uses some special hardware synchronization and therefore would not need to be modified. This is true even for synchronization for DoAcross loops and barriers.[3]

In a weakly ordered system, the only restriction on the access order of normal memory references is that imposed by the program's own dependencies. All other memory references may be executed in any order. Therefore an order that maximizes system performance may be used. This includes letting reads and instruction fetches take prece-

[3]There are various barrier algorithms [61] which are commonly used and can be implemented using normal loads and stores. These would need to be modified in some way for weakly ordered systems.

dence over writes, prefetching data, issuing instructions out of order or letting them complete out of order, and delaying cache coherence invalidation signals until after the write to a line in the cache has been performed.

Although weak ordering was the first weak memory model proposed, it is not the only one. The others that I will describe are processor consistency and release consistency. I will also talk about the more informal model of lazy release consistency. In addition, there are the more software oriented views of DRF0, DRF1 and PLpc which I will also discuss. However, a more detailed explanation of synchronization is needed first.

3.3 Synchronization - Acquires and Releases

Synchronization is used to coordinate the work done by different parts of a parallel program. It is usually done using barriers, locks and unlocks for mutual exclusion, or synchronizing reads and writes in a DoAcross loop or a producer-consumer type problem. The semantics of synchronization operations are such that they can be classified as one of two different types. Those operations which are signaling to other processors that something has been done (usually that data has been written) are called *release* operations (also called export) and operations that delay a processor until a signal from another processor has been received are called *acquire* operations (also called import). Release operations are releasing data to be seen by other processors, or exporting it to them. For example, a synchronizing write, such as in a source statement of a DoAcross loop, is a release. The signal is the writing into a flag to indicate that a certain operation has completed. The write is a signal that something has been done, and the processor doing the write does not really care exactly when the write completes. [4] Acquire operations are acquiring or importing the latest version of some data before proceeding. For example, the synchronizing read in a DoAcross loop is an acquire. It is in a loop checking

[4] In fact, there is no reason at all for the processor writing the flag to wait for the writing of the flag to complete. It should be able to start the write and then continue onward. A release is basically an asynchronous synchronization. Acquires are synchronous though.

to see if the value has been written into the flag. The processor doing the read is explicitly waiting until it gets the signal that it can continue. Among usual synchronization operations, locks are considered acquire operations, unlocks are releases; synchronizing reads are acquires, and synchronizing writes are releases. A barrier is both an acquire and a release, with the release implicitly being done first. Note that this taxonomy of synchronization operations is not necessary for weak ordering, since all synchronizations are treated in the same manner.

3.4 Processor, Release, and Lazy Release Consistency

3.4.1 Processor Consistency

The definition of *processor consistency* (PC) is different from most of the other models in that it does not mention synchronization at all. It was originally defined by Goodman [54] (see Appendix A). However, Gharachorloo et al.'s [53] definition is the one now commonly used:

1. before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed, and
2. before a STORE is allowed to perform with respect to any other processor, all previous accesses (LOADs and STOREs) must be performed.

This basically means that loads are allowed to bypass pending stores, which can be very useful in hiding write latency [50]. In a PC system, acquire operations are treated as loads and releases as stores. Synchronizations are not treated any differently than other shared accesses.

Under constraints that are similar to those of Adve and Hill's, Gharachorloo et al. [49, 53] showed that programmers can obtain sequentially consistent executions on a system implementing PC.

Processor consistency is not clearly less or more restrictive than weak ordering. Each model permits some ordering that the other does not permit. In a PC system, when a

release is done, the processor continues executing even if the release has not completed, because a release is treated as a write, which can be bypassed by loads. In contrast, in a weakly ordered system, when a synchronization point is reached, the synchronization must complete totally before any new shared accesses may be issued. However, normal accesses may be done in any order, whereas in a PC system the only optimization is that reads may bypass writes. Clearly it would be nice to combine both models. This has been done and the result is *release consistency.*

3.4.2 Release Consistency

Release consistency (RC) is an attempt to combine the features of weak ordering and processor consistency. This results in an even more relaxed model of consistency which stalls under fewer conditions. It is defined [53]:

1. before an ordinary LOAD or STORE is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and

2. before a release access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and

3. special accesses are processor consistent with respect to one another.

Special accesses can be considered to be just synchronization accesses. Distinctions are examined in Gharachorloo et al. [49, 53].

RC differs from weak ordering in a number of ways. First, at an acquire operation it is no longer required that all outstanding shared memory accesses complete before the acquire is started. The purpose of the acquire is to make sure the issuing processor does not try to access data that is due to be updated by another processor. The fact that accesses of the processor issuing the acquire may be outstanding is unimportant. Second, a release operation may be outstanding when an acquire is issued (if they were operations on the same memory location, then there would be a uniprocessor data dependence, which is assumed to be observed). The release, a signal to other processors, has nothing

to do with the acquire, a reading of a signal from another processor. Third, normal shared accesses after a release operation are not delayed if the release has not completed. Again, the release is a signal to other processors, so its lack of completion should not cause the processor issuing it to stall. So, by taking advantage of the semantics of synchronization, RC is able to relax the constraints of weak ordering and PC even further.

As in weak ordering, there are constraints on software, which, if obeyed by the program, guarantee that all executions of the program on RC hardware will be sequentially consistent. The constraints are similar to those for weak ordering. Programs must be properly labeled (PL). The exact definition of PL and the proof that PL programs will have SC executions on RC hardware appears in Gharachorloo et al. [49, 53]. However, being PL and being data race free are very similar constraints on a program.

RC as I have described it here is sometimes called RCpc. That is, release consistent with processor consistent synchronization operations. A variation, RCsc, where the synchronization operations are only sequentially consistent has been considered [53]. In that model a release operation must be performed before an acquire operation can be issued (this means the second difference between RC and WO mentioned in the above paragraph would not exist). This distinction matters because of certain properties that have been proven about RCsc but not RCpc [1] and is discussed further in Section 3.5 and Chapter 5. Note that this difference will only matter when the frequency of synchronization or the time to perform a release are so high that release operations are frequently outstanding at the time an acquire would be issued. Unless the notation RCpc or RCsc is used instead of just RC or release consistency, RCpc is to be assumed.

Although RC relaxes the constraints on the memory system greatly, more can be done.

3.4.3 Lazy Release Consistency

Keleher et al. [65] introduced the idea of lazy release consistency (LRC). The difference between RC and LRC is based upon the semantics of what an acquire and release mean to

a program. Keleher et al. pointed out that a processor that needs to access data that has been updated by another processor will always do an acquire to insure that it is seeing the updated data. There is no need for all accesses that appear in program order before a release to be globally performed at the time of a release. Rather, they do not need to be performed until a following acquire, and then only with respect to the processor doing the acquire (this basic idea also appeared in Adve and Hill [1] and Zucker [96]). The difficulty with LRC is that there is a transitive effect that must be obeyed. Consider the example in Figure 3.3. If P_1 writes some data while in the critical section, then when P_3 does the acquire associated with its lock of X, not only must all of P_2's writes from within the critical section be performed with respect to P_3, but so must all of P_1's writes. This aspect of LRC was noted in Keleher et al. [65] and a way to implement it in software in Munin [18, 28] was described, but a true memory model description was not given. It can be tricky to describe the transitive aspect of the definition in terms of hardware conditions, but without an actual memory model definition, we cannot prove that a properly labeled or data race free program will execute in a sequentially consistent fashion on a system implementing LRC. Intuitively it does seem like it would, since the updates are propagated to those processors that need the data. Although the transitive performing of accesses could be done in software in a system such as Munin, in hardware it is probably too complex to implement.

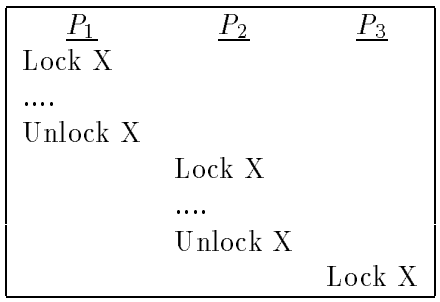


Figure 3.3: Transitive performing of accesses in LRC

3.5 Software Centric Views: DRF0 and DRF1

All the models described up to this point have a basic framework in common. They all describe how the hardware is to operate and then sometimes describe programming rules for programmers, which if obeyed by a program, guarantee sequentially consistent executions for that program on the hardware obeying the consistency model. For example, weak ordering gives a precise definition of what happens at synchronization points and when memory accesses need to be performed. The architect can design any system that obeys WO's rules, but the hardware *must* obey them. An architect might be able to design a system which is not WO, but was functionally equivalent. That is, if a program were run on this system, it would produce the exact same result as if it were run on a WO system. The programmer would not be able to write a program that would run differently on this new system than on the WO system. But this new system would not be considered WO because it did not obey the rules of WO (such a situation is very possible for most programs in which we are interested. Consider LRC versus RC). This problem is dealt with by Adve and Hill.

Adve and Hill [4] take a different approach to the idea of relaxed memory models. They first describe what conditions the software must obey (e.g., no data races) and then say that any hardware that guarantees sequentially consistent executions to any software obeying their conditions conforms to their memory model. This is a different concept of memory model, but it makes more sense in some respects. It gives the architect greater freedom, and makes life easier for the programmer. The programmer can still think about sequential consistency and does not even need to be told what the actual hardware implementation is, and the architect can implement any system that provides SC executions to programs that obey the conditions on the software.

Originally Adve and Hill said [4]:

> Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.

They proposed a synchronization model, data-race-free-0 (DRF0). It is defined as:

1. all synchronization operations are recognizable by the hardware and each accesses exactly one memory location, and

2. for any execution on the idealized system (where all memory accesses are executed atomically and in program order), all conflicting accesses are ordered by the happens-before relation corresponding to the execution.

The happens-before relation is defined formally in Adve and Hill [4]. Basically it means that there can be no data races in the program (hence the name data-race-free). Later in the paper they define a set of sufficient conditions which if met by the hardware, mean that the hardware is weakly ordered with respect to DRF0 (see Appendix B.1).

The end result of this is that the programmer is told to write a program without data races and with hardware visible synchronization operations accessing only one memory location, which is not a problem if system provided synchronization routines are used. The architect can then implement any system that provides SC executions to programs that obey DRF0. The hardware does not need to obey the conditions repeated in Appendix B.1 as they are sufficient conditions, but not necessary ones. So, the architects are able to define their own set of conditions. For example, in Zucker [96] I suggested that condition five of the sufficient conditions for DRF0 be changed so that the writes of P_i do not need to be globally performed, but rather, performed with respect to any processor doing a later synchronization operation on that location before that processor does that later synchronization operation.

Adve and Hill [1] put forth another software-oriented view of relaxed memory models. In this paper they defined a model, data-race-free-1 (DRF1):

> Hardware obeys DRF1 if and only if the result of every execution of a data-race-free program on the hardware can be obtained by an execution of the program on SC hardware.

This is an even simpler definition than that for DRF0. There are no longer restrictions on the type of synchronization. Adve and Hill [5] provide a set of sufficient conditions, which I will refer to as the DRF1 Condition (The condition is in Appendix B.2; This condition, its terminology and components will be explained in more detail in Chapter 5). If a system obeys the DRF1 Condition, it is proof that the system obeys DRF1. The term system is used because there are certain situations where it is necessary to consider more than just the hardware, but also guarantees that the compiler will make as well (see Chapter 5).

The first condition of the DRF1 Condition says that synchronization operations must be sequentially consistent. They may not be processor consistent. So, RCpc does not obey DRF1, but RCsc does. This will also be discussed at greater length in Chapter 5.

It is interesting to note that in some ways sequential consistency has more in common with the software-oriented view than the hardware-centric ones. The definition of SC does not say that hardware must obey certain rules. It simply says that the end result of any execution must be the same as if the hardware had obeyed certain constraints. Whether it actually obeys those constraints or not is for the architect to choose, as long as the end result of all executions is the same as if they were obeyed. The programmer again does not need to know how it is implemented.

3.6 Effects of Relaxed Consistency

The differences between most of the relaxed models involve how they treat synchronization operations. In general, normal reads and writes between synchronization points are allowed to be performed and issued in any order that obeys the uniprocessor dependencies. This allows numerous architectural features to be used that cannot be used in sequentially consistent multiprocessors. In the next two chapters I will present several studies considering the benefits of using various of these architectural features in multiprocessors.

The feature most commonly mentioned in the literature when discussing advantages

of relaxed models is the bypassing of writes by reads (the whole basis of processor consistency). Although several writes in a row can be buffered in an SC system, when a read is about to be issued, the writes must first complete before the read can be done. But in a relaxed system the read is allowed to bypass the writes in the write buffer. This is considered in Section 4.2.

Non-blocking loads can also be used more fully in a relaxed system. Non-blocking loads mean that the processor does not stall just because there is a cache miss on a read request. The processor can still issue and execute instructions until an attempt is made to use the destination register of the load request. If the load and the first register-register operation that uses the destination register are far enough separated in the instruction stream, then the processor will not have to stall. Non-blocking loads can be used in an SC system. But while the load is outstanding, the processor cannot issue any other memory references. It can issue other references though in a relaxed system and take better advantage of the non-blocking loads. In both cases, to take full advantage of non-blocking loads compiler support is needed. This will be considered in Section 4.3.

The logical progression from non-blocking loads, whose efficient use depends upon a static scheduling of instructions for hiding read latency, is to dynamic instruction scheduling [62], where instructions that are ready to execute are *issued* out of order, not just performed out of order. Again, dynamic scheduling can be done in an SC system. But to gain the full benefit, as in non-blocking loads, a system implementing a relaxed memory model is needed. I will not present any studies of relaxed memory models and dynamic scheduling in this thesis, but such a study has been done [52].

The way cache coherence is maintained can also be changed when relaxed models are used. In a typical system with hardware enforced cache coherence using an invalidate-based protocol [12], before a write can be done to a word belonging to a line in a shared state, an invalidation signal must first be sent out and acknowledged (although the acknowledgment may be implicit, as in a bus-based system) before the write is actually

done or the processor can continue. In a relaxed system the write can be done and the processor may continue executing before the invalidation signal is sent out, let alone, acknowledged. This option is considered in Section 4.2.

Software controlled cache coherence (SCCC) becomes far more practical with relaxed memory models than under sequential consistency. However, it is not easily implemented on all systems. Under weak ordering or release consistency it is still difficult to implement SCCC. It can be done though on a system obeying DRF1. This is examined in Chapter 5.

Chapter 4

Performance Evaluation of Relaxed Models

4.1 Introduction

In the previous chapter I discussed several aspects of relaxed models of memory consistency. What we really want to know though, is what, if any, are the performance benefits of using these relaxed models. The purpose of this chapter and the next is to report on performance studies of relaxed models. In this chapter I will present a trace-driven simulation of a weakly ordered system [16]. Then I will discuss some of the limitations of that study, and finally present a more detailed instruction level simulation of various relaxed memory models [97, 98].

4.2 Trace Driven Simulation Study

Although it appears intuitive that a relaxed model of memory consistency will provide a performance gain, we would like to know how much of a gain it will really provide. To that end I have done a study to see what performance advantage there is from some features allowed under a weakly ordered system.

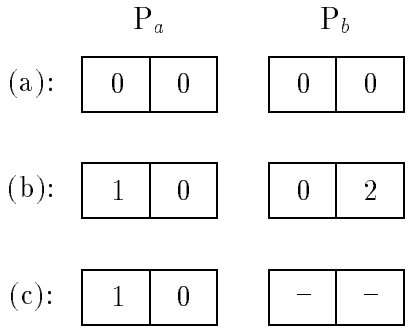
The study was done in the same manner as described in Section 2.3. It uses the same traces, and for the baseline sequentially consistent system, the same simulated architecture. For the weakly ordered system the bypassing of accesses in the buffers between the caches and the bus is simulated. Any memory reference whose miss in the cache would cause the processor to stall may be placed at the front of the bus access buffer for that processor. Such references are loads and instruction fetches (this could also be done in a sequentially consistent system for the instruction fetch). The accesses that can be bypassed are writes, write backs, and invalidation signals. An access is considered performed when it is in the cache and any writes to it have completed. Once it is in the cache, its value is visible to the other processors due to the hardware enforced cache coherence.

When a synchronization operation occurs, the processor stalls until all the memory accesses currently buffered or underway complete. This also means that all the lines for the cache misses have returned and been installed in the cache. Once this has been completed, then the synchronization variable may be accessed.

In my model there is neither prefetching, nor out of order issue or completion of instructions, nor delaying of invalidation signals. Prefetching into the cache can be done in both sequentially consistent and weakly ordered models since the hardware enforced cache coherence protocol will assure correctness. Therefore its performance impact would be essentially the same for both models. Moreover, any prefetching strategy, either hardware or software based, is fairly elaborate and cannot be included given the trace information. Similarly, modeling out of order instruction issue or completion is not possible given the traces used. No instruction types are given, only the number of cycles for the instruction to complete.

The delaying of invalidation signals would work if a single-word line size were simulated, since both processors would be writing to the same word. The first invalidation to be done (since this is a bus-based system, one must be done before the other) should not only invalidate the other line but also cancel the second invalidation. It would not

matter which value survives the invalidations, since if there are no synchronization operations to order them, either is a valid value according to the memory model. However, this delaying is not possible with multi-word lines. If two processors both have a write hit on a line in a shared state, then whichever invalidation is done first will cause the cancellation of the other invalidation and convert it into a write miss. It will also invalidate this other processor's copy of the line. If the two writes are directed to different words in the same line (false sharing) and are executed before either invalidation, then the first invalidation would cause the value written by the other processor to be lost (cf. Figure 4.1). Since the model uses a multi-word line size, it does not allow the delay of the invalidation and when there is a write hit on a line in a shared state, the write is not done until the invalidation completes successfully.



In (a) the caches of both processors have the same values for both words in the cache block. In (b) they have each written to a different word in the block before the invalidation signal has been sent out. In (c) P_a's signal has been sent out and invalidated this line in other caches including P_b's. The value of 2 which P_b had written, is now lost.

Figure 4.1: Write before invalidation problem

4.2.1 Weak Ordering Results

Tables 4.1 and 4.2 summarize the results from running my benchmarks assuming a weakly ordered memory system. The column difference (%) shows the percent decrease

Table 4.1: Weak Ordering Runtime Statistics
Difference is the difference between these numbers and those in Table 2.3.

Program	run-time (cycles)	Processor Util. (%)	Difference (%)	Write Hit (%)
Grav	9,221,719	32.6	0.08	90.9
Pdsa	7,084,835	40.5	0.29	90.5
FullConn	4,381,518	95.5	0.31	91.6
Pverify	5,987,383	96.3	0.17	98.4
Qsort	4,306,958	67.9	0.02	99.0
Topopt	13,796,023	99.4	0.17	97.4

Table 4.2: Weak Ordering Lock Contention Statistics
Differences with results from Table 2.4 are not significant.

Program	Time held	Transfer Lock Stats		
		Number	Waiters at Transfer	Time held
Grav	211	28,468	5.25	338
Pdsa	203	16,919	6.26	357
FullConn	390	373	0.34	857
Pverify	3758	21	0.00	40
Qsort	100	151	1.05	155

in execution time for the weak memory model. As can be seen, in all cases it is less than 1%, and sometimes much less than that. Recall that in the system modeled, the only benefit of weak ordering is bypassing. This will usually result in a performance improvement when there is a write miss that would otherwise cause a processor stall. Grav and Pdsa, the first two programs in table 4.1, have cache write hit ratios lower than the other programs but weak ordering has no significant effect because the very high lock contention overwhelms any possible benefit. In the other cases, the programs with lower write hit ratios show the best improvements but these improvements are still minuscule. Qsort's improvement is surprisingly low given that its write hit ratio is almost the same

as that of Pverify. But its processor utilization is low because of a large number of read misses due to the magnitude of the data set being sorted. These read misses dominate (recall table 2.3) and are much more frequent than write misses since the reads almost always precede the exchanges (writes) of the same lines.

Given these results it does not seem as if weak ordering is worth implementing in a shared-bus system such as the one simulated. One cannot forget that there would be extra costs involved in implementing weak ordering. Bypassing must be possible in the memory access buffer. Since the cache must be able to handle requests while a request is outstanding, the caches must be lockup-free [67] thus increasing the complexity of the cache and possibly increasing the cache cycle time. It is conceivable that this cost could more than outweigh the benefit obtained by reducing the number of stalls on write misses.

Although weak ordering does not appear worthwhile in this architecture, it does not mean that it is not worth investigating. If the miss penalty were greater, e.g., because the memory latency is much higher as in a multistage interconnection based system, or the number of writes to memory increased (as in the case of a write-through cache), then the benefit would be greater and might justify the cost [50]. Delaying of various cache coherence signals does not appear to be as promising. These signals are generated when there is a write hit on a line in a shared state. Since this is a write, it does not cause a stall and I have already demonstrated that there are not enough writes to make a significant difference.

4.2.2 Limitations

There are various limitations with this study. It was done in the context of shared-bus multiprocessor and with arguably low memory latency. In a system with a multistage or mesh interconnection the memory latencies would be higher. In addition, the number of processors was fairly low. Most importantly though is that the trace-driven nature of the study is limiting. Without knowing what instructions are being executed, the

simulator must block for every load access. There is no information in the trace about which registers are used by which instructions and no way to determine if a given load need not cause an immediate stall if it results in a cache miss. It would also be useful to simulate a greater number of processors. All these issues will be addressed in the next section.

4.3 Instruction-Level Simulation Study

In order to address some of the deficiencies of the trace-driven study, I did an instruction-level simulation. In this study I considered two variations of a sequentially consistent system and several systems using relaxed consistency. In particular, I assessed the impact of various architectural enhancements that the different models allow. In addition, instruction-level simulation allowed me to consider features such as non-blocking loads, which cannot be studied with trace-driven simulations. An interesting by-product of instruction-level simulation was the ability to verify that programs did execute correctly since they actually produce output, which was very important in debugging certain implementations.

4.3.1 Methodology

Base Architecture

The architecture simulated is a typical "dance-hall" architecture (see Figure 4.2). It consists of a number of processors, each with a local memory for private data and its own cache, and a number of memory modules for global memory that contain shared data. Processors and global memory are connected via two identical Omega networks, one for the processor requests to memory and one for memory's responses. Cache coherence is enforced by a full directory scheme [29].

I used the Cerberus instruction-level simulator [24] for my simulation studies. The processor it simulates is a RISC processor similar to the Ridge 32 [84]. The caches are

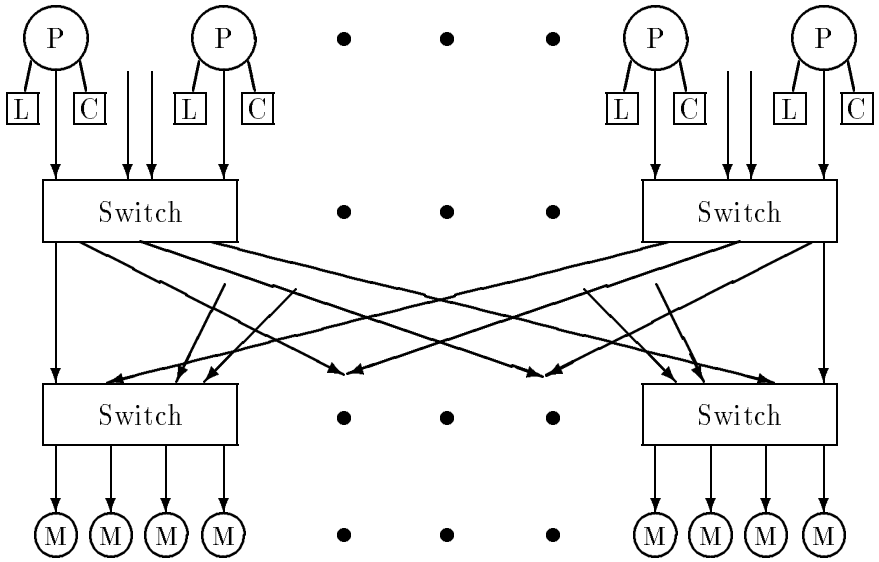


Figure 4.2: Model Architecture

two-way set associative and use a write-back write-allocate policy; I experimented with a range of cache and line sizes. The caches are only for shared data. It is assumed that there are no instruction cache misses and that the private data can be accessed from local memory. The simulator is configured to use 4×4 switches in the interconnection networks. There is a four element buffer between the processor and the request network as well as between the memory and the response network. The memory latency (no contention) for the fetching of the first word of the line is 18 cycles for 16 processors and 20 cycles for 32 processors. The memory latency is independent of the line size due to the pipelined nature of the network and of the memory accesses. However, the length of time that the response network and the memory are busy is proportional to the line size. Each stage of the network takes one cycle for every eight bytes. The memory access takes seven cycles to initiate, after which the first word is available and put onto the network. The rest of the line follows and the memory is kept busy for as many cycles

as there are words in the line. If the requested line is dirty in other caches or clean in other caches and requested for write, then the latency is correspondingly higher due to the need to send coherence messages and wait for the corresponding acknowledgments.

Model Implementations

Five system types were simulated. Two are sequentially consistent, two are weakly ordered, and one is release consistent. A summary is given in Table 4.3.

Table 4.3: Summary of Implementation Features

System	Major Features
SC1	sequentially consistent, non-blocking loads
SC2	SC1 + hardware directed non-binding prefetch at stalls
WO1	SC1 + hw visible synchronization operations, no stalls on memory access while outstanding references
WO2	WO1 + bypassing of pending messages by loads
RC	WO1 + no stalling while a release completes no stalling for outstanding accesses at an acquire

Sequentially Consistent 1 - SC1 SC1, a sequentially consistent implementation, is the baseline system. The programmer has complete freedom for the scheduling of inter-processor communication and synchronization, but the hardware does not allow a subsequent memory access[1] if one access is already outstanding. However, one optimization is allowed, namely non-blocking loads. Therefore, the processor only stalls if it tries to read from a register that is the destination of an uncompleted load or it attempts another memory access. Since there can be only one outstanding reference at a time, lockup-free caches [67] are not necessary; however, some form of register interlock or scoreboarding is needed for the correctness of register-register operations. In the case of a cache hit, the loads are delayed loads, and the value is not available for four cycles

[1] All references to memory accesses and locations mean shared memory.

(The simulator initially forced me to use a (arguably long) delay of four cycles. I have since repeated the experiments with a smaller delay and found the absolute benefits of the models to be roughly the same. See Section 4.3.3). The processor also does not stall in the case of a write miss. The value to be written is sent to the cache where the write is performed when the line is received from memory. At the same time the source register may be overwritten by a register-register operation.

Sequentially Consistent 2 - SC2 A more aggressive version of SC that does not change the programmer's view of the system was also simulated. Non-binding prefetches on stalls, as suggested by Gharachorloo et al. [51], were added to SC1. The processor still stalls when there is an outstanding memory reference and another memory reference is about to be made. But a request for this second access is nonetheless sent to the cache. If the access to the corresponding cache line results in a miss, the line is prefetched into the cache. Note that this prefetch is non-binding; the contents of the prefetched line are still visible to the cache coherence mechanism since no value has been loaded into a register in the case of a load, nor has a new value been written into the line in the case of a store. The performance advantage brought upon by this prefetch is that the miss rate, or at the very least the perceived memory latency, is reduced since the memory accesses are pipelined.

For SC2 the cache is more complicated. The cache controller must be able to handle a prefetch request while there is an outstanding reference. SC1 only needed to distinguish between coherence requests and a line returning from memory, of which there could be only one. SC2 must be able to determine which of its (two) outstanding requests is returning from memory, and whether or not the processor can now be unstalled.

Weakly Ordered 1 - WO1 The first relaxed memory model I simulated obeys the definition for weak ordering given in Section 3.2. With a WO1 implementation the programmer is limited to synchronization that is visible to the hardware. So WO1 provides special synchronization instructions in the *lock* and *barrier* routines. The programmer

also has the option of using a SYNC instruction to indicate a synchronization point, such as when writing into a shared flag. With respect to the hardware, the processor must now stall at each synchronization point until all outstanding memory references complete. Except for that restriction, memory accesses are allowed to complete in any order as long as the data and control dependencies are observed.

With WO1 several memory references can be outstanding at a time and hence the cache must be lockup-free. Information on each outstanding reference must be maintained in a miss information/status holding register (MSHR) [67] (five MSHR's are simulated). The MSHR's will cause the processor to stall when it makes a reference which must be delayed due to data dependency requirements. In a WO1-like system, the non-blocking loads can be especially useful since several loads can be initiated successively, even in the absence of intervening register-register operations. The overlap of the memory accesses can hide some of their latency (Both read and write latency are hidden here. In Section 4.3.3 blocking loads are used to determine how much of the hidden latency is due to reads and how much is due to writes).

The hardware costs of WO1 can be significant. The cache must be lockup-free; its MSHR's must be (associatively) consulted when requests return from memory, for cache coherence messages, and each time the processor issues a reference (in order to check for an outstanding reference to that line and for data dependencies). Hardware recognizable synchronization instructions are required. The processor must stall when it encounters such an instruction if there are any outstanding references and be able to restart when all references have completed. This requires the use of a counter and an associated stall/restart mechanism.

Weakly Ordered 2 - WO2 WO2 is the same as WO1 except that it allows some bypassing of stores by loads. Bypassing does not change the programmer's view of the hardware. It is done because the completion of a load is more important than the completion of a store since the loaded value generally will be required sooner than the global performing of the store. Bypassing is limited to the buffer that serves as

an interface between the processor and the interconnection network. Bypassing is not simulated in the Omega network switches since the speed at which they must operate seems to prevent such an optimization (moreover, since the requests would come from different processors, it is not certain that the priority argument still holds).

Bypassing introduces additional hardware complexity. The cache must treat load and store misses differently when sending them to the network. Loads must be sent to the head of the buffer while the stores must be able to enter in the normal FIFO manner. Unlike a system with blocking loads, where only one load can be outstanding, the capability of dealing with multiple loads bypassing stores must exist. In my implementation, a bypassing load could bypass a load that is at the front of the buffer. This could be prevented if an entry could be inserted into the buffer after the last load but before the first store, or if there were two FIFO buffers, one for loads and one for stores and a priority scheme always favoring the former. I will discuss in Section 4.3.2 why the results are still valid despite the simple, but slightly flawed, implementation.

Release Consistent - RC In the release consistent system the acquire and release synchronization operations are treated differently, thus requiring the programmer to be aware of the type of synchronization needed. However, the need to make this distinction will not usually be a problem if system-provided synchronization routines are used. Under the simulated RC, when an acquire operation is attempted, the processor stalls until the acquire completes. It does not matter if there are any outstanding memory accesses.

There are several possible situations which can occur when a release operation is encountered in the instruction stream. In all cases though, the processor continues executing past that point in the program. The release operation can be issued if there are no outstanding accesses. However, the release cannot proceed immediately if the locations it operates on are not in the cache. In that case the request to do the release is sent to the cache which will delay issuing the operation until the necessary lines arrive from memory. If there are outstanding references when a processor encounters a release operation, then a special bit is set in all the valid MSHR entries and a counter is set

equal to the number of such entries. Whenever an outstanding reference is resolved, this bit in the corresponding MSHR entry is checked. If it is set, the counter is decremented. When the counter is decremented to zero, the pending release operation is issued. This scheme requires an extra bit in the MSHRs, a counter for the number of references still outstanding from the time of the release operation (as opposed to a general counter of outstanding references as in WO1), and the ability to keep track of the pending release operation since the processor has continued executing past the location in the instruction stream where the release was encountered.

Benchmarks

The benchmark programs used to compare the various memory models are all written using PCP [23], a simple parallel extension to C. They were compiled to the simulator's machine code and linked using Cerberus's compiler. Due to compiler limitations there is no static allocation of private data. Any variable allocated globally is a shared variable and resides in a shared memory module. The only private data are variables declared in functions. Since the architecture supports non-blocking loads, the compiler attempts to schedule the code so that loads to registers are issued far enough ahead of their use so that the processor will hopefully not have to stall if there is a cache miss.

Table 4.4: Benchmark Statistics for SC1 for 16K and 64K caches
References are averages per processor and are in 1,000's.

Programs	References (1,000's)		Hit Rate (%) by line and cache size					
			16K cache			64K cache		
	Reads	Writes	8	16	64	8	16	64
Gauss	1074	314	64.3	80.9	94.4	95.9	97.4	98.8
Qsort	1171	310	70.2	73.8	81.8	73.2	76.0	82.3
Relax	2132	329	78.8	89.3	97.0	79.6	89.7	97.2
Psim	1827	319	88.4	88.6	90.7	89.5	89.6	91.3

The four benchmark programs are described below. Table 4.4 shows statistics on

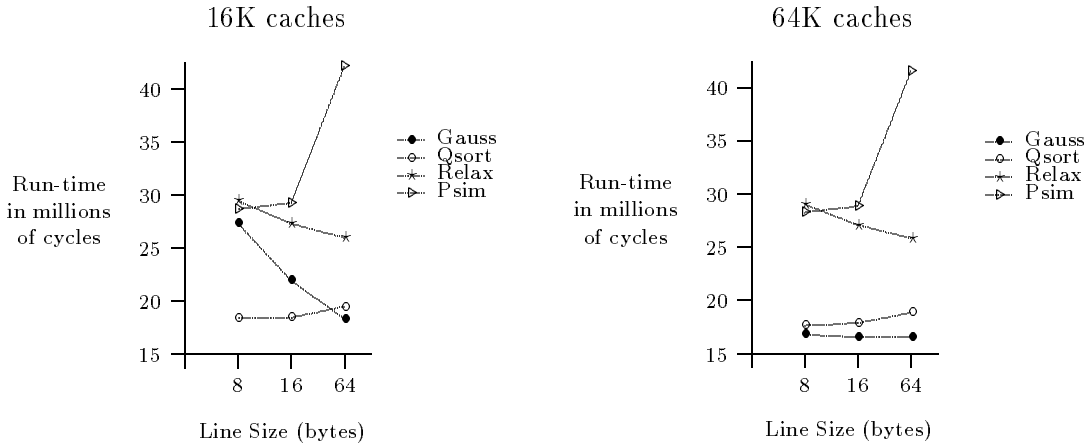


Figure 4.3: Performance by Line Size for SC1

the benchmarks for 16 processors for various cache (16K and 64K) and line sizes (8, 16 and 64 bytes) under SC1 and their run-times are given in Figure 4.3 (there are more detailed statistics in Appendix C). The exact number of memory references can change based upon the consistency model, and this can impact the hit rates as well. I observed a change in the number of references of up to 8% and in the hit rates of up to 2.8%.

Gauss *Gauss* [39] performs the gaussian elimination of a 250 $\times$ 250 matrix. This program exhibits a large amount of spatial locality, as can be seen from the 16K cache data in Table 4.4 and Figure 4.3. The larger line sizes result in large performance gains: the 64 byte line configuration is 50% faster than with 8 byte lines and almost 25% faster than with 16 byte lines. This gain is to be expected since the hit ratios increase from 64% for 8 byte lines to 81% for 16 bytes and to 94% for 64 bytes. However, when 64K caches are used the hit ratios are uniformly high and the run-times vary very little with the line sizes. Clearly the data set of each processor fits in a 64K cache, but not in a 16K one.

Qsort *Qsort* [63] executes a parallel quicksort of 500,000 integers. It is dynamically

scheduled, unlike the other benchmarks which are statically scheduled. Units of work are pushed onto and popped off of a shared stack and allocated to processors on a FCFS basis. Since any change in the architecture influences the relative rate of execution of each processor, the order in which tasks are pushed onto and popped off the stack can change and thereby affect the way the work is partitioned as well as the size of the partitions [63]. In one case, when the implementation changed from WO1 to WO2, the number of synchronization operations increased by a third, thereby demonstrating the effect of dynamic scheduling. This natural variability prevents me from reaching general conclusions from a single run of the benchmark. Because of the severe load on CPU time brought upon by instruction-level simulations, I did not repeat this experiment with different seeds and compute appropriate confidence intervals. I will not discuss the generally small variations in performance I observed, since I cannot isolate the likely causes due to the dynamic nature of the program.

As shown in Table 4.4, the hit ratios of *Qsort* are fairly low (69-81%). This is to be expected due to its sequential access pattern. Also, when partitioned among the 16 processors, each processor's data set is still too large to fit into even the 64K cache. As long as the cache capacity is too small, there is very little variation for a given line size in *Qsort*'s hit ratio between 16K caches and 64K caches. The line size does matter though. With 8 or 16 byte lines the run-times are roughly the same. However, the systems with 64 byte line caches are the slowest in spite of the higher hit rates.

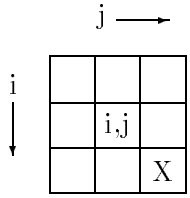
Initially I found the low write hit ratios curious. In the trace-driven study *Qsort* had a write hit ratio of close to 100%. This is because before values are swapped, they must be compared, which means that they are always read before being written, and hence, are already in the cache when they are written. However, in this study write hit ratios were around 70-80%. I finally realized that this was because of a change in the way write hits are counted due to the different cache coherence protocol. In the study reported in Chapter 2, a bus-based system was simulated. On that system, when a line is brought into the cache, even if it is just for read, it could be written once the other

lines were invalidated, a fairly simple operation on a bus-based system and one that I still counted as a write-hit. However, in this system, a line is requested for read or write. If it has been brought in for read and then there is a write, the current copy of the line is invalidated, and a new copy with write permission fetched. This is counted as a write miss, and the cause of the lower write hit ratio. It is not strictly necessary to invalidate the read-only line. However, it makes the protocol simpler at the expense of consuming some additional network bandwidth. Regardless, the request still needs to be sent to memory, and may need to wait for invalidations to be done.

This case does demonstrate the usefulness of a read with ownership request, a request which would not require the message to memory before proceeding with the write. However, the compiler would need to be able to recognize the situation where this would be useful. It is not automatically useful in *Qsort* as written. During the partition phase, which is done in parallel, a processor references every nth element. The locations are not strip-mined and therefore each processor references one word in a given cache line. [2] Since there is a great deal of sharing during this phase, it could be detrimental to performance to do a read with ownership, when in the end ownership may not be needed if there were no swapping by that processor. Later, when each processor is sorting its own partition, then a read with ownership is worthwhile.

Relax *Relax* is an iterative relaxation procedure using a nine point stencil over a 514 x 514 matrix. Its main computation consists of summing the values of nine grid points laid out in a square. Its access pattern can be described by using Figure 4.4. The nine point stencil is moved to the right as j is incremented. When i is incremented, the stencil is moved back to the first column, but one row further down. References to the top two rows in the stencil will always result in cache hits at this point since they were used while processing the previous row. Also, for a line size of eight bytes and a matrix

[2] For the shared-bus system for which the program was written [63], the overhead of the loop index computation was found to remove any benefit strip-mining would have otherwise generated. However, this may be different in the case of a higher latency system such as the one simulated here.



X - cache miss

```
for i = 2 to n do
  for j = 2 to n do
    <do relaxation>
```

Figure 4.4: Relax access pattern

of doubles (so each element takes up a single line), after the first two relaxations for this value of i, references to locations $(i+1,j-1)$ and $(i+1,j)$ will also be hits since they were referenced in the previous iteration as locations $(i+1,j)$ and $(i+1,j+1)$. The reference to $(i+1,j+1)$ will miss every time and only that reference will miss on any given relaxation (except at the beginning of a new row or when there is cross or self-interference [69]). There will also be one write miss per relaxation since *Relax* writes the result of each relaxation into a temporary matrix (if the line size is 16 bytes, then the read of location $(i+1,j+1)$ is a miss once every two relaxations as is the write of the result. For 64 byte lines, it is a miss once every eight relaxations for both the read and the write). Once all the processors have completed the relaxation phase for a given iteration, then the relaxed values are copied back into the main matrix. During this phase there will be one read miss and one write miss per inner loop iteration as each value must be read from the temporary matrix and written into the main matrix.

If the compiler produces code that loads a value and adds it into the sum before loading the next value, the processor will stall whenever it tries to add in a value whose loading is still pending because of a cache read miss, and this will happen regardless of the consistency model. However, after the write, there are a number of register-register operations which hide the latency of the memory access even in SC1. The Cerberus compiler does schedule all the loads before all the additions (this is discussed further in Section 4.3.2).

The computation pattern described above dominates *Relax* so much that the memory access pattern is extremely regular. The cache hit rates and the number of references are almost the same for all processors. In fact, the access patterns are so regular that the observed cache hit rates match up almost exactly with those predicted. As long as the cache is large enough to hold two rows or columns of the sub-matrix the processor is processing (whether it is a row or column depends upon the loop structure), then the hit rates can be calculated ahead of time very easily. Any differences are due to self or cross-interference [69] or coherence effects due to sharing at the edges of the sub-matrices (although given the right matrix size relative to the cache size, the self-interference of row $i-1$ and row i could be very significant).

As mentioned above, if the two previous rows do not fit in the cache, then the hit rate will drop significantly. However, as long as those two rows are still in the cache, the hit rate is independent of the size of the data set. Unlike *Gauss*, the data set size would need to be significantly larger for there to be a large change in the cache hit rate.

Psim *Psim* is the simulator of the multi-stage network that the simulator itself uses (Cerberus is written in PCP). It simulates a 64 processor network using 4×4 switches with each processor issuing 513 references. *Psim* differs from the other benchmarks in various ways. 70% of its cache misses are invalidation misses, which is indicative of a high level of sharing. This increases the traffic on the interconnection network. Also, *Psim*'s accesses to memory are not spread out evenly across the memory modules. The utilization of the modules varies by as much as a factor of six, which produces some minor hot spots. Both of these factors result in a much higher actual memory latency in *Psim*. The latency is proportional to the line size, and as seen in Figure 4.3, can have a major performance impact. The data set of *Psim* is fairly small and fits into a 16K cache, and hence its performance is fairly insensitive to the cache size. The hit rate (but not the run-time) is also insensitive to the line size. Finally, *Psim* has the highest synchronization rate of any of the benchmarks (over 80,000 synchronizations per processor).

4.3.2 Simulation Results

My experiments were instruction-level simulations of 16 processor systems for all benchmarks and of 32 processor systems for *Gauss* only because of time limitations. The simulations were run with cache line sizes of 8, 16 and 64 bytes and two cache sizes: 16K and 64K bytes. The results for 16 processors and 16K caches are shown in Figure 4.5, the results for 16 processors and 64K caches are shown in Figure 4.6, and the results for *Gauss* with 32 processors are shown in Figure 4.7. In these figures, each graph has plots for each of the three line sizes. The y-axes show the relative (percent) performance improvements of the various memory models over the SC1 system for that line size. Note that the scales of the y-axes are different for each benchmark.

In the following sections I discuss the results for each benchmark and then compare the results for the various memory models. This will allow me to draw some conclusions on the relative merits of each hardware addition.

Analysis of Benchmarks

Gauss The greatest performance improvement from using relaxed models is attained in the *Gauss* benchmark. In the case of 16K caches with 8 byte lines the systems implementing relaxed models show a performance gain of over 35%. With 16 byte lines the gain is about 20% and with 64 byte lines it is 8%. The greater gain for 8 byte lines is due to the lower hit ratios since cache misses provide the opportunity to draw benefits from the relaxed models. With the relaxed models there is also less of a gap between the performance of the different line sizes. Under WO1 there is only a 16% and 10% improvement for 64 byte and 16 byte lines over the 8 byte line versus the 50% and 25% mentioned earlier for SC1.

The case of the 64K cache is quite different. As mentioned above, the cache hit rates for 64K caches are quite high. Therefore we cannot expect much gain from the relaxed models; the benefits never reach 2%.

The 32 processor results show the same trends that are seen with 16 processors. The

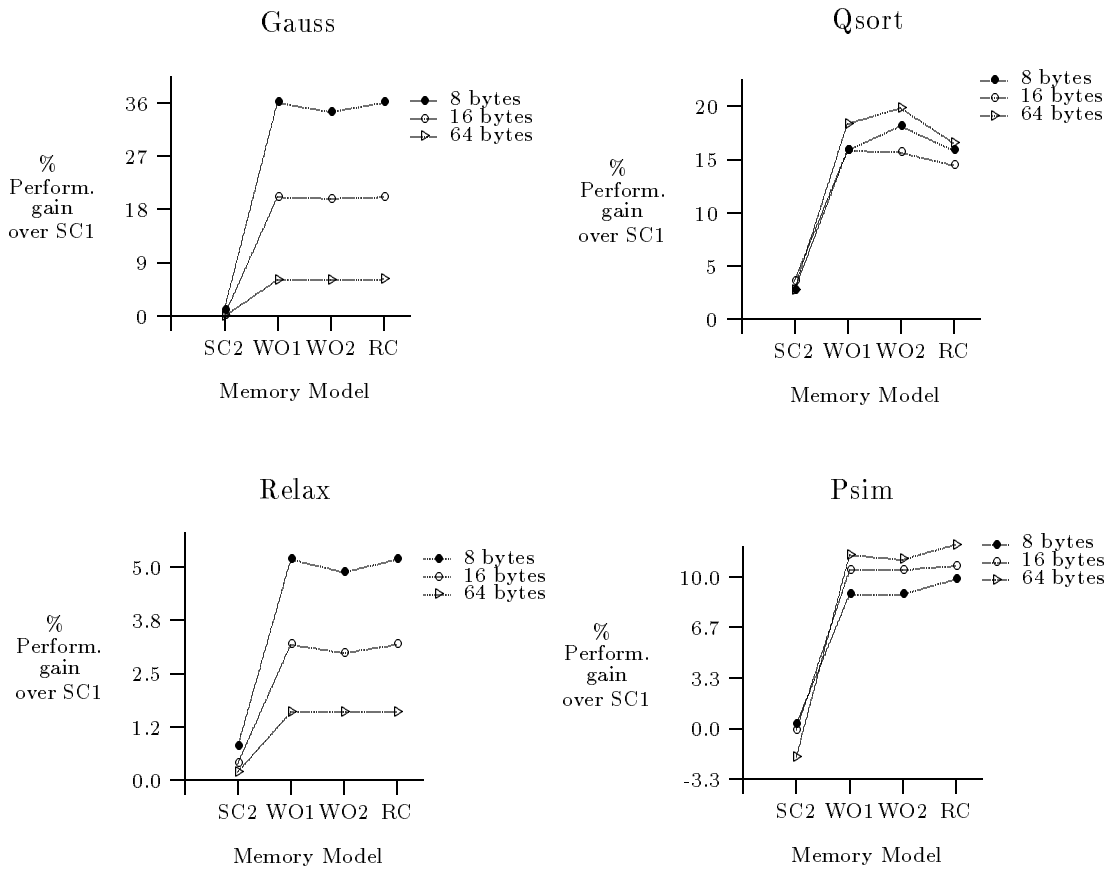


Figure 4.5: 16 processors, 16K caches

The performance improvement is relative to SC1 for that line size.
Note that the scale of the y axis in each graph is different.

data set still does not fit in the cache when 16K caches are used. However, the performance benefit for each line size is slightly higher than was observed with 16 processors. This is to be expected since the memory latency is higher due to the extra level needed in the network. Despite the slight increase in memory latency, the speed-up with 32 processors was good. The system was faster by 80-86% with one instance of 76%. The relaxed models showed a 1-4% better performance gain than SC1 did when using the same cache structure. The reason studies of WO2 for 32 processors were not done is explained in Section 4.3.2.

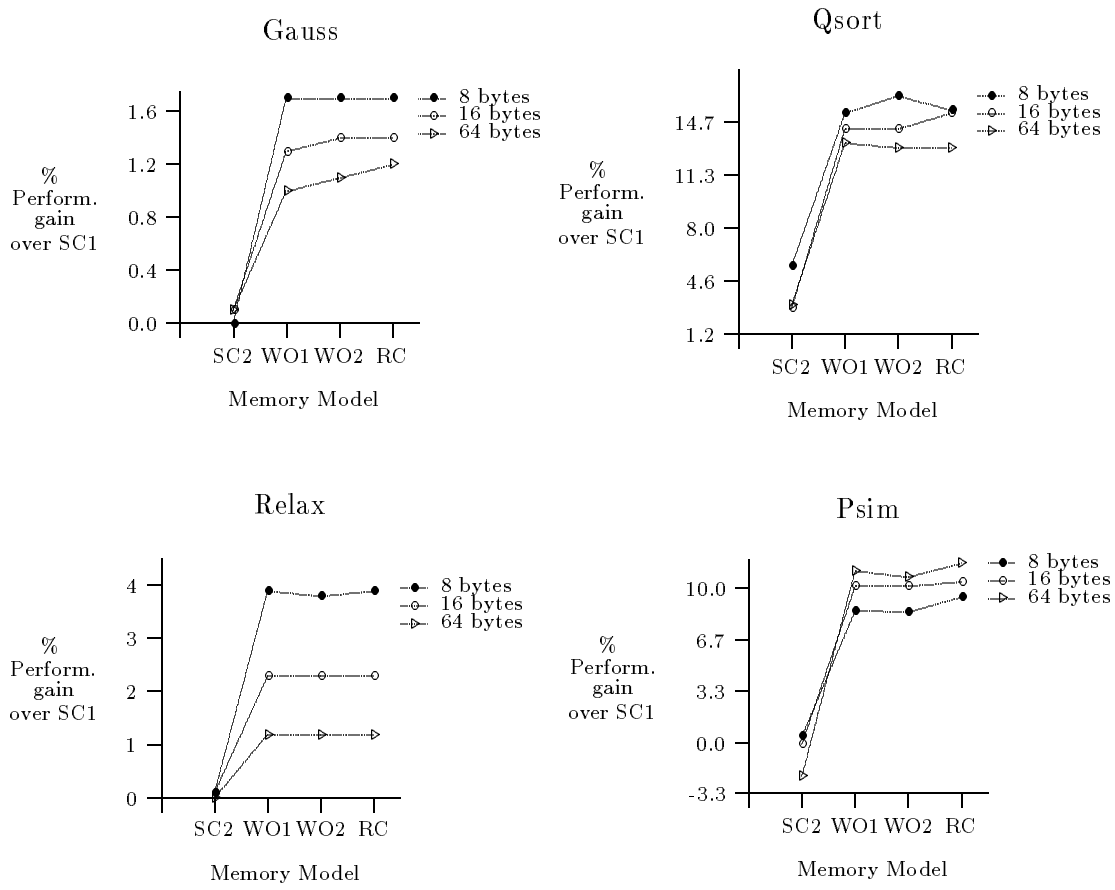


Figure 4.6: 16 processors, 64K caches

The performance improvement is relative to SC1 for that line size.
Note that the scale of the y axis in each graph is different.

One should not reach undue conclusions from the 64K cache experiment. Recall that the data set was of a size that made it amenable to be run on a simulator. The data set was a 250 $\times$ 250 matrix of doubles, which occupies only half a megabyte of memory. Gaussian elimination on such a matrix takes about 10 CPU seconds on a modern workstation. The problems that would be run on a real parallel processor will be much larger. The data sets will probably be too large for the cache and the hit ratios will be more like those recorded for the 16K cache.

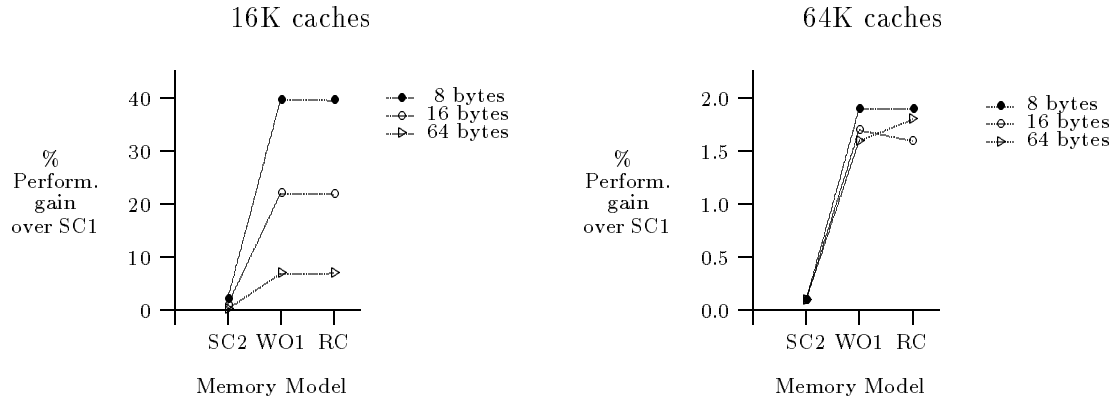


Figure 4.7: 32 processors, Gauss

The performance improvement is relative to SC1 for that line size.
Note that the scale of the y axis in each graph is different.

Qsort The performance of *Qsort* is significantly improved by the implementation of relaxed models. The benefits range from 13% to 18% and are realized for both cache sizes. These improvements stem from a moderately low hit rate (69-81% for both cache sizes since neither cache size has the capacity to store the processor's working set) accompanied by memory access patterns that allow overlapping. As mentioned in Section 4.3.1, the variability of results for the various line sizes and models is caused by dynamic scheduling effects.

Relax As can be seen in Figures 4.5 and 4.6, *Relax* obtains very little benefit from the relaxed models. The largest gain is 5%. Part of the reason for the lack of improvement is *Relax*'s high cache hit rate, from 79% to 98% depending mostly on the line size. Note that the 79% hit rate is comparable to *Qsort*'s and for that latter benchmark the relaxed models yield better improvements. The compounding reason for *Relax*'s low improvement is its access pattern. The relaxed models cannot hide much of the memory latency. As described in Section 4.3.1, most of it is already hidden in the case of a write miss (in all implementations) by the network access buffer and the register-register operations

following the write, or it cannot be hidden in the case of a read miss because an attempt to use the destination register occurs almost immediately after the read miss, and this results in a stall as described in Section 4.3.1.

This analysis of *Relax* made me realize that how the program is written or compiled for peak performance depends upon the memory model to be used. If the compiler can rearrange the code to schedule all the loads at the top of loop, then there may be some benefit from the relaxed models for *Relax* for 8 byte lines. The access that will cause the miss (see Section 4.3.1) should be done first among the nine loads, thereby allowing the main memory access to be overlapped with the loading of the other eight registers from cache. For larger lines with high hit rates (over 90%), the effect of this optimization, for this size data set, will probably be minimal. Even in SC1 the code can be rearranged to minimize the memory access penalty. All the loads should again be done at the top of the loop, but the one that causes the miss should be done last. Otherwise the processor will stall when attempting a load after the miss. While that line is being fetched from memory, the other eight values can be summed so that by the time the register whose value was coming from main memory is added into the sum, the line has arrived from memory (rearranging the additions can also be done for the relaxed models). The optimizer in the Cerberus compiler does reorganize the code so that all the loads are at the top of the loop. However, it is not smart enough to realize which load will miss in the cache and schedule the code accordingly. I conducted some experiments where I manually scheduled the code taking this effect into account. The results are described in Section 4.3.3.

Psim *Psim* has a moderate (8-10%) performance improvement from the relaxed models which is to be expected given its hit rate of approximately 90%. Since *Psim* has a much higher average memory latency due to its high level of sharing and skewed memory access distribution, it gets greater benefit than programs with similar hit rates. For SC2 with 64 byte lines *Psim* takes longer than for SC1. I surmise this is because of the extra contention on the networks due to the additional requests. The observed memory

latency does increase due to the increased traffic for 64 byte lines.

Relative Performance of the Consistency Models

Relaxed Model Benefits As a general rule, the relaxed models show a non-negligible performance improvement as long as the cache hit rates are not too high. If hit rates are in the high nineties, then the memory access times are less important and, as could be expected, the use of sequential consistency won't degrade performance. In the case of lower hit rates, the benefits of relaxed models can be mitigated if either a single non-blocking load or store (as in SC1) already allows a substantial overlap of memory access and computation, or, conversely, if a fetch for some value is almost immediately followed by its use. In the latter situation, it is likely that the code could be reorganized to take better advantage of the relaxed models, especially if the architecture supplies a sufficient number of registers. I further discuss the general benefits of the relaxed models in Section 4.4.2.

RC versus WO1 In all of the runs RC and WO1 performed in a similar manner. If there was any difference, RC's improvement over SC1 was slightly better. The largest instance, less than 1% better relative to SC1, occurred for *Psim*, the program with the most synchronizations. Recall that RC and WO1 differ only in the way acquires and releases are implemented. Under RC when a release is encountered the processor does not stall either for outstanding references or for the release to complete, whereas under WO1, the processor does stall for the references to complete and then for the release to complete. Also, for an acquire RC does not stall while outstanding memory accesses complete. Clearly neither of these events were frequent enough for the processor to spend a significant portion of its time stalling at releases in the WO1 system. From these benchmarks there is nothing to indicate that it is worthwhile implementing RC instead of WO1 if implementing RC would in any way be more costly or detrimental to the processor's cycle time.

WO2 - Bypassing Overall, it appears that bypassing of stores by loads is not worthwhile with lockup-free write-back caches and a pipelined interconnection network. Recall that WO2's implementation of bypassing is such that loads bypass waiting stores and waiting loads. On the average only one request was bypassed. In only one of the benchmarks, *Psim*, was the bypassed request usually a write and this benchmark had the most bypasses, 125,000 (roughly one every 200 cycles). However, as can be seen in Figures 4.5 and 4.6 this produced no difference in performance. In the other cases the number of bypasses of writes was insignificant. Note that even if a read bypasses a write, the odds are that the request is going to a different memory module. Bypassing does not help a read to get preferred treatment at the memory module where it is contending with requests from other processors. Therefore the only advantage is for the read to be on the network first, i.e., gain a cycle in the case of no or low contention. The slight drop in performance for *Gauss* and *Relax* was probably due to my implementation since those two benchmarks showed a small number of bypasses of reads by reads. *Qsort* showed both a performance gain and a loss which is mostly due to variability. It almost never had a write bypassed by a read. Since the WO2 experiment was sufficiently convincing, I do not see any reason to consider implementing bypassing in a release consistent system or studying it with 32 processors.

SC2 versus SC1 In general there is very little benefit in prefetching one line when a processor is stalled due to a cache miss. In fact, in one case (*Psim*, 64 byte lines) prefetching was detrimental because it increased the congestion in the network. As noted earlier, that benchmark has a high level of network traffic due to coherence effects and the network traffic was exacerbated by the large line size. SC2 causes a clustering of memory requests that further saturate the network, resulting in a net rise in the memory latency, the opposite of what was sought. The probable cause for the general lack of performance gain in SC2 is that the opportunity for it to be of any help, i.e., to have consecutive cache misses to different lines without an intervening register interlock causing a processor stall, is rare. This paucity of limited prefetches increases for programs

with good locality.

4.3.3 Architectural Variations and Results

Blocking Loads

With my implementations of the relaxed models, both read and write latencies are overlapped with computation. However, it cannot be determined how much of the hidden latency is due to reads and how much is due to writes. Therefore, I modified two of my implementations to use blocking loads and compared the results with those I already had. With blocking loads, when there is a read miss, the processor stalls until the requested line returns from memory. So none of the read latency is hidden. Since there was little variation among the sequentially consistent implementations and among the relaxed consistency ones, the only two implementations that I modified were SC1 and WO1. The versions with blocking loads are designated bSC1 and bWO1. Graphs showing the performance of SC1, bWO1 and WO1 relative to bSC1 are in Figures 4.8 and 4.9.

Non-blocking loads appear to have no significant effect for the two systems that are sequentially consistent. The performance of bSC1 and SC1 are basically the same (differences for *Qsort* are still due to its dynamic nature). However, non-blocking loads can still provide some benefit in such systems (see the section on manual scheduling of *Relax* below).

The effect of blocking loads in relaxed models varies with the benchmarks. In the case of *Relax* there is almost no difference between bSC1, SC1, and bWO1 regardless of line or cache size. However, there is a noticeable performance gain for WO1. Obviously the non-blocking loads are important here. It is clear from the structure of the *Relax* program that another memory access must occur shortly after loads that are missing; this causes a stall in SC1 and bWO1. Thus, almost all the latency hidden by WO1 for *Relax* is read latency.

In the case of *Psim*, a weakly ordered system with blocking loads provides 75-85% of the performance improvement that is obtained with non-blocking loads. This shows

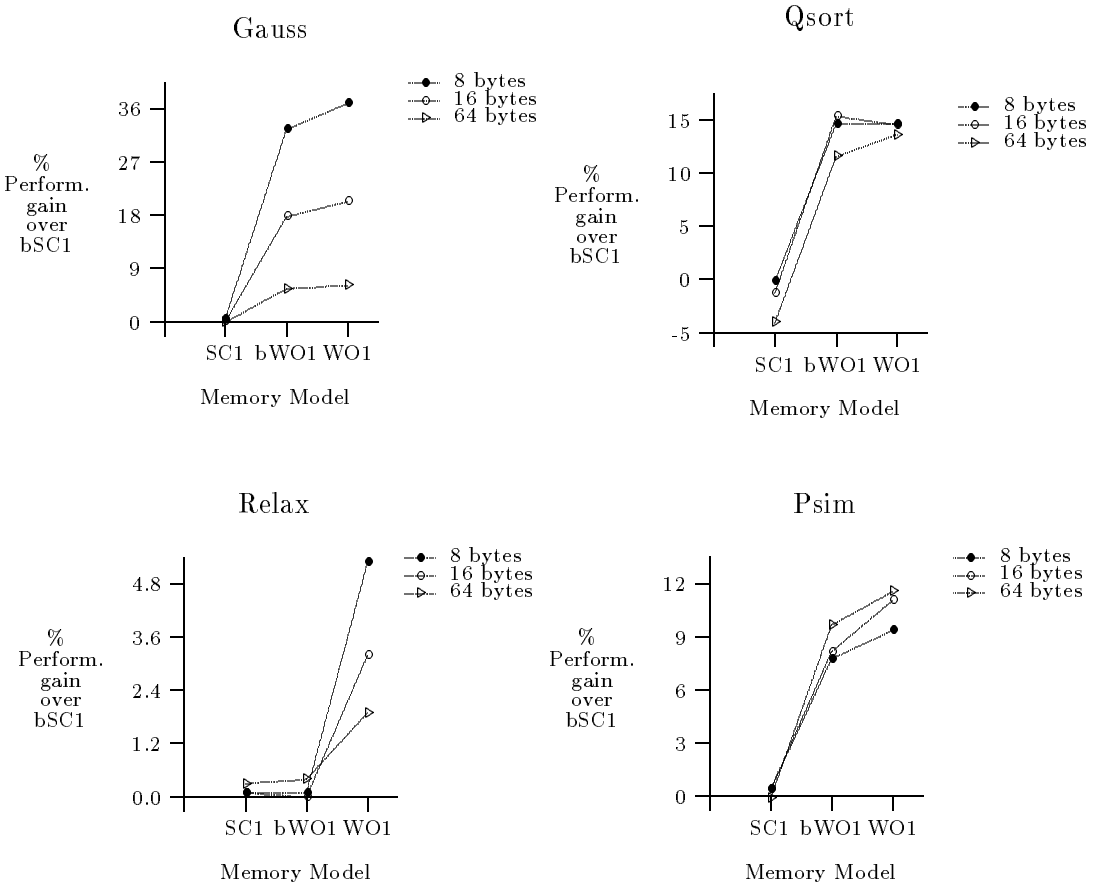


Figure 4.8: 16 processors, 16K caches, blocking loads

The performance improvement is relative to bSC1 for that line size.
Note that the scale of the y axis in each graph is different.

that although most of the latency being hidden is write latency, there is definitely a noticeable amount of read latency which is hidden as well.

For *Gauss* with 16K caches it is mostly write latency which is overlapped with computation. However, the scale of the y axis in Figure 4.8 can be misleading. There would be a noticeable loss of performance in some cases if blocking loads were used. In the case of 64K caches there appears to be a great deal of variability in *Gauss'* behavior (cf. Figure 4.9). However, the differences are actually so small as to be unimportant.

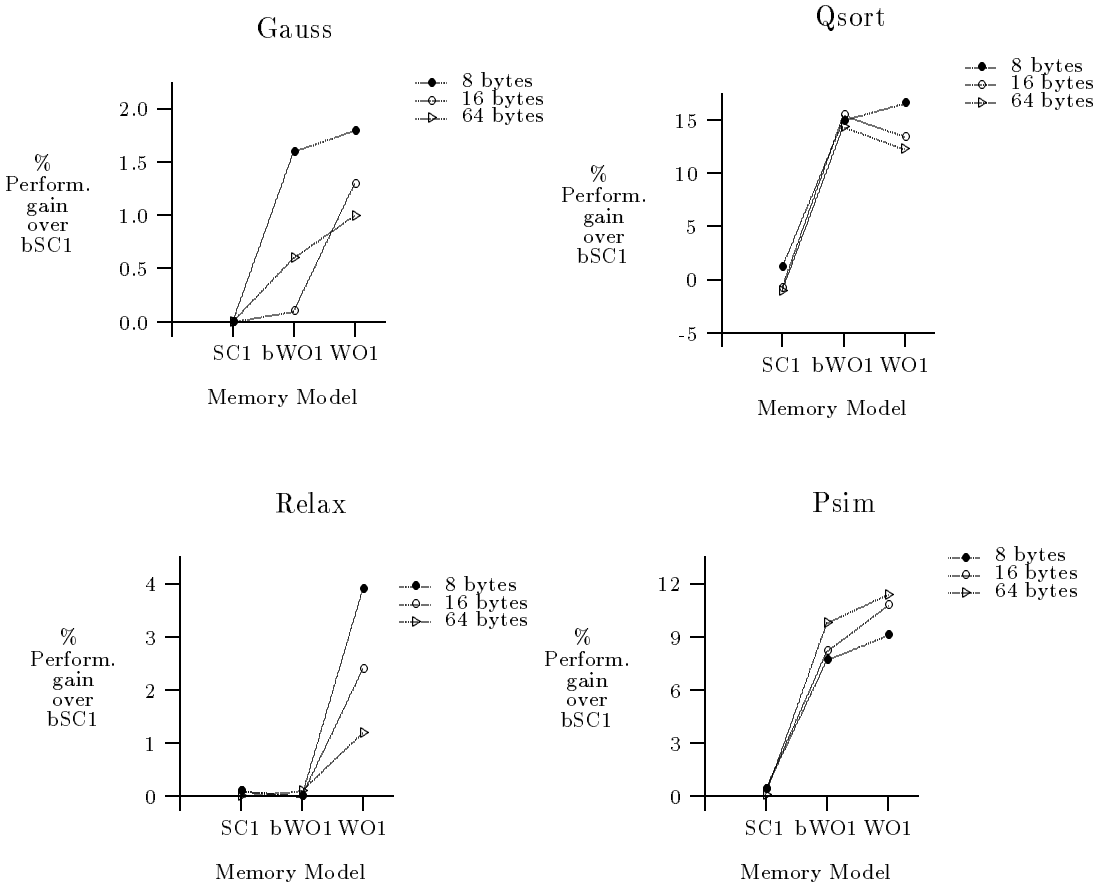


Figure 4.9: 16 processors, 64K caches, blocking loads

The performance improvement is relative to bSC1 for that line size.
Note that the scale of the y axis in each graph is different.

Hand Scheduled Relax Code

In Section 4.3.2 I described how simply moving all the loads to before all the floating point adds, as the compiler did, would not automatically improve the effect of using relaxed models of memory consistency as much as possible. The order of the loads is important. In order to test the importance of the scheduling of the loads, I manually scheduled the code in a more efficient order, with one schedule for the SC systems and one for the WO systems. As I described in Section 4.3.2, I took into account the knowledge

of which load (if any) would miss in the cache and scheduled the loads and additions so that there would be the maximum amount of time between the load and operations dependent upon the load (in the SC case this also meant guaranteeing that there were no other memory access between those instructions). I also manually scheduled the code in such a way to produce a deliberately bad schedule given the knowledge of which load would miss. Given the compiler's lack of knowledge of the architecture, these schedules were equally likely to have been produced.

Figure 4.10 shows the relative performance of the codes manually scheduled for good and bad performance compared to the optimizer's default schedule. There is a noticeable difference (up to 8%) in performance, and the range between the best and worst schedules is even greater (about 10%). This shows that although rules on how to write programs that execute correctly for systems implementing relaxed consistency (i.e., no data races) have been proven correct, gaining maximum performance on such systems is still an open question. Clearly, how programs are optimally compiled for such systems may be different than for a sequentially consistent system (manual scheduling of the code will be impractical most of the time). For a program like *Relax* a compiler can produce the same optimizations which I produced by hand [26]. However, *Relax* is a very regular program, i.e., one which is relatively easy to analyze. Although there is some research in this area [32], there is clearly a need for more.

Two Cycle Load and Branch Delays

Due to limitations of the simulator, my initial studies used a load and branch delay of four cycles. It can be argued that this is overly long (although superpipelined machines may have long delays as well). I duplicated the studies of SC1 and WO1 with load and branch delays of two cycles (this includes the load delay for loads of private data). The performance of the systems with this new parameter is shown in Tables 4.5 through 4.8. The results show a great deal of variability. There are some cases where the relative and absolute performance gains for two cycle delays are greater, some cases where the results

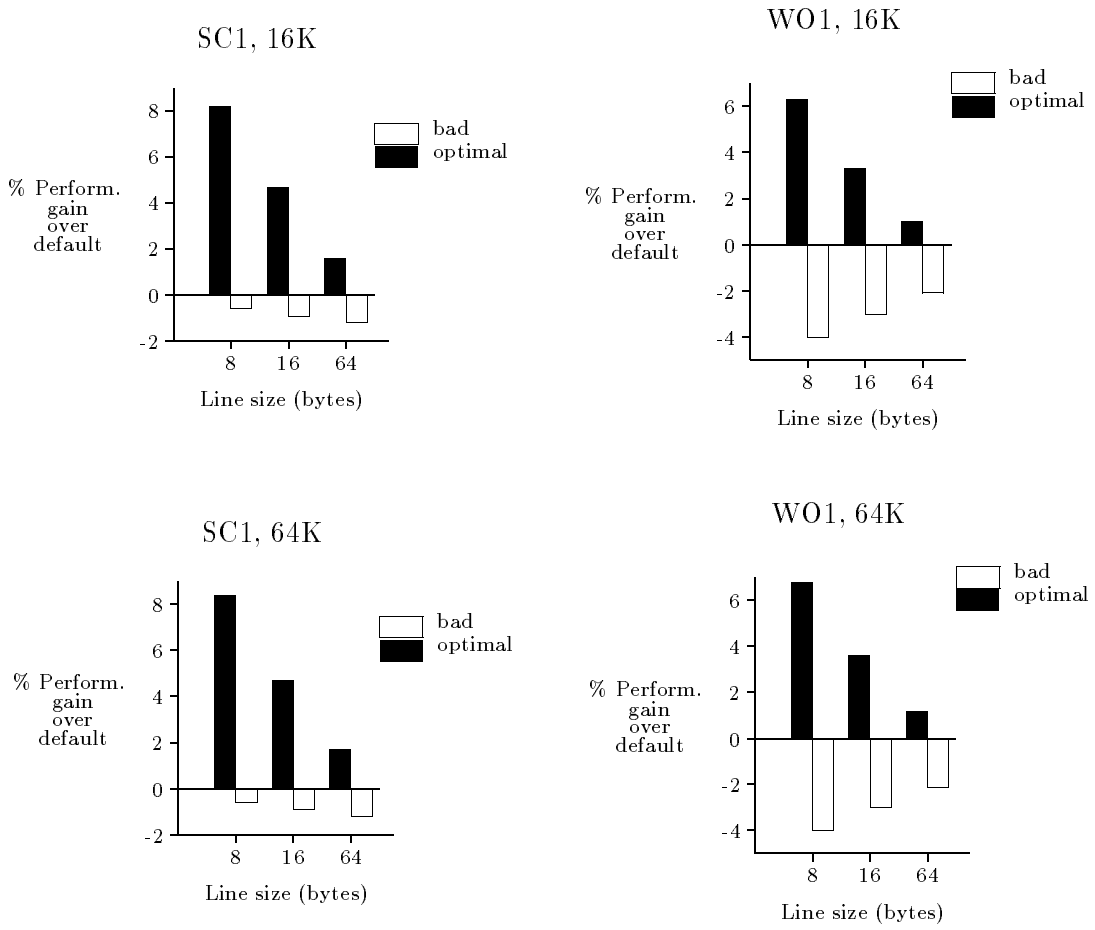


Figure 4.10: Effect of improved code scheduling

The y axis is the change in run-time when using the optimal schedule and a deliberately bad schedule compared to the compiler's default.

are mixed and some cases where four cycle delays have greater relative and absolute gains. Thus, no furthere insight can be gained from the two-cycle experiments.

Table 4.5: Gauss, absolute and relative benefits

Benefits of WO1 over SC1 for load and branch delays of two and four cycles. Absolute is in 1,000's of cycles, relative is in percent improvement.

Cache Size	Delay (cycles)	8 byte lines		16 byte lines		64 byte lines	
		Absolute	Relative	Absolute	Relative	Absolute	Relative
16K	Two	7,938	46.8	3,528	22.9	855	6.1
	Four	7,278	36.2	3,705	20.2	1,075	6.2
64K	Two	308	2.4	239	1.9	157	1.2
	Four	281	1.7	219	1.3	172	1.1

Table 4.6: Qsort, absolute and relative benefits

Benefits of WO1 over SC1 for load and branch delays of two and four cycles. Absolute is in 1,000's of cycles, relative is in percent improvement.

Cache Size	Delay (cycles)	8 byte lines		16 byte lines		64 byte lines	
		Absolute	Relative	Absolute	Relative	Absolute	Relative
16K	Two	2,348	16.0	2,671	18.9	2,109	13.9
	Four	2,530	15.9	2,544	15.9	3,042	18.4
64K	Two	2,261	16.5	1,860	13.0	2,337	15.1
	Four	2,347	15.3	2,247	14.3	2,232	13.4

4.4 Related Work and Comparisons

4.4.1 Study Comparison

The study in Section 4.2 has very different results than the one in Section 4.3. This confirms some of the observations made at the end of Section 4.2 about the limitations of that study. Higher memory latencies needed to be considered and it can be argued that the latencies I used still are not high enough [74]. Also, since read latency was clearly being hidden in the case of *Relax*, some of the performance gains that were observed in the instruction-level study could not have been seen in a normal trace-driven study.

Table 4.7: Relax, absolute and relative benefits

Benefits of WO1 over SC1 for load and branch delays of two and four cycles.
Absolute is in 1,000's of cycles, relative is in percent improvement.

Cache Size	Delay (cycles)	8 byte lines		16 byte lines		64 byte lines	
		Absolute	Relative	Absolute	Relative	Absolute	Relative
16K	Two	1,651	6.3	993	4.1	543	2.3
	Four	1,466	5.2	836	3.2	416	1.6
64K	Two	1,256	4.8	779	3.2	435	1.9
	Four	1,080	3.9	624	2.3	306	1.2

Table 4.8: Psim, absolute and relative benefits

Benefits of WO1 over SC1 for load and branch delays of two and four cycles.
Absolute is in 1,000's of cycles, relative is in percent improvement.

Cache Size	Delay (cycles)	8 byte lines		16 byte lines		64 byte lines	
		Absolute	Relative	Absolute	Relative	Absolute	Relative
16K	Two	2,229	10.6	2,555	12.0	3,037	9.3
	Four	2,383	9.0	2,806	10.6	4,387	11.6
64K	Two	2,137	10.3	2,484	11.9	2,916	9.1
	Four	2,267	8.7	2,711	10.3	4,211	11.3

4.4.2 Previous studies

There have been simulation performance studies of relaxed consistency models using the architecture of a mesh connected multiprocessor based upon DASH, an experimental system being built at Stanford [50, 52, 80]. These studies, which used instruction-level simulation, are similar in scope to the study of Section 4.3 but their architectural framework and memory model implementations are significantly different.

Methodology - 1st Stanford Study

The Stanford studies are based on a variant of the DASH [73, 74] architecture with 16 processors connected in a mesh-like fashion. Memory is globally distributed with a portion of global memory associated with each processor and forming its local, or home, memory; the remainder of the global memory is the remote (for that processor) memory.

Each processor has a two-level cache hierarchy with a write-through first level cache and a write-back second level cache. There is a write buffer between the two caches and if a relaxed consistency model is being implemented, loads can bypass the stores that are in the write buffer. The first level cache has a one word line size and a fetch size on read misses of four words [48]. Cache coherence is enforced by a hardware full directory scheme with the directories being associated with the home memories. The memory latency depends upon the type of memory reference (read, write, synchronization), the memory location of the missing line (home or remote), and the state of the line (clean or dirty). The memory latencies range from 20 to 80 cycles in the case of no contention on the interconnect.

Gharachorloo et al. [50] studied a number of consistency models including: two forms of sequentially consistent systems, a weakly ordered system, and a release consistent system. The major difference between the systems involved the selection of events that led to processor stalls. Specifically:

- In all four cases, after the issue of a read or an acquire the processor stalls until the operation completes, i.e., *blocking loads* are used.
- In all systems except the release consistent one, all writes have to be performed before an acquire or a release is issued.
- In the release consistent system, releases are delayed until all writes are performed but the processor is not stalled. Pending writes do not cause the processor to stall at an acquire.
- Upon issue of a write, either:
  - the processor stalls until the writes are performed (base sequentially consistent)
  - writes are sent to the write buffer but reads won't be allowed until writes are performed (aggressive sequentially consistent)

- writes are sent to the write buffer and the processor continues on (weak ordering, release consistency)

Note that since blocking loads are used, no read latency is being overlapped with computation. Only write latency is.

Results and Comparisons

My experiments show that a relaxed memory model (WO1) using non-blocking loads, lock-up free caches, synchronization primitives visible to the hardware, and stalls on all synchronization points when outstanding memory references are present is worthwhile. Performance improvements over an aggressive sequential consistency model can reach 35%. However, with high cache hit rates the benefit is limited.

The trends observed in the first Stanford study were the same as mine, but their maximum reported performance gain, 40% over their base system in the case of the most aggressive models, was higher. There are several reasons for the quantitative differences. First, the memory latencies for the first Stanford study are much larger, thus giving a greater advantage to relaxed memory models. Second, the implementation of my base system, SC1, is more aggressive since it uses non-blocking loads and a compiler that schedules for them. Third, the cache structures that were used are different. In the Stanford study the first level cache is direct-mapped and write-through (although write hits in the second level cache take only two cycles). Therefore, writes are not performed as quickly as in my write-back cache and consequently the relaxed models have more opportunity to gain since all writes take several cycles to complete. Also, while simulating several cache organizations, I showed that the benefit could vary greatly depending upon the cache and line size; it would be interesting to see whether this is true too of a DASH-like architecture.

There was one case where we reached different conclusions about the benefit of one of the models. In the Stanford study one benchmark, *Pthor*, was found to have significant performance improvement when run on RC compared to when run on WO1.

In my comparable benchmark, *Psim*, which had a similar hit rate and a high level of synchronization, although still only a third of *Pthor*'s, the benefits of RC and WO1 were essentially the same. Most likely the two main reasons for the difference in the level of improvement are: (i) *Pthor*'s higher synchronization rate and, as observed earlier, the more frequent synchronization, the greater the opportunity to benefit from RC, (ii) the higher memory latency in the DASH-like system results in a greater likelihood of a write being outstanding at a release (it is not expected that a read would be outstanding at a release since its value would need to have been used before the release). These factors provide RC with a greater opportunity for a benefit over WO1.

2nd Stanford Study

Gharachorloo et al. [52] presented another study on an architecture similar to DASH. Unlike their previous study, in this one they allowed loads to be non-blocking, thereby allowing read latency to be hidden. However, unlike the study in Section 4.3.3, they saw very minimal effect from the use of non-blocking loads since their compiler does not schedule the code with non-blocking loads in mind as Cerberus' does.

In an attempt to further exploit the out of order execution allowed under relaxed models, Gharachorloo et al. studied the effect of using a processor that allows dynamic scheduling of instructions [62] on a system that implements RC. With this capability instructions do not just complete out of order; they may actually be issued out of order. Therefore the processor can often find instructions to execute in situations where it would otherwise be stalled in the case of a statically scheduled processor. They did find that with a large enough instruction window they were often able to hide the memory latency of almost all normal memory accesses. But for maximum benefit that instruction window needed to be large enough to contain as many instructions as would be executed in the length of time equal to the memory latency. Due to the complexity involved in building such a large instruction window (since the size will have to be proportional to the high memory latencies of a large scale multiprocessor), they were skeptical of the feasibility

of depending solely on dynamic scheduling to hide read memory latency and suggested that a combination of memory latency tolerating techniques such as those described in the introduction of Chapter 3 would more likely be a practical solution [57].

This study cannot really be compared to the study done using Cerberus. Due to the difference in memory latency, modeling technique, benchmarks, compiler, and most importantly, the scheduling of the processors, it is unfair to compare the two studies.

4.5 Relaxed Consistency in Software

Although all references in this dissertation to relaxed models of memory consistency have been made to implementing these models in hardware, there is no need to restrict it to that level. Several systems have been suggested, and some implemented, that relax the consistency model at a higher level, usually in a system implementing some form of shared virtual memory (SVM) [18, 20, 21, 28]. I will give a brief overview of some of these systems. A more in-depth explanation is beyond the scope of this dissertation.

Munin [18, 28] runs on a network of workstations and provides the user with an image of a single address space using shared virtual memory. However in order to provide reasonable performance Munin supports release consistency. This is implemented in software at the page level. Normally in an SVM system, when a processor wants to write a page, it must either invalidate all other copies of that page before writing, or it must send updates every time that it writes. Clearly both of these are more expensive operations than the equivalent operations at the cache level in a shared-memory multiprocessor with hardware coherent caches. Either there will be large numbers of write messages being sent over the network, causing huge amounts of networks traffic, or there will be possibly large numbers of invalidation misses, causing fewer but much larger messages to be sent. Because it uses release consistency, Munin does not require that all copies of the page being written be invalidated on a write or that updates be sent with every write. It delays the writes until a release is performed. Then it packages all the writes into a single message. If the program has no data races, then it will still execute correctly. But

now there will only be one large message, instead of many smaller ones, each paying the price of message overhead.

Because Munin is a software level system which must deal with much higher latencies than at the cache level, it can treat different parts of the system in different ways and still provide reasonable performance. For example, Munin allows the user to specify whether or not an object should be kept coherent by using updates or invalidates, and allows the program to change that option dynamically, something that would be impractical to implement at the level of a hardware cache.

Midway [20] is another software level SVM system. It implements what the authors describe as *entry consistency*. This is similar to release consistency. However, the key difference is that in a system that is entry consistent, each synchronization variable has specific shared data associated with it. When a synchronization operation is done, whether it be an acquire or a release, the only accesses that need to be performed with respect to the appropriate processor are those involving the data associated with the synchronization variable upon which the acquire or release is being done. Clearly this reduces the amount of network traffic, always a major problem in an SVM system.

Entry consistency also allows the user to specify whether an acquire is for exclusive access (such as a lock) or non-exclusive access (such as a barrier or a synchronizing read). This distinction allows the run-time system to decide whether or not to replicate data, again reducing network traffic by not needlessly invalidating.

Software level systems can take advantage of relaxed models of memory consistency. However, it is clear that due to the latencies of the operations they perform, the tradeoffs are very different than at the hardware level. Optimizations that can be done in software are impractical in hardware. Also, software level systems can have access to information that is not available to the hardware level. For example, an entry consistent system needs to know what data is associated with which synchronization variable. This would be difficult to do in hardware, but fairly easy in software. Also, it would be difficult to implement in hardware the multiple coherence protocols that Munin supports, to say

nothing of changing them dynamically.

4.6 Conclusions

Relaxed consistency can clearly provide a significant performance gain and I would recommend implementing a relaxed model of memory consistency in future shared-memory multiprocessors. However, given the benchmarks and parameters used in my study there is nothing to indicate that one form of relaxed consistency is superior over another. When choosing one to implement one relaxed model over another, no choices should be made that might be detrimental to the performance of other aspects of the system.

In this chapter I have presented studies using lock-up free caches and bypassing. However, there are other architectural choices allowed with relaxed models of consistency. One very important choice that may provide additional gain is the use of software controlled cache coherence. This is considered in the next chapter.

Chapter 5

Software Controlled Cache Coherence

For bus-based shared-memory multiprocessors cache coherence is usually maintained using snoopy protocols [12]. But these protocols are impractical for large-scale machines which do not have a single broadcast medium such as a bus. Instead, directory based coherence schemes [30] have been put forward as the solution to the cache coherence problem in such machines. Implementing directory schemes presents some difficulties though. Full directories do not scale well and directories in any form complicate the memory controller. Also, the hardware enforced cache coherence (HWCC) that directories provide can lead to a loss of performance because of false sharing [44]. There have been many suggestions on ways to deal with the scalability issue with directories [11, 25, 31, 58, 81]. However, these proposals still result in false sharing and at least some increased hardware complexity and directory memory. All these problems are dealt with by using software controlled cache coherence (SCCC) [2, 36, 38, 82].

In an SCCC system the hardware is much simpler than in an HWCC system. There is neither a hardware directory nor a state machine at the memory or cache controllers for processing the sending and receiving of coherence messages to and from processors and memory modules. An SCCC system maintains cache coherence through actions

initiated by the processor and which have only a local effect. These local actions are under software control and are due to special instructions inserted into the program (usually a task of the compiler). The instructions read values directly from memory or invalidate and write-back cached memory locations that are being actively shared (if write-through caches are used, then no write-backs are needed [33]). There is very little extra hardware needed at the memory controller to support SCCC (relative to a uniprocessor memory controller) and false sharing need not be a problem. The main drawback of SCCC is that because the decisions on when to invalidate or write-back a memory location are made at compile-time, the decisions must be conservative. This can cause unnecessary invalidations and writes to memory, leading to more traffic on the interconnection network and higher memory latency, and lower cache hit rates. The question is whether the benefits of SCCC outweigh these costs. That is a question I want to address in this chapter.

The rest of this chapter is organized as follows. Section 5.1 describes SCCC in more detail. Section 5.2 is an explanation of why relaxed models are an integral part of any system using SCCC. It also explains which memory models can be used in an SCCC system, and which cannot as well as explaining why there have been some misunderstandings about this in the past. In Section 5.3 a simulation of an SCCC system is presented and in Section 5.4 its results are discussed. In Section 5.5 is a discussion of previous work in this area.

5.1 SCCC in Detail

Directory-based HWCC is based upon having information that identifies the processors which have cached copies of a line. That information is usually located in a main memory directory associating states with lines.[1] When a processor needs an exclusive copy of a line, a message is sent to invalidate all other copies of the line. Global information is

[1] An exception is SCI [59] which keeps a chain of pointers in the caches of the processors caching a line.

maintained and action is taken globally. SCCC on the other hand relies only on local information and local actions. Neither other processors nor the main memory know anything about whether a processor is currently caching a copy of a line nor if it has written to it.

Software controlled cache coherence can be implemented in several different ways. Regardless of how it is implemented, the main concept of SCCC is that it does not rely on hardware to guarantee that the processor is reading the latest values for a location. The compiler or the programmer inserts special instructions, described below, which ensure that the values to be read are the latest. Which instructions are inserted and where depends upon the program and the SCCC scheme. The set of instructions which are implemented on a given system may vary. However, they are usually chosen from a larger set (cf. Table 5.1.1) that I will review now.

5.1.1 Instructions for SCCC

An instruction that is almost always available is one that *invalidates* a specific memory location in the cache. This instruction guarantees that the next time this address is read, the line will be brought into the cache. If the line is dirty when the invalidation is executed, the semantics of the operation may cause an exception or may simply write the dirty value back to memory while leaving the line clean or invalid (the latter choice is often called a *flush*). If a write-through cache is used, there are no dirty lines and this is not an issue. Another instruction sometimes proposed in the literature is one that is able to invalidate the entire cache; it is usually only considered for systems with write-through caches and will not be considered in this dissertation. Invalidating the entire cache is not usually considered an option for a write-back cache since writing back to memory all the dirty lines in the cache would take too long and would require too much hardware complexity. Instead of invalidation instructions, there could be a *read memory* instruction that forces a value to be read from main memory, even if it is present in the cache [34].

Table 5.1: Special Instructions for SCCC

Instruction	Effect
Invalidate	causes the word (or line) in the cache to be invalidated
Write Back	writes the value in the word back to memory if dirty
Flush	same as invalidate, but writes back dirty words as well
Read Memory	reads a location from memory, even if present in the cache
Write Through	writes a value into the cache and to memory

If write-through caches are used, instructions to locally invalidate words are sufficient. However, if write-back caches are used, then the program needs the ability to forcibly write values back to memory. This can come in one of two forms: either write a specific location back to memory (if dirty), a write-back or *post* instruction, or a normal write to the cache of a new value accompanied by a write through to main memory, a *write-through* instruction. Either alternative is sufficient, but there are gains to be made from providing both, if practical (this is discussed further in Section 5.3.3).

All these instructions may be used in a number of ways. Typically though, before a release operation all values that have been updated will be written-through or forcibly written-back to memory, and before an acquire, any data to be accessed subsequent to the acquire which might have been updated elsewhere will be locally invalidated in order that the updated values will be subsequently read from memory. Some special cache states may be provided which reduce how often the program will need to take action to maintain coherence (see Section 5.3.1).

5.1.2 SCCC Line Size

An issue that is often raised in SCCC systems concerns the choice of the line size in the cache. Most previous proposals for SCCC have assumed a line size of one word. This assumption facilitates the analysis necessary to insert the instructions described in Section 5.1.1. If the line size were greater than one word, then values could be brought

into the cache without being explicitly referenced. This complicates the determination of what needs to be invalidated.

Most modern caches have a line size greater than one word due to the performance benefits for programs that exhibit spatial locality (e.g., see *Gauss* in Figure 4.3). So limiting cache line sizes to only one word is a choice most architects would not support. In addition, SCCC may benefit relative to HWCC when line sizes are greater than one word because false sharing is not a problem for an SCCC system given the proper hardware support. False sharing occurs in HWCC systems when multiple processors are each writing different words in the same cache line, causing thrashing of the cache line. This can be avoided in an SCCC system by maintaining dirty bits on a per word basis. When a line is replaced in the cache and must be written back to memory, the dirty bits can be included with the line to make certain only the new values are written back to memory. Otherwise a problem similar to that described in Figure 4.1 will occur.

5.2 Relaxed Models and SCCC [2]

5.2.1 Why Relaxed Models and SCCC

Relaxed models of memory consistency are an important element of any system that uses SCCC. To understand why this is, we must go back to the definition of sequential consistency, repeated here:

> [A 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.

It is not clear what is meant by executed, a term for which we do not have a formal definition. Performed seems the most likely synonym for this term. The definition of

[2] The reader is advised to review Section 3.5 before reading this section and Section 5.3.2

performed again is:

> A LOAD by processor I is considered *performed* with respect to processor K when issuing of a STORE to the same address by processor K cannot affect the value returned to processor I. A STORE by processor I is considered *performed* with respect to processor K, at a point in time when an issued LOAD to the same address by processor K returns the value defined by the STORE. An access by processor I is *performed* when it is performed with respect to all processors. A STORE is *globally performed* when it is performed with respect to all processors. A LOAD is *globally performed* if it is performed with respect to all processors and if the STORE which is the source of the returned value has been globally performed.

Note that the definition of performed says that a store is performed with respect to a processor K at a point in time when K could read the new value if it were to issue a load instruction. As far as the definition is concerned, it does not matter that we may know that the processor will not be reading that location (e.g., because we know all the programs have no data races). If K could issue a load and still read the old value, the store is not performed with respect to K. If the term *executed* in the definition of sequential consistency is really *performed*, then there can be no sequential order of accesses in an SCCC system because accesses are not performed atomically. Consider the following example. If a new value for location X has been written by processor A into its cache, the access has been started. But the access has not yet completed. It will not be performed until all processors have invalidated their old cached copies of X and A has written the new value to main memory, which may be much later. In the meantime, another access, a write of Y by processor B, may have been initiated and performed. So the write of Y started after the write of X, but completed before the write of X. This implies that there can be no sequential ordering of these two accesses, and therefore the system is not sequentially consistent.

Since an SCCC system is not sequentially consistent, it is natural to consider if it obeys any relaxed models. Consider the system Chen and Veidenbaum simulated in their study of SCCC [33]. It had write-through caches, and the programs only used DoAll loops to express parallelism, which meant that the only synchronization operations needed were barriers. Since write-through caches were used, memory was always up-to-date and no forced write-backs were ever needed. At every barrier the entire cache was invalidated. This way, if data which had been updated by another processor in the previous epoch, i.e., the code between two barriers, was needed by this processor in the next epoch, the updated copy would be read from memory. Chen and Veidenbaum said that the system they simulated is weakly ordered. However, it is not. It does not obey the second rule of weak ordering: no access to a synchronizing variable is issued in a processor before all previous global data accesses have been *performed*. Part of the definition of *performed* says: a STORE by processor I is considered *performed* with respect to processor K, at a point in time when an issued LOAD to the same address by processor K returns the value defined by the STORE. Given this definition, it is clear that a write of a variable in Chen and Veidenbaum's system is not performed until such time as all the processors have invalidated their caches. If processor A writes location X, the store is not performed until all processors have invalidated their caches. Until that time, another processor, B, could read from its cache a value previously stored in X. Since a processor's cache is not invalidated until that processor is about to enter the barrier, A can enter the barrier, that is, start the synchronization operation, before B has entered the barrier, and hence before B has invalidated its cache. So, A will be issuing an access to a synchronization variable before all its normal accesses have been performed (because B will not have yet invalidated its cache, and could read an old value of X, if it were to try).

The argument might be made that because only data-race-free programs are considered, the system is still weakly ordered. This is not the case, as was pointed out in Section 3.5. The definition of weak ordering describes what the hardware must do. No mention is made about the properties of programs that run on such hardware. The

equivalence for programs without data races of weakly ordered (WO) and sequentially consistent (SC) machines was not shown by Adve and Hill [4] until four years after weak ordering was first proposed [41].

A similar problem exists in a paper by Dubois et al. [42]. In that paper the authors present the idea of *delayed consistency*. They consider an HWCC multiprocessor that is release consistent, and whose caches have a multi-word line size. In that system, an invalidation message from a sending processor to a receiving processor is queued at the receiver's end. The invalidation is not processed until the receiving processor is about to perform an acquire. This delay in processing invalidation messages is what motivates the name "delayed consistency". If the programs do not contain data races, then any invalidation message must be due to false sharing, not true sharing. So delaying the invalidation does not cause an incorrect execution since the value that was updated and caused the invalidation message to be sent will not be used by the processor receiving the message until it performs an acquire (this is the reasoning of lazy release consistency). But a system that implements delayed consistency is not release consistent (RC). It is the same problem that was shown above in Chen and Veidenbaum's system. In an RC system all the accesses must be performed before the release is performed. Even though data race free programs will not read the locations in the cache that are to be invalidated by the queued messages, if the processor *were* to read one of those locations, it would be reading an old value, and the ability to read an old value is a violation of the conditions of release consistency.

The common problem in both these cases is the insight from lazy release consistency and my dissertation proposal [96]; that in SC executions of data race free programs accesses do not need to be globally performed at a release, but rather at the following acquire, and this is not in agreement with the definition of WO or RC. The meta-problem though is that the hardware definition of weak ordering and release consistency gave the architects no latitude. Both Chen and Veidenbaum and Dubois et al. proposed systems that would give SC executions to data race free programs. But they were not WO or

RC. This is why the software-centric view of relaxed models presented in Section 3.5 is so important. In fact, it would be very difficult to design a system using SCCC which is WO or RC. But it is not difficult to design one that obeys Adve and Hill's DRF1 Condition. In fact, in Appendix 5.2.3 I show that Chen and Veidenbaum's system obeys DRF1, and therefore the validity of their work remains unchanged. In Section 5.3.2 I show that a system I study obeys DRF1. However, DRF1 needs to be explained in more detail first.

5.2.2 DRF1 in More Detail

As described in Section 3.5, Adve and Hill presented a set of conditions that I will refer to as the DRF1 Condition [5]. They have proven that these conditions, repeated in Figures 5.1 and 5.2, are sufficient conditions for DRF1; that is, any hardware that obeys these conditions obeys DRF1, and therefore provides SC executions to any data race free program. Before showing that these conditions are obeyed by the SCCC system in the previous section, they need to be explained in greater detail.

The following are a series of definitions for *paired operations*, $\xrightarrow{\text{so0}}$, $\xrightarrow{\text{so1}}$, $\xrightarrow{\text{xo}}$, $\xrightarrow{\text{po}}$, $\xrightarrow{\text{hb1}}$, sub-operations of a memory access (X (i)) and *control*. All these definitions are taken from Adve and Hill [1, 5].

> Synchronization operation S_1 is a *release* operation, synchronization operation S_2 is an *acquire* operation, and S_1 and S_2 are *paired* with each other if and only if
>
> 1. S_1 is a write operation, S_2 is a read operation, S_1 and S_2 access the same location, and S_2 returns the value written by S_1,
> 2. the processor that executes S_1 uses S_1 to communicate the completion of all its memory operations ordered before S_1 by program order to the processor that executes S_2, and
> 3. the processor that executes S_2 uses S_2 to conclude the completion of

the operations of the processor that executed S_1 that are before S_1 by program order.

Synchronization-order-1: In an execution, memory operation S_1 is ordered before S_2 by the $\xrightarrow{\text{so1}}$ relation if and only if S_1 is a release operation, S_2 is an acquire operation and S_1 and S_2 are paired with each other.

Synchronization-order-0: $X \xrightarrow{\text{so0}} Y$ if X and Y are conflicting synchronization operations and $X(i) \xrightarrow{\text{xo}} Y(i)$ for some i.

The *program order* (denoted by $\xrightarrow{\text{po}}$) for an execution is a partial order on the memory operation of the execution defined by the text of the program [87].

An *execution order*, denoted by $\xrightarrow{\text{xo}}$, for an execution of a program is a total order defined on the sub-operations (defined by Collier, below) of the execution such that a read sub-operation returns the value of the write sub-operation ordered last before it that is to the same location and executes in the same memory copy as the read. There may be more than one $\xrightarrow{\text{xo}}$ corresponding to an execution. Sub-operations of an execution *appear* to have executed in some order if an $\xrightarrow{\text{xo}}$ for that execution satisfies the order.

The $\xrightarrow{\text{hb1}}$ relation for an execution is the irreflexive transitive closure of $\xrightarrow{\text{po}}$ and $\xrightarrow{\text{so1}}$, i.e., $(\xrightarrow{\text{po}} \cup \xrightarrow{\text{so1}})^+$ ($\xrightarrow{\text{hb1}}$ stands for happens before).

From Collier [37]:

A shared-memory system with n processors, P_1, P_2, ..., P_n is represented as follows.

1. Each processor has a copy of the shared memory.
2. A write operation W, on location x, is comprised of atomic sub-operations $W(1), W(2), ..., W(n)$, where the sub-operation W (i) atomically updates the location x in the memory copy of $P(i)$ to the specified value.

3. A read operation R, by processor P_i, on location x, is comprised of a single atomic sub-operation $R(i)$, that results in returning the value of x in the memory copy of P_i.

DRF1 Condition: Hardware satisfies Condition 2.1 (defined earlier in Adve and Hill [5]) and therefore obeys the data-race-free-1 memory model if for every execution, E_{drf}, of a program, *Prog*, on the hardware, there is an $\xrightarrow{xo}$, that satisfies the following conditions:

1. *Data* - Let *Rel* and *Rel'*be release operations and *Acq* and *Acq'* be acquire operations. Let Z be any operation. Let X and Y be conflicting operations such that at least one of X or Y is a data operation.
   (a) *Release-Acquire* -
      i. If $Rel \xrightarrow{so1} Acq$, then $Rel\ (i) \xrightarrow{xo} Acq\ (j)$ for all i, j.
      ii. If $Z \xrightarrow{po} Rel' \xrightarrow{so1} Acq' \xrightarrow{po} Rel \xrightarrow{so1} Acq$, then $Z(i) \xrightarrow{xo} Acq\,(j)$ for all i, j.
   (b) *Post-Acquire* -
      i. If $Acq \xrightarrow{po} Z$, then $Acq\,(i) \xrightarrow{xo} Z(j)$ for all i, j.
      ii. If $X \xrightarrow{po} Rel \xrightarrow{so1} Acq \xrightarrow{po} Y$, then $X(i) \xrightarrow{xo} Y(i)$ for all i.
   (c) *Intra-processor* - If $X \xrightarrow{po} Y$, then $X(i) \xrightarrow{xo} Y(i)$ for all i.
2. *Synchronization* - Let X, Y and Z be synchronization operations.
   (a) If $Y \xrightarrow{po} Z$, then $Y(i) \xrightarrow{xo} Z(j)$ for all i, j.
   (b) If X is a write operation, Y is a read operation that conflicts with X and $X \xrightarrow{so0} Y \xrightarrow{po} Z$, then $X(i) \xrightarrow{xo} Z(j)$ for all i, j.
   (c) If X and Y are conflicting write operations, then either $X(i) \xrightarrow{xo} Y(i)$ for all i or $Y(i) \xrightarrow{xo} X(i)$ for all i.

Figure 5.1: DRF1 Condition: Sufficient Conditions for DRF1, parts 1 and 2

Adve and Hill go on to say [5]:

> Although real systems do not usually provide physical copies of the entire memory to any processor, a logical copy of memory can be assumed to be associated with every processor. For example, in a cache-based system, the

3. *Control* -

   (a) Let read R control an operation X or determine the value that X writes (if X is a write). Then $R(i) \xrightarrow{\text{xo}} X(j)$ for all i, j.

   (b) Consider any sequentially consistent execution, E_{sc}, of *Prog* and operations X and Y such that $X \xrightarrow{\text{po}} Y$ and either X and Y conflict, or X is an acquire, or Y is a release, or X and Y are synchronization operations in E_{sc}. Let operation X not be executed in E_{drf} and operation Y be executed in E_{drf}. Let read R control operation X in E_{sc} and let R be one of the reads in E_{drf} whose value determined that X would not be executed in E_{drf}. Then $R(i) \xrightarrow{\text{xo}} Y(j)$ for all i, j.

   (c) If $X \xrightarrow{\text{po}} Y$ and X is an acquire, then $X(i) \xrightarrow{\text{xo}} Y(j)$ for all i, j.

   (d) Let X and Y be synchronization operations. If $X \xrightarrow{\text{po}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(j)$ for all i, j.

Figure 5.2: DRF1 Condition: Sufficient Conditions for DRF1, part 3

1. If X and Y are conflicting operations, at least one of X and Y is a data operation, and $X \xrightarrow{\text{hb1}} Y$, then $X\ (i) \xrightarrow{\text{xo}} Y\ (i)$ for all i.

Figure 5.3: Alternate Data Requirement for DRF1 Condition

logical copy of memory for a processor may be the union of the processor's cache and all the lines from the main memory that are not in the cache. Also, in a real system, some sub-operations may not be distinct physical entities. However, logically distinct sub-operations can be associated with every operation and a memory copy. For example, an update of main memory on a write constitutes the sub-operations of the write in the memory copies of the processors that do not have the line in their cache. Finally, in most real systems, sub-operations appear to execute atomically, i.e., the result of an execution is as if the sub-operations executed in some sequential order.

A read R *controls* memory operation X if (a) both R and X are by the same processor, and (b) the value that R returns determines if X will be

executed, or determines the location accessed by X, or determines the value written by X (if X is a write).

A condition such as "$X\ (i) \xrightarrow{xo} Y\ (j)$ for all i, j" implicitly refers to values of i and j for which both $X\ (i)$ and $Y\ (j)$ are defined.

Adve and Hill also say that a reference to $X(i)$ for all i in the case where X is a read is only defined for the processor issuing the read. Also, "$X(i) \xrightarrow{xo} Y(i)$ for all i" implicitly refers to values of I for which both $X(i)$ and $Y(i)$ are defined.

The DRF1 Condition in Figures 5.1 and 5.2 has three main parts. The data requirement has a number of subcomponents. Adve and Hill [5] also show that an alternate data requirement with fewer subcomponents may be used in instead of sub-condition 1. In certain proofs of obeying DRF1 I will use this alternate condition, which is in Figure 5.3.

5.2.3 Obeying DRF1

Chen and Veidenbaum

The proof that Chen and Veidenbaum's system obeys DRF1 is in Appendix D.

Dubois et al.

There is not enough information in Dubois et al. [42] to determine if their system obeys DRF1. However, most likely it does as most systems obey DRF1.[3] Adve and Hill [5] do explain why most systems obey DRF1 by describing what architectural features result in a system obeying each of the requirements.

5.3 Methodology

In this section I will describe the methodology used in my simulation of an SCCC system. Section 5.3.1 describes the basic architecture used. Section 5.3.2 shows that this system

[3]If the system is RCpc instead of RCsc, it cannot obey DRF1. However, it is not clear in from their paper [42] if the system is RCpc or RCsc.

does obey DRF1. Section 5.3.3 describes the benchmarks used.

5.3.1 Basic Architecture

This simulation uses the same Cerberus simulator and basic architecture described in Section 4.3.1. Only differences between the two architectures will be expounded upon.

There is no cache coherence hardware at either the cache or the memory controller. The memory controller now simply receives memory requests without any indication if a request is for read or write. The request is satisfied by the memory module and the line is returned to the requesting processor. There is no possible delay for coherency messages to be sent and acknowledged. Also, the memory latency in the case of no contention is now one cycle shorter (17 cycles) since there is no longer any directory look-up phase.

There are three new features for the memory controller. The first two are both related and simple features to implement. They entail writing of individual words or selected words within a line rather than whole lines. First, the memory must now be able to accept short messages which are updates to single words. Second, for reasons explained below, when an entire line is written back, there is now a mask associated with the dirty line. The mask is a bit vector indicating which words in the line from the processor are actually dirty and need to be written back. If the bit for a given word is not set, then no change is made to the memory copy of that word.

The third new feature of the memory controller relates to synchronization. Since there is no hardware to maintain cache coherence, synchronization variables are not cached, and all synchronization is done at the main memory. To avoid polling across the network, the synchronization primitives implemented are similar to full/empty bits [88]. However, the bits are not separate from the memory word, but rather, are stored in the word. So any memory location can be used for synchronization, but it must be accessed with one of three special instructions. The first instruction is *read_empty*, which is used for locking. *Read_empty* checks a memory location and if the location contains a non-zero value, the instruction will return successfully allowing the processor to proceed while

also storing a zero in that location. If the location contains a zero, then the processor's request is enqueued at the memory module until such time as it can be executed (a FIFO queue of outstanding synchronization requests is maintained per memory module) and the requesting processor blocks while the read_empty request is outstanding. *Read_full* is very similar to *read_empty*. However, instead of leaving a zero in the memory location after reading it (reading it and leaving it empty), in this case it leaves the value unchanged in the memory location. This instruction is useful when a number of processors need to read a global flag for a synchronizing read. The third instruction is *write_sync* which is used for unlocking. It writes a given value into a memory location and then attempts to see if the memory module's enqueued synchronization requests can now succeed. *Write_sync* does not stall the processor.

Three other instructions are added for SCCC. The executing program can direct that a specific word in its cache (if present) be invalidated, thereby forcing the next reference to that location to be fetched from main memory (an exception occurs if the word is dirty). There is also an instruction to forcibly write a word from the cache back to the main memory (if present in the cache). Finally, there is an instruction that performs a write-through of the cache. If the word is in the cache, the word is updated there and in main memory. Otherwise the word is just updated in main memory. If it is known that a write to a location will always be executed and it is the last write to that address before a release, it is preferable to use a write-through instead of write-back in order to improve code density (this is discussed further in Section 5.3.3).

In the cache there is no longer a single dirty bit associated with each line. Instead there are three state bits associated with each word in a cache line, and these can represent one of five different states (cf. Table 5.3.1). INVALID and DIRTY are self-explanatory. Before explaining the other states, the programming model assumed must first be explained.

In all programs the software will need to forcibly write-back to memory new values

Table 5.2: SCCC Cache States

State	Meaning
INVALID	word not valid, fetch from memory
STALE	word valid if read before next barrier
DIRTY	word in cache is dirty
FRESH	word valid
GOOD	word valid, just written back to memory

for a location.[4] If the program only uses barriers for synchronization (e.g. when only DoAll loops are used to express parallelism), then the software need never explicitly invalidate cached memory locations (why is explained in the next paragraph). If locks or synchronizing reads or writes are used, then the software must explicitly invalidate memory locations in addition to writing them back in order to maintain coherence.

To correctly execute programs that use only barrier synchronization, the architecture takes advantage of the semantics of such programs. Between two barriers, i.e., an epoch, there can be no inter-processor data dependency. If there were, then some form of synchronization such as a synchronizing read/write pair would be needed. Therefore, between barriers it is never the case that a processor writes data that another processor needs before the next barrier. Before the barrier is reached the software must guarantee that all data that is dirty and may be needed by another processor in the next epoch is written back. If a processor reads a location in epoch i that was written by a different processor during epoch $i-1$, then it needs to read the updated value from main memory, not its cache. The read from memory is guaranteed by the "aging" of words in the cache.

When there is a read miss in the cache, the necessary line is fetched from memory. The word that is read is brought into the cache in state FRESH. All the other words in the line are brought in state STALE. If a STALE word is read at any point, it is set to state FRESH. When the barrier is reached, those words which have been referenced

[4]Software here can refer to actions taken by the programmer or the compiler to ensure coherence. It is irrelevant to the architecture who has that responsibility.

during the past epoch will be FRESH (or DIRTY). If they have been brought into the cache only because of spatial locality, then they will be STALE. As part of the barrier transition, FRESH words are changed to STALE ones, and STALE ones are changed to INVALID ones. If the word was STALE, it was not referenced by this processor in this past epoch, and therefore may have been written by another processor, and should be invalidated since its value may be out of date during the next epoch. If the word was FRESH, it can be kept valid in the cache into the next epoch since it could not have been updated by another processor during that past epoch (if it had been, there would be an inter-processor data dependency). However, it must be kept in the cache in state STALE since we do not yet know if it will be referenced in the next epoch by this processor. If it is not referenced by this processor in the next epoch, another processor may write it, in which case this copy needs to be invalidated at the following barrier, which happens if it is still STALE. If it is referenced by this processor during this epoch, it will automatically be set to state FRESH by the read.

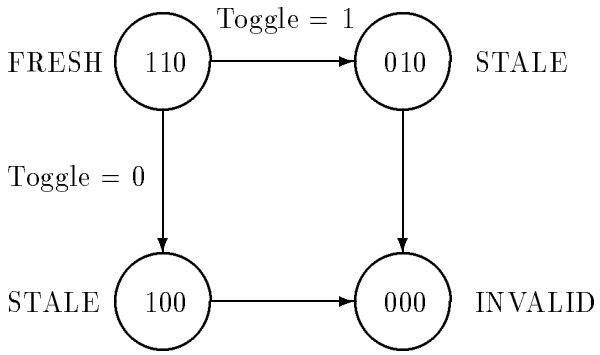


Figure 5.4: State transitions at barriers for "aging" of cache words

Although "aging" was simulated, the length of time for the process was not modeled. There are two reasons that choice was made. Since "aging" occurs as part of a barrier, it could easily be overlapped with the barrier code and not require extra time to implement

it, except perhaps for the final processor that arrives at the barrier, and in that latter case it still may depend upon the type of barrier implemented and the number of processors synchronizing. The other reason for not counting cycles "aging" is that given the right state encodings, it can be implemented fairly easily and take very little time. Consider the DIRTY, FRESH, INVALID and STALE states. Suppose three bits are used and their encodings respectively are 001, 110, 000, and either 010 or 100 for STALE (see Figure 5.4). When a word is set to the STALE state (e.g., when it is brought into the cache because another word in the same line is referenced), the encoding for STALE that is used is based upon a one bit toggle. When a barrier is reached, the first or second leftmost state bit of every word in the cache is cleared and the bit that controls the STALE state is toggled. Whether it is the first or second state bit that is cleared is determined by the toggle bit. This way the bit cleared alternates at every barrier. This also means that every word that is FRESH will be set to STALE after one barrier and then INVALID after a second one (unless some other action changes the state). So a word starts out in state 110, then transitions to 100 or 010 and then to 000. The state transformation can be accomplished in a cycle or two since the bit is cleared for every word in the cache, regardless of the state it is in.

An additional cache state is useful. This happens in the case of a synchronizing read/write, where a memory location needs to be written back by the writing processor and invalidated (forcing a memory read) in all the reading processors. In some programs, such as gaussian elimination, the processor writing the value back also needs to read it. There is no reason for it to invalidate its own copy, which is up-to-date. Rather than requiring the software to check and see if it was the writing processor, and hence, it need not do the invalidation (this extra test has detrimental aspects; see Section 5.4), a simple solution was found. A new state, GOOD, was introduced. A word that is written back is set to GOOD rather than FRESH. A GOOD word is treated the same as FRESH except when an attempt is made to invalidate a GOOD word. In that situation its state is changed to state FRESH rather than INVALID. If both the GOOD state and the fast

mechanism for "aging" are to be used, then four states bits would be needed.

Finally, it was decided that there is no change to the state of a word that is DIRTY at a barrier. Initially it seemed that this should be an error and cause an exception. However, it was realized that there are points in the program's execution, such as in the iterative relaxation benchmark, when the programmer may realize that shared data is not being shared. If the software does not force the data to be written back, then the hardware trusts the software to know what it is doing.[5]

Since there is no longer a directory at the memory controller, when a line is replaced due to a conflict or capacity miss, it is not the case that a message is automatically sent to the memory to report the spill. A message is sent only if there is a dirty word in the line being replaced. But care must be taken when writing the line back to memory. In an SCCC system it is possible for two processors to have the same line dirty. Processor A may have written the first word in a line and processor B may have written the second word in the same line. If A writes back the line and memory totally overwrites its own copy with A's and then B does the same, then A's update is lost (this is similar to the problem described in Figure 4.1). So, when a dirty line is replaced in the cache, the entire line is sent to main memory. However, a bit vector is sent along as well. A bit is turned on only for those words which are dirty, and only those words with a bit on are updated in main memory (recall the features required for the memory controller at the beginning of this section).

5.3.2 Conforming to DRF1

While showing that my system obeys DRF1, I will not use the data requirement of the DRF1 Condition in Figure 5.1. I will show that my SCCC system obeys the alternate data requirement in Figure 5.3, although for the rest of the proof, I will use the control and synchronization requirements of the DRF1 Condition.

[5]This may only be necessary because the Cerberus compiler does not allow global private data, forcing *Relax* to use the shared data space at times when it is not truly necessary.

Data Requirement

There are two major situations to consider for the condition in Figure 5.3. X and Y could both be issued by the same processor or they could be issued by different processors. If they are on the same processor and one of X and Y is a read, then $X\ (i) \xrightarrow{\text{xo}} Y\ (i)$ is only defined for i equal to the issuing processor. Since they are conflicting operations, they are accessing the same location, and the data dependence hardware will not allow Y to be issued until X completes. The hardware always stalls whenever an attempt is made to execute an access to an address that already has an outstanding access.

If both X and Y are writes, then it is more complicated. X has completed when the new value written by X is written back to main memory and each processor has invalidated its cached copy of the location (or did not have a cached copy of the location initially). As said in the previous paragraph, if an attempt is made to execute Y if X has not completed, it will result in the processor stalling. So $X(i)$ for i equal to the issuing processor will always complete before $Y(i)$. If the value in the cache is to be written back to memory, and the value is still that written by X, it cannot be received at memory after Y's value if Y's value is written back later or even immediately following as part of a write-through. There is no bypassing in the network, and it is an omega network, so there is only one path from each processor to memory. A message M sent from A to B before a message N, will always arrive at B before N. So there is no way for the write by X to arrive at memory after the write of Y, and other processors will only see the new value by loading it from memory. There is no direct processor to processor communication which would allow the value of Y's write to arrive at another processor before X's.

If X and Y are on different processors, the process is similar. For the $\xrightarrow{\text{hb1}}$ relation to hold, there must be an intervening $Rel \xrightarrow{\text{so1}} Acq$ relation. The release operation will not be issued until such time as all reads have returned values and all writes have completed to main memory. The acquire paired with the release will not complete until the release has completed and the processor issuing the acquire will stall until the acquire completes.

Remember that if X or Y is a read, $X(i) \xrightarrow{xo} Y(i)$ for all i refers only to i equal to the processor issuing the read. If Y is the read, then the reading processor will have invalidated the location Y references before the acquire (this is part of the system, as enforced by the compiler). Therefore $Y(i)$ will refer to the memory copy, which will be up-to-date because it will have been written back to main memory before the release was issued, and $X(i)$ will have completed for i equal to the reading processor before Y (i) is issued. If X is the reading processor, the read will have completed before the release, and hence, before Y is even issued.

If X and Y are both writes, then $X(i) \xrightarrow{xo} Y(i)$ for all i refers to all processors. This condition is only not obeyed if $Y(i) \xrightarrow{xo} X(i)$ for some i. As mentioned in the previous paragraph, before Y is even issued, X's value is written to main memory. A processor cannot read the value Y writes and then the value X wrote. A request is satisfied first by the cache, and then by memory if necessary, not by another processor. Until Y's value is written back to memory, thereby overwriting X's, only X's value is read from memory. The processor that originally executed X will not write the value back again since it is clean in its cache and neither will any processor that cached it. The processor that executes Y will read Y's value from its cache before Y is written back to memory. But it will never read X from memory since before it would read that location from memory, it would have written the dirty value in its cache back.

Synchronization Requirement

Sub-condition 2.a basically says that synchronization operations need to be sequentially consistent. My implementation of SCCC implements processor consistent synchronization operations. However, this is not really an issue in performance or correctness. Synchronizations were made PC for implementation simplicity. However, given the data sets used in my simulation, synchronizations do not occur frequently enough that a release overlaps with a subsequent acquire. Therefore, the fact that the synchronizations are PC instead of SC never comes into play. However, if I were to rewrite the simulator,

I would implement SC synchronization operations.

As in the Chen and Veidenbaum system, sub-conditions 2.b and 2.c are obeyed because synchronization variables are not cached.

Control Requirement

3.a: There is a data dependence somewhere between R and X. Since instructions are issued in order, the processor will stall before possibly executing X due to the hardware enforcement of data dependences and the condition is obeyed.

3.b: In SCCC instructions are issued in order. Since R controls X, R must complete before it can be determined that X is not executed. Therefore R (i) for all i is true before Y is even issued since $X \xrightarrow{\text{po}} Y$ and there is in order instruction issue.

3.c: Since X is an acquire, and the processor issuing X stalls until X has completed and there is only one copy of a synchronization variable, $X(i)$ for all i completes when the acquire has completed, which is before Y is even issued. So X (i) $\xrightarrow{\text{po}}$ Y (j) for all i, j holds.

3.d: This is the same condition as 2.a.

This completes the proof that my implementation of SCCC on the model architecture obeys DRF1.

5.3.3 Benchmarks

The two benchmarks used are *Gauss* and *Relax* from Section 4.3.1. In order to run them on the SCCC version of the simulator, special instructions for invalidations, write-backs and write-throughs were added by hand to the source and assembly language versions of the programs. The analysis needed for the addition of these instructions was too complex to allow the use of the other Section 4.3.1 benchmarks, *Qsort* and *Psim*. It would have been too laborious to do the marking manually, although a compiler would hopefully be able to do the marking automatically (albeit more conservatively).

To provide some idea of the work needed and to facilitate later observations, I will

present the modifications that were necessary for the SCCC versions of the programs. However some constructs need to be explained first. The statement MASTER guarantees that only one processor will execute the following statement or block. It is used during initialization or to simulate a sequential code block (SCB) of a barrier by using MASTER between two of Cerberus' butterfly barriers [22], which is a type of barrier that does not permit an SCB. "read_full" is a routine that implements the read_full instruction described previously. The unlock routine implements the write_sync instruction. Asm("sync") inserts into the assembler code a SYNC instruction, which stalls the processor until such time as there are no outstanding memory accesses. This was used in the WO system since there was no instruction to simulate synchronizing reads and writes.

```
for (steps = 0; steps ¡ iters; steps++) {

  /* first do all the calculations. */
  for (x = xstart; x < xstart + xrange; x++) {
    for (y = ystart; y < ystart + yrange; y++) {
      mytile [x−xstart] [y−ystart] =
          /* average of A [x] A [y] and its eight neighbors */
    }
  }
  BARRIER;
  /* then write them into the matrix. */
  for (x = xstart; x < xstart + xrange; x++) {
    for (y = ystart; y < ystart + yrange; y++) {
*** A [x] [y] = mytile [x−xstart] [y−ystart];
    }
  }

  BARRIER;
}
```

Figure 5.5: *Relax* code

The code for the heart of *Relax* on a hardware cache coherent machine is given in Figure 5.5. Each processor computes the result of the relaxation on a sub-matrix of

size xrange by yrange starting at location (xstart, ystart). The processor writes the result of this computation into the matrix mytile (each processor has its own, dynamically allocated mytile matrix, but it is in shared memory). After the barrier synchronization, each processor copies the new sub-matrix from mytile back into the main matrix, A. The only difference between this code and the version for the SCCC system is that the line marked with $\star\star\star$ is a write-through for SCCC. That is, it is written directly to memory as well as the cache.

```
for (k = 0; k ¡ dims; k++) {
  asm( "sync" );
  while (flags [k] == 0);
  for (i = k + 1 + (_TINDEX + _TSIZE − (k % _TSIZE)) % _TSIZE;
       i < dims;
       i + = _TSIZE) {
    double temp = A [i] [k];
    if (temp != 0.0) {
      A [i] [k] = 0.0;
      temp /= A [k] [k];
      for (j = k+1; j < dims; j++) {
        if ((i != j) || ((A [k] [j] * temp) != A [i] [j]))
          A [i] [j] − = A [k] [j] * temp;
      }
      B [i] − = B [k] * temp;
    }
    if (i == k+1) {
      asm( "sync" );
      flags [i] = 1;
    }
  }
}
```

Figure 5.6: *Gauss* code, hardware cache coherence, reduction

The code for *Gauss* on a hardware coherent machine is given in Figures 5.6 and 5.7. Figure 5.6 has the code for the reduction phase and Figure 5.7 the code for the back substitution phase (including the SCB between the two phases of the computation).

```
BARRIER;
MASTER {
   B    [dims − 1] /= A [dims−1] [dims−1];
   flags [dims − 1] = 0;
}
BARRIER;
for (i = dims−1; i >= 1; i−−) {
   if (_TINDEX == ((i−1) % _TSIZE)) {
     asm( "sync" );
     while (flags [i] == 1);
     B [i−1] − = A [i−1] [i] ∗ B [i];
     B [i−1] /= A [i−1] [i−1];
     asm( "sync" );
     flags [i−1] = 0;
     }
   else {
     asm( "sync" );
     while (flags [i] == 1);
   }
   for (k = _TINDEX; k < i − 1; k + = _TSIZE) {
     B [k] − = A [k] [i] ∗ B [i];
   }
}
```

Figure 5.7: *Gauss* code, hardware cache coherence, back substitution

The SCCC version is similarly divided in Figures 5.8 and 5.9. Write-throughs are again labeled with $\star\star\star$. Write-backs and invalidations are indicated by routines with those names (in actuality, the assembly language version of the program was modified manually for all three operations).

If Figures 5.6 and 5.8 are compared it is clear that a fair amount of code has been added for the SCCC version and this code needs to be explained. The SYNC instruction and the while loop on the second and third lines of Figure 5.6 are replaced by the read_full call in Figure 5.8. This synchronization is necessary so that the processor does not attempt to read A [k] [j], j > k and B [k] before they have been updated (the

```
for (k = 0; k < dims; k++) {
  if ((k > 0)) {
    INVALIDATE( &B [k] );
    for (j = k+1; j < dims; j++) {
      INVALIDATE( &A [k] [j] );
    }
  }
  read_full( &flags [k] );
  for (i = k + 1 + (_TINDEX + _TSIZE - (k % _TSIZE)) % _TSIZE;
       i < dims;
       i + = _TSIZE) {
    double temp = A [i] [k];
    if (temp != 0.0) {
      A [i] [k] = 0.0;
      temp /= A [k] [k];
      for (j = k+1; j < dims; j++) {
        if ((i != j) || ((A [k] [j] * temp) != A [i] [j])) {
          A [i] [j] − = A [k] [j] * temp;
        }
        if (i == k+1) {
          WRITE_BACK( &A [i] [j] );
        }
      }
      B [i] − = B [k] * temp;
    }
    else {
      for (j = k+1; j < dims; j++) {
        WRITE_BACK( &A [i] [j] );
      }   }
    if (i == k+1) {
      WRITE_BACK( &B [i] );
      unlock( &(flags [i]) );
    }
  }
}
```

Figure 5.8: *Gauss* code, software cache coherence, reduction

```
 BARRIER;
 MASTER {
     B [dims − 1] /= A [dims−1] [dims−1];
     for (i = 1; i < dims; i++) {
⋆⋆⋆    flags [i] = UNLOCKED;
     }
⋆⋆⋆flags [dims − 1] = LOCKED;
 }
 BARRIER;
 for (i = dims-1; i >= 1; i−−) {
     if (_TINDEX == ((i-1) % _TSIZE)) {
       INVALIDATE( &B [i] );
       read_full( &flags [i] );
       B [i−1] − = A [i−1] [i] ∗ B [i];
⋆⋆⋆    B [i−1] /= A [i−1] [i−1];
       unlock( &flags [i−1] );
     }
     else {
       read_full( &flags [i] );
     }
     for (k = _TINDEX; k < i − 1; k + = _TSIZE) {
⋆⋆⋆    B [k] − = A [k] [i] ∗ B [i];
     }
 }
```

Figure 5.9: *Gauss* code, software cache coherence, back substitution

updating is signaled by the writing of flags by the call to unlock at the bottom). The two invalidates, one in a loop, just before the call to read_full are to ensure that when this processor reads A [k] [j], j > k and B [k], it reads the new values for those locations that were updated by the processor that "unlocked" flags. Before flags is "unlocked", the new values must be written to memory. Hence the write-back of B before the call to unlock, and the write-back of A [i] [j]. The write-back of A [i] [j] is a good example of the complexity of efficient marking for SCCC. When I first modified the code for SCCC, there were far fewer additions. However, that code did not perform very well. It was

clear that new values were being written back to main memory for a given location too many times and that too many invalidations were being done. The value computed in the line A [i] [j] $-$ $=$ A [k] [j] $*$ temp; was originally written-through every time this statement was executed. But that caused many unnecessary writes. Instead it is now forcibly written-back in one of two places: in the if statement inserted after the above mentioned assignment statement or in the for loop in the else clause of the test of temp.[6] This reduces the consumption of interconnection network bandwidth and produces a shorter run-time. However, there is a cost paid in how much code is executed. This is examined in more detail in Section 5.4.

The code added to the back substitution phase is fairly straight-forward. The writing of flags is replaced by calls to unlock and the spin-waiting on flags is replaced by read_full calls. The new values of B are written-through to main memory. This phase could be more efficient. However, the program spends only 5% of its time back substituting and this simple scheme already was faster than on WO.

Another aspect of the SCCC modifications that I realized while conducting my experiments is the difficulty in "getting it right". The original, overly conservative marking of the program was easy to produce, and executed correctly. However, it executed a huge number of unnecessary write-throughs to memory. The more efficient version whose end product can be found in Figures 5.8 and 5.9 required a fair bit of work. There were several versions between the original and final ones that were incrementally more efficient, but not efficient enough and not all the modifications were immediately more efficient and correct. Of course these transformations will hopefully all be done automatically by a compiler or a tool that is part of a parallel programming environment. However, as is always the case, the programmer may be able to place the modifications more efficiently based upon knowledge of the program's behavior.

[6]There could be an if statement here checking to see if i equals k+1, but this was found to be of negligible benefit.

5.4 Results

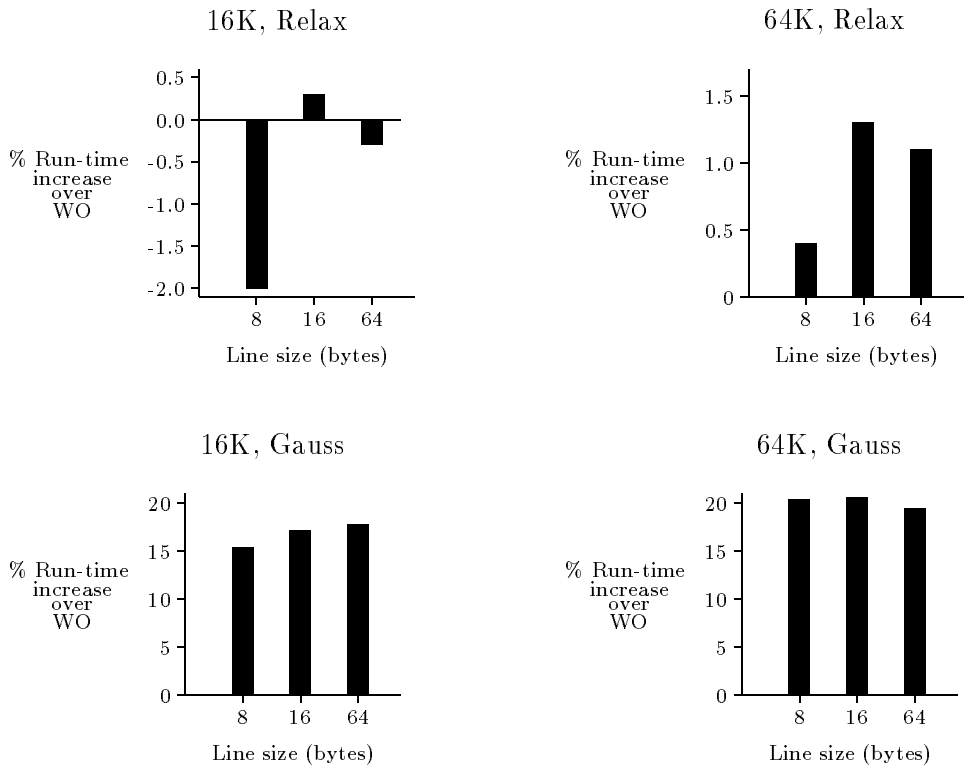


Figure 5.10: SCCC performance relative to WO

In Figure 5.10 are the results of comparing the performance of SCCC to WO (the WO numbers are from Section 4.3.2). The bar graphs show the percent run-time increase/decrease from using SCCC compared to WO for the range of cache line sizes used previously.

5.4.1 Relax

For *Relax* SCCC provided equivalent performance to WO. The performance varied from 2% better to 1.5% worse. However, this should not be a surprise for *Relax*. The only change made to the program was that one write be write-through. As described in Section 4.3.2, those values that are being written-through would be replaced in the

cache naturally as part of the normal replacement process due to conflict misses. So the quantity of memory bandwidth consumed by SCCC and WO is the same. SCCC simply uses the bandwidth a little sooner. As seen in Figures 5.11 - 5.13, there is little difference in latency and memory module utilization between SCCC and WO. There is a difference in the number of messages received at the memory modules, but it does not appear to be large enough to make a difference in performance. WO performs slightly better with larger line sizes and SCCC with smaller line sizes. There are two probable sources of this difference. One is the replacement of clean lines. In WO when a clean line is replaced in the cache, a message indicating the spill of the line must be sent to main memory so that the directory can be updated. In the case of SCCC, if a clean line is replaced, there is no message sent. The second difference occurs when writing the new values back into the A matrix. Since write-through is used, if there is a miss in the cache, the line is not fetched. That is, there is no write-allocate. In WO when there is a miss for this write, a request for the line (which is not actually needed) is sent to memory, which does not occur in the SCCC system. It should be noted that because a write-through is executed for each element of the matrix, as the line size increases, there should be a benefit for WO since there will only be one message per line sent to memory when the line is replaced instead of one per matrix element, a floating point double in SCCC. This is reflected by the graphs in Figure 5.13 which show more messages to memory for WO with 8 byte lines, but fewer with 64 byte lines.

There was one statistic which indicated a significant difference in the behavior of the two systems. The SCCC system stalled for millions of cycles due to all the MSHR's being full. Recall that whenever a write-through or write-back is done in the SCCC system, the hardware must keep track of this outstanding access. This is needed so that at a release point it is known whether or not all outstanding accesses have completed. Thus for every write to memory, which refers to spills of dirty lines, write-throughs and write-backs, an MSHR entry is used to keep track of the outstanding access. Upon completing the actual write, the memory module must then send an acknowledgment so

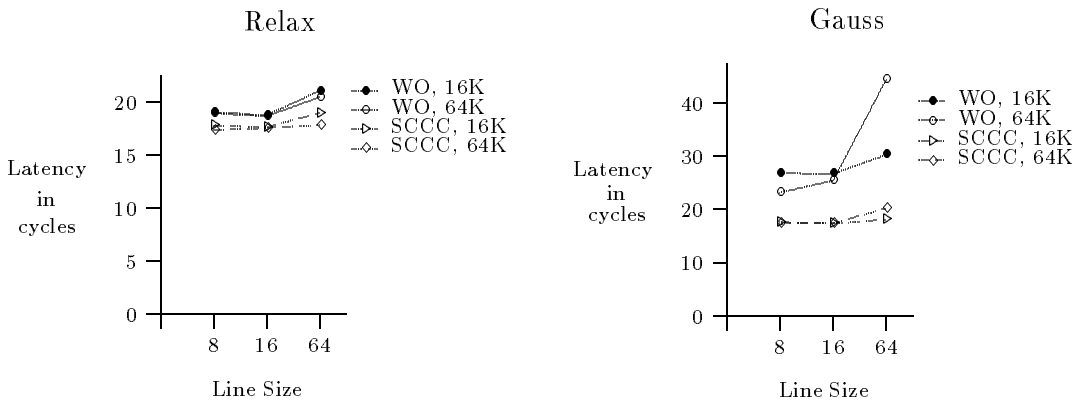


Figure 5.11: Memory Latencies

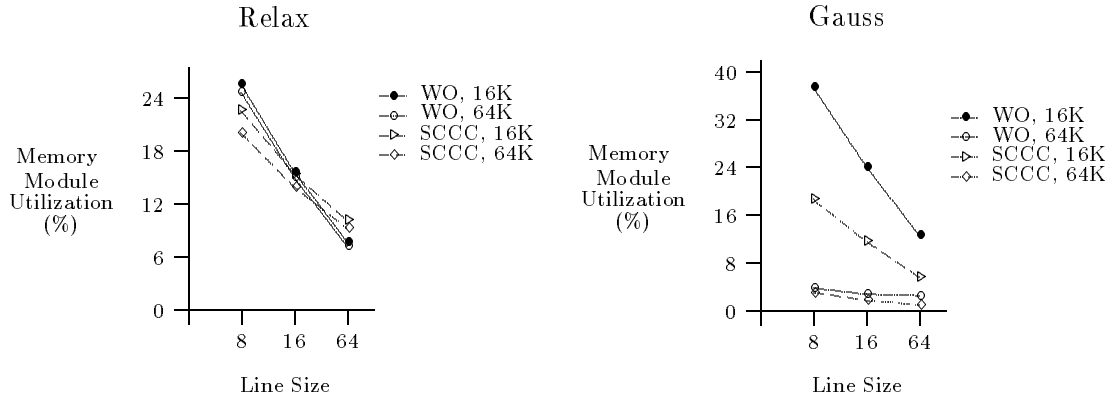


Figure 5.12: Memory Module Utilization

that the cache controller knows that it can clear the corresponding MSHR entry. When a large number of writes to memory are done in comparatively short order, as in *Relax*, there are often not enough MSHR entries. The processor is then stalled, hence the large number of stall cycles. This does not appear to be a performance loss since SCCC was still competitive with the WO system. A hypothesis was that SCCC would perform even better if this bottleneck were removed. To this end I increased the number of MSHR's from five to one hundred. The run-times were essentially the same. Given this behavior,

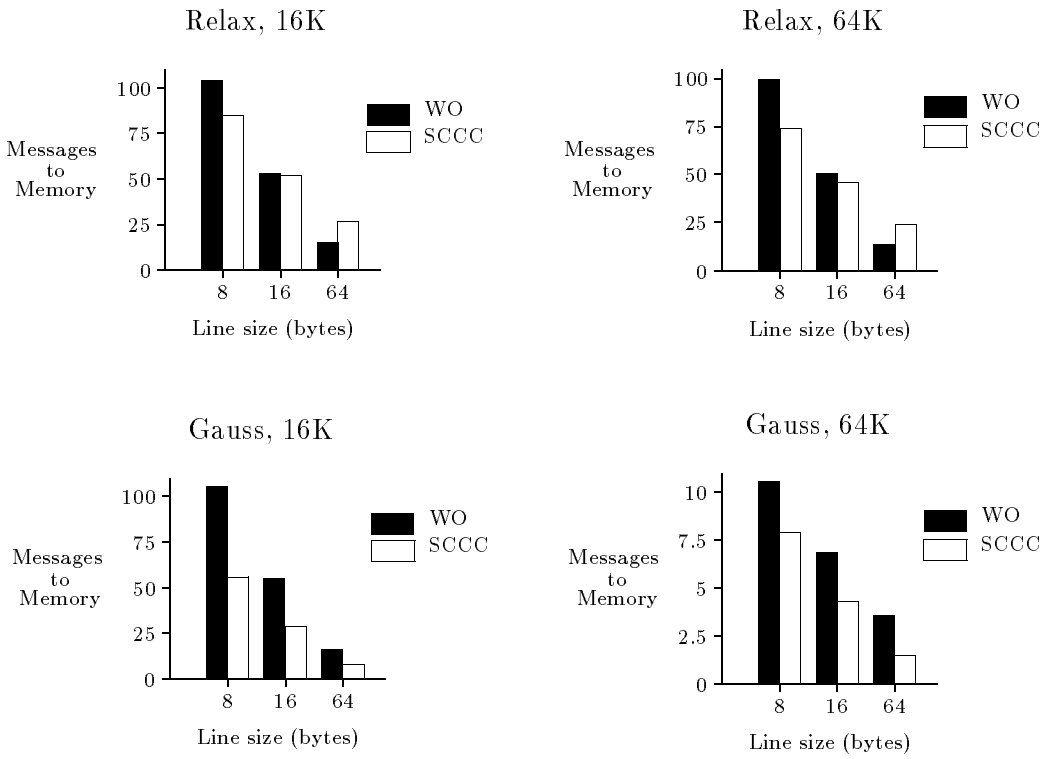


Figure 5.13: Number of Messages (10,000's) to Memory

it is clear that stalling until these writes can be completed at the memory module is not a bottleneck. It seems likely that the memory module is continually busy during this phase due to the large number of writes. There are enough writes to process that it does not matter whether the processor sends out five and then sends out pending ones individually as MSHR's become available, or if 100 are sent to memory in comparitively short order. The memory is apparently equally busy in both cases. It may not seem like this is the case given low memory module utilization (approximately 15%). However, it should be remembered that this is an average over the entire run of the program and that *Relax* writes to memory in a bursty fashion, mostly in the second phase of its computation. Since there is a barrier at the end of this phase, the processor cannot enter the barrier until all the writes have completed. It does not matter if the processor

stalls for a while before sending a write to memory which cannot be processed because the memory module is already busy. If it did not stall, it would simply end up waiting longer before entering the barrier.

Although lack of MSHR's was not a major bottleneck in this case, it could be in some situations. For example, if a large number of writes were initiated with no barrier following shortly after, then a processor would stall due to lack of MSHR's. Increasing the number of MSHR's is undesirable as the MSHR's are a very complex structure [75] and using them to monitor outstanding writes to memory is overkill. All that is needed is a counter since it is the number of outstanding writes that must be kept track of, not which individual ones are outstanding. With a counter, every time that a write-through, write-back or a spill of a dirty line is initiated, we increment the counter. When an acknowledgement returns from memory, we decrement the counter. When the counter is zero, a pending release may be initiated. With a simple eight bit counter up to 255 outstanding accesses could be kept track of, a negligible amount of hardware compared to a much smaller number of MSHR's.

There is one problem with the counter scheme and it occurs when a release operation is delayed due to outstanding accesses, but the processor continues executing past that point in the program. If the processor initiates more writes to memory, the counter will be incremented and the release will be delayed until these post-release writes have completed. The release cannot be initiated unless the counter is zero. One way to deal with this problem is to stall the processor at the first write to memory after an uninitiated release and restart the processor once the outstanding writes have completed and the release has been initiated. A higher performance solution is to have two counters. One of the counters is the current counter and all writes to memory increment that counter. When a release is encountered in the instruction stream while there are outstanding writes, all subsequent writes to memory increment the other counter. Each write to memory carries a one bit tag to indicate which counter it used, and this tag is included in the acknowledgement sent back to the processor. So the processor knows which counter

to decrement. Since the processor cannot have two release operations outstanding, two counters are sufficient. The counter used by writes to memory would simply be toggled every time an attempt is made to initiate a release. If there is a release operation waiting on outstanding writes, it can be issued once its counter goes to zero.

5.4.2 Gauss

SCCC produces a noticeable performance difference for *Gauss*. Run-time increases 15-20% compared to WO. These numbers are rather surprising at first given some of the other statistics (see Figures 5.11 - 5.13) about the behavior of *Gauss* under HWCC and SCCC which showed superior performance for SCCC in each of those statistics. It was not a surprise that HWCC had a memory latency about 50% higher than SCCC. After all, under HWCC the memory module may need to wait for responses to coherence messages. However SCCC also had a lower memory module utilization and fewer requests received by memory, indicating that the instructions added to maintain coherence were added in a very efficient manner. This demonstrates that the marking used was not overly conservative and leading to excessive memory traffic and increased run-time. The reason for the increase in run-time was found through careful instrumentation of the program. It turns out that the processors are spending millions of cycles executing extra instructions to explicitly write-back and invalidate shared-memory locations. This overhead is from all the extra instructions manually added as seen in Figures 5.8 and 5.9. However, when approaches were used with fewer added loops, then the run-time was even greater due to the excessively conservative approach producing extra messages to memory.

An increase of run-time of 15-20%, although not as good as the -2% to 1.5% performance of *Relax*, is still competitive. In addition, it should be remembered that in SCCC the cache and memory controllers are much simpler. This may provide other gains that are difficult to simulate (as mentioned above, I have already simulated a one cycle speed-up in memory access time due to the lack of a directory look-up at the memory module). The simpler controllers should be faster. In addition, cache coherence

hardware is notoriously difficult to design, and the simpler design may speed-up the time to market.

5.4.3 Higher Latency

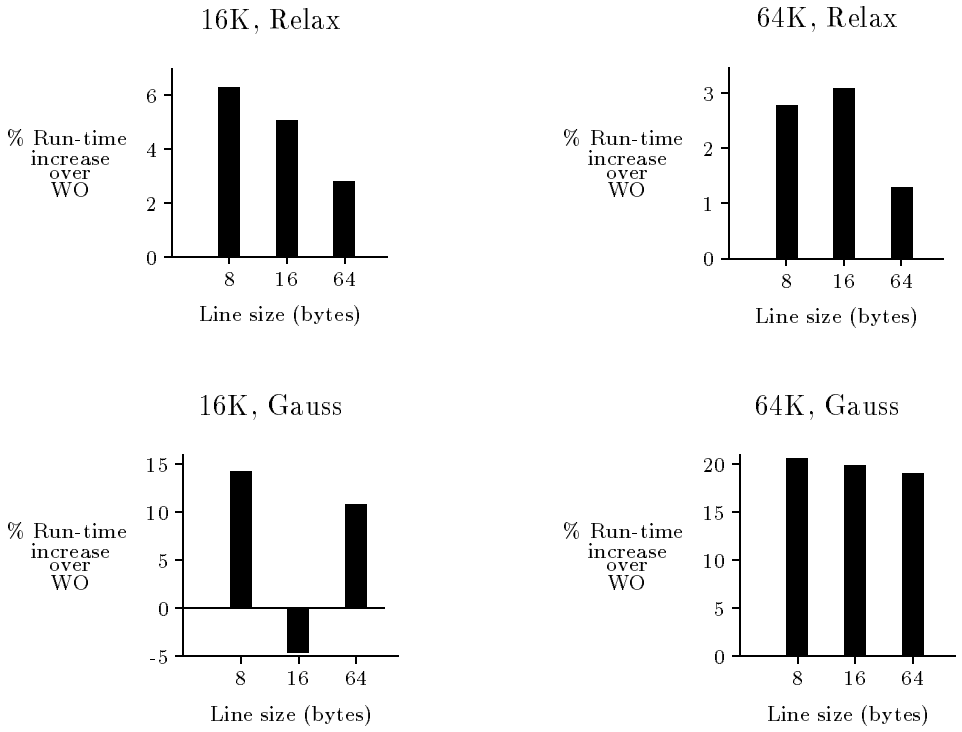


Figure 5.14: SCCC results graph, high latency

The observed memory latency in SCCC is much lower than in HWCC systems since no coherence messages need to be sent when a line is requested from memory. This raises the question of whether SCCC would perform better relative to WO if the memory latency were higher since it would have more to gain. To attempt to answer that hypothesis I increased the memory latency as best I could given the constraints of my simulator's implementation. The memory access time was increased from seven to twenty-one cycles. The network cycle time was also tripled. Due to the pipelined nature of the network this did not totally increase the network latency in the desired way, but it did increase

the network contention. The observed latencies roughly doubled. The results are in Figure 5.14.

Relax now has a slight across the board advantage when using HWCC. It is not quite clear why this is the case or if it is large enough to be significant. However, with the higher latency, the number of stalls due to all MSHRs being full more than doubled. Increasing the number of MSHRs to 100 did not produce a major change in the number of stall cycles. So it seems likely now that stalls due to full MSHRs is a source of the performance difference in this case (the number of stalls is much higher than it was for the lower latency). However, it is not clear if the stalls are all from the writes in the write-back phase of the program, or if some now are from the computation phase of the program.

Gauss produces some interesting results with higher latency. For the 64K system WO still runs about 20% faster than SCCC. The hit rates are still high. So SCCC, which has a lower memory latency, has very little opportunity to provide benefit. Instead the overhead of writing-back and invalidating dominates. For the 16K cache system though, where before WO was 15-18% faster than SCCC, with the higher latency it is now from -5 to 14% faster, a much greater spread, and one data point, for 16 byte lines, where SCCC is faster. The explanation of this range a performance is somewhat complex as it is the result of the interaction of two factors. First, *Gauss* has a tendency to write large amounts of data sequentially. This is a gain for large cache lines for HWCC. If 40 consecutive doubles are to be written, then with 64 byte lines only five requests for write access are made to memory (plus possible invalidation messages). If 16 byte lines are used, then there are 20 such requests to memory. This is part of the reason that *Gauss* has higher hit rates for larger line sizes. When hit rates are higher, it is usually to the advantage of HWCC, since SCCC has lower latency and gains when there are more messages to memory. The other factor to consider is that *Gauss* has some false sharing, a behavior which does not adversely impact SCCC the way it does HWCC (see Section 5.1.2). When 8 byte lines are used, false sharing is not a problem since *Gauss*

operates on doubles. So for *Gauss*, SCCC gains relative to HWCC when false sharing is more of a problem, but HWCC gains when the hit rate increases with large line sizes. This results in SCCC performing better for 16 byte lines, but HWCC performing better for 8 and 64 byte lines. It should always be remembered that SCCC is fighting against the overhead of the invalidation and write-back instructions.

5.5 Prior Work

There have been very few trace-driven or instruction level simulation studies comparing software controlled cache coherence to hardware coherent systems. The paucity of studies is mostly due to the difficulty of experimentation. Marking of programs for SCCC is difficult. Address traces do not normally have the information necessary for a simulation of SCCC; for example, when does a location need to be invalidated or written-back. Therefore most studies have either used numerical models or probabilistic simulations.

Owicki and Agarwal [82] compared HWCC to SCCC using an analytic model. They only compared SCCC to HWCC in a bus-based system which is really not comparable to a large-scale multiprocessor with a multistage interconnection network.

Adve et al. [2] compared SCCC to HWCC using mean value analysis [90]. Using various parameters, such as the type of access and sharing, they determined the relative performance of SCCC to HWCC systems over a range of parameter values. They found that for a large range of programs SCCC provided performance to within 10% of that of HWCC depending upon the type of accesses. Not surprisingly, the major determining factor for the system performance was how conservative the SCCC system was. However, it also depended upon the type of data that was hitting in the cache; whether it was read-mostly, migratory, write-mostly, etc. They only considered systems with one word line sizes, so they do not take into account the advantage SCCC gains dealing with the false sharing problem. Also, there is no indication in their paper that they consider the effect of increased memory latency that HWCC must deal with as I observed in *Gauss*.

Chen and Veidenbaum [33] did a trace-driven study comparing SCCC and HWCC.

They considered caches with both eight and thirty-two byte lines. So they did consider systems with line sizes larger than one word, which allowed them to study the benefit of SCCC over HWCC when there is false sharing. Chen and Veidenbaum had a very simple architecture and set of benchmarks. As described in Section 5.2.1 their program contained only DoAll loops and only synchronized using barriers. Their caches were write-through, so main memory was always up-to-date. At barriers they simply invalidated the entire cache. Their results showed that SCCC provided equivalent performance in some cases, and was not a major loss of performance in others. However, their study had several weaknesses. They did not consider the programs' run-time, only the hit ratio. So it is difficult to say what the real difference in performance was. Also, write-through caches certainly make it easier to implement SCCC, but are not the usual choice in a modern, large-scale, shared-memory multiprocessor. Finally, there are many parallel programs that do not use DoAll loops, or only do so for part of their execution. The effect of the explicit invalidations and write-backs that such programs require as well as the effect of such instructions on code density was not considered.

Min [78] used trace-driven simulation to compare a SCCC system which he proposed to other SCCC systems and to a HWCC system using directories. He found that his system had equivalent hit ratios to the HWCC system. The HWCC system had lower levels of write traffic, but higher levels of network traffic (due to invalidation messages). However, this does not tell us what the run-time is, and therefore, which system has greater performance. The study also only used programs with DoAll loops to express parallelism and the caches had a line size of only one word, neither of which is realistic.

5.6 Conclusions

The experiments in this chapter showed performance for SCCC that was no worse than 20% relative to HWCC. This was while quantifying almost no benefit from the simplification of the cache and memory controller (memory access was one cycle faster in SCCC). In many cases the performance of SCCC was similar to that of HWCC. In a few cases

the performance was slightly to noticeably better. The benefit varied depending upon the memory latency, the program and the cache size and structure. Based upon these results it would seem that SCCC is definitely an option to consider in future large-scale shared-memory multiprocessors, where the sheer system size results in excessive costs in hardware, design time and run-time overhead for supporting hardware enforced cache coherence.

The study in this chapter used only *Gauss* and *Relax* two array processing programs. They were chosen because the relative simplicity of their access patterns and the shortness of the programs made it practical to do the marking for SCCC for them manually. However, in the future a large variety of program types must be used as benchmarks. This will clearly require the use of compiler marking of programs. The lack of compiler support has been the paramount reason for the lack of good simulation study of SCCC in the general case (as opposed to programs that only use DoAll loops) and must be rectified. Also manual optimization of the marking process as I did for *Gauss* may not provide us necessarily with an accurate image of what would happen on a real system. Many compilers will not be as good as a person who knows something about the programs expected behavior. That is why tools that allow the user to give the compiler some idea about the behavior of the program (e.g., range of values for a variable, especially index variables, access patterns) are also necessary.

Chapter 6

Conclusion

In this dissertation we have seen how various features of the architecture and program can affect the performance of parallel programs. The choice of synchronization primitives and their use in high-level constructs such as barriers can have a definite impact on the execution time of parallel programs. However, the pattern of synchronization operations used by the programs is even more important. I have also demonstrated that implementing a model of memory consistency that is more relaxed than sequential consistency can yield significant performance improvement by allowing the use of architectural assists whose features that cannot be used under sequential consistency.

6.1 Synchronization

If a program's critical sections are extremely short (perhaps 100 cycles or even less) and there is high contention for locks, then the lock algorithm will become a major performance factor. However, the situation did occur in my experiments. There was high contention for locks, but the critical sections were too long and they themselves became the bottlenecks. There was too much sequential code and Amdahl's Law was the limiting factor. So although how synchronization is supported in the hardware and implemented in the software can have some performance impact, how the programmer

uses synchronization can be far more significant.

Various types of synchronization including locking have been studied using artificial benchmarks [10, 13, 56, 61, 76, 95]. Although these techniques are adequate for studying different alternatives for implementing synchronization, they only provide an "upper bound" on the efficiency of the mechanisms. The effect of different synchronization techniques on real programs, should be considered, a study I performed using trace-driven simulation. However, a comparison of the two locking algorithms I chose could easily be performed using actual runs of the programs on a real system. Future research into synchronization should be encouraged to use real systems and real programs where practical, something that is not usually done in the same study.

6.2 Relaxed Consistency Models

We have seen that sequential consistency provides a very natural memory model for the programmer. However, although SC can be efficiently implemented on a uniprocessor, in the case of multiprocessors the model excessively constrains the architect. Under SC many techniques for defraying the effects of memory latency cannot be used. Therefore, relaxed models of memory consistency were developed. Relaxed models allow the architect to use more advanced architectural techniques while allowing the programmers to still view machines as being SC if they are willing to accept the very natural restriction of data-race-free programs.

My execution-driven simulation of four benchmarks in a "dance-hall" architecture demonstrated that the use of relaxed models can provide significant performance improvements to parallel programs. The performance gain is usually inversely proportional to the cache hit rate. As memory latencies increase, relaxed models will provide even greater benefit.

I also showed that how programs are compiled for different memory models matters. A schedule of instructions that provides the best performance for a program on a weakly ordering system may provide sub-optimal performance on a sequentially con-

sistent version of the same architecture. In one of my benchmarks I uncovered a 10% range in performance according to the ability of the instruction scheduler to take advantage of features of the architecture that varied depending upon the consistency model implemented.

Relaxed models do not specify architectural features to be used. They simply permit these features to be used without violation of the memory model. A question that I addressed was what features provide the greatest benefit. It would appear that most important was lock-up free caches, which in my system allowed reads to complete while writes were outstanding (this can be allowed in different ways on other systems [50]). The next most important feature was non-blocking loads. However, they require compiler support to be effective in a statically scheduled processor. The choice of which relaxed memory model was implemented did not appear to be significant.

Software controlled cache coherence, which can only be used on a system that implements a relaxed model of consistency, is an idea for future computer systems that should be studied in more depth. I have presented a more realistic study than any previous one. It used DoAcross loops instead of just DoAll loops, simulated write-back caches with a multiword line size and was execution driven, so programs had actual invalidation and write-back instructions included. I showed that depending upon the program, cache structure and size, and the memory latency, SCCC can provide a slight performance degradation, equivalent performance, or even a performance benefit. However, I used two fairly simple, array processing programs. More complicated programs must be considered and compiler support and tools to allow for programmer guidance must be developed if SCCC is ever to be practical. A true quantification of gains from the simplification of the cache and memory controllers would also be useful information.

I have also explained why SCCC systems can obey neither sequential consistency nor most of the usual relaxed models. I have shown that my and other SCCC systems do obey the constraints of a software-centric model, data-race-free-1. DRF1 integrates the rules a programmer must obey to obtain sequential consistent execution with the

consistency model, but it provides the architect with greater freedom in his or her design and divorces the programmer from concerns about how the hardware is implemented.

Although multiprocessors have brought about huge increases in computing power/$, many programs have yet to take full advantage of these machines. Simply building machines with large numbers of processors, which have high peak performance rates is not enough [17]. We need to learn how to better use these machines and how to design them so they can be better used.

Bibliography

[1] Sarita Adve and Mark Hill. A unified formalization of four shared-memory models. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[2] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of hardware and software cache coherence schemes. In *18th Annual International Symposium on Computer Architecture*, pages 298–308, 1991.

[3] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In *1990 International Conference on Parallel Processing*, pages I–47–50, 1990.

[4] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition. In *17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[5] Sarita V. Adve and Mark D. Hill. Sufficient conditions for implementing the data-race-free-1 memory model. Technical Report #1107, Department of Computer Science, University of Wisconsin, Madison, September 1992.

[6] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *17th Annual International Symposium on Computer Architecture*, pages 104–114, 1990.

[7] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computers Systems*, 10(1):53–79, February 1992.

[8] Thomas E. Anderson. The performance implication of spin-waiting alternatives for shared-memory multiprocessors. In *International Conference on Parallel Processing 1989*, pages II–170–174, 1989.

[9] Thomas E. Anderson. The performance implication of spin-waiting alternatives for shared-memory multiprocessors. Technical Report 89-04-03, Department of Computer Science, University of Washington, April 1989.

[10] Thomas E. Anderson. The performance of spin-lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[11] James Archibald and Jean-Loup Baer. An economical solution to the cache coherence problem. In *11th Annual International Symposium on Computer Architecture*, pages 355–362, June 1984.

[12] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation. *ACM Transactions on Computers Systems*, 4(4):273–298, November 1986.

[13] Norbert S. Arenstorf and Harry F. Jordan. Comparing barrier algorithms. Technical Report 87-1-2, Department of Electrical and Computer Engineering, University of Colorado, June 1987.

[14] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91*, pages 176–186, 1991.

[15] Jean-Loup Baer and Wen-Hann Wang. Multi-level cache hierarchies: Organizations, protocols and performance. *Journal of Parallel and Distributed Computing*, 6(3):451–476, 1989.

[16] Jean-Loup Baer and Richard N. Zucker. On synchronization patterns of parallel programs. In *1991 International Conference on Parallel Processing*, pages II–60–67, 1991.

[17] Gordon Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):26–47, August 1992.

[18] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–175, March 1990.

[19] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software - Practice and Experience*, 18(8), August 1988.

[20] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, September 1991.

[21] Lothar Borrmann and Petro Istavrinos. Store coherency in a parallel distributed-memory system. In Arndt Bode, editor, *2nd European Distributed Memory Computing Conference*, pages 32–41, April 1991.

[22] Eugene D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–308, August 1986.

[23] Eugene D. Brooks III. PCP: A Parallel Extension of C that is 99% Fat Free. Technical Report UCRL-99673, Lawrence Livermore National Laboratory, 1988.

[24] Eugene D. Brooks III, Tim S. Axelrod, and Gregory A. Darmohray. The Cerberus multiprocessor simulator. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 384–390. SIAM, 1989.

[25] Eugene D. Brooks III and Joseph E. Hoag. A scalable coherent cache system with incomplete directory state. In *1990 International Conference on Parallel Processing*, pages I–553–554, 1990.

[26] David Callahan. personal communication.

[27] Nicholas Carriero and David Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computers Systems*, 4(2):110–129, May 1986.

[28] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *13th ACM Symposium on Operating Systems Principles*, pages 152–164, 1991.

[29] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[30] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990.

[31] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 1991.

[32] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, 1992.

[33] Yung-Chin Chen and Alexander V. Veidenbaum. Comparison and analysis of software and directory coherence schemes. In *Supercomputing '91*, pages 818–829, 1991.

[34] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-directed cache management in multiprocessors. *Computer*, 23(6):39–47, June 1990.

[35] David R. Cheriton, Anoop Gupta, Patrick D. Boyle, and Hendrik A. Goosen. The VMP Multiprocessor: Initial experience, refinements and performance evaluation.

In *14th Annual International Symposium on Computer Architecture*, pages 410–421, 1987.

[36] David R. Cheriton, Gert A. Slavenburg, and Patrick D. Boyle. Software-controlled caches in the VMP Multiprocessor. In *13th Annual International Symposium on Computer Architecture*, pages 366–374, 1986.

[37] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, Inc., 1991.

[38] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches. In *1988 International Conference on Parallel Processing*, pages 229–238, 1988.

[39] Gregory A. Darmohray. Gaussian techniques on shared-memory multiprocessors. Master's thesis, University of California, Davis, April 1988.

[40] Edgar W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages*. Academic Press, 1968.

[41] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[42] Michel Dubois, Jin Chin Wang, Luiz A. Barroso, Kangwoo Lee, and Yung-Syau Chen. Delayed consistency and its effects on the miss rates of parallel programs. In *Supercomputing '91*, pages 197–206, 1991.

[43] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *ACM SIGMETRICS and Performance '90, International Conference on Measurement and Modeling of Computer Systems*, pages 37–47, 1990.

[44] Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *1991 International Conference on Parallel Processing*, pages I–377–381, 1991.

[45] Susan J. Eggers and Randy H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *16th Annual International Symposium on Computer Architecture*, pages 396–406, 1989.

[46] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[47] E. Felten. personal communication.

[48] Kourosh Gharachorloo. personal communication.

[49] Kourosh Gharachorloo, Sarita Adve, Anoop Gupta, John Hennessy, and Mark Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 13(2):399–407, August 1992.

[50] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.

[51] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *1991 International Conference on Parallel Processing*, pages I–355–364, 1991.

[52] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *19th Annual International Symposium on Computer Architecture*, pages 22–33, 1992.

[53] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[54] James R. Goodman. Cache consistency and sequential consistency. Technical Report #1006, Computer Sciences Department, University of Wisconsin-Madison, February 1991.

[55] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *1990 International Conference on Supercomputing*, pages 354–368, 1990.

[56] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared memory multiprocessors. *Computer*, 23(6):60–70, June 1990.

[57] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *18th Annual International Symposium on Computer Architecture*, pages 254–263, 1991.

[58] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *1990 International Conference on Parallel Processing*, pages I–312–321, 1990.

[59] David B. Gustavson. The Scalable Coherent Interface and related standards projects. *IEEE Micro*, 12(1):10–22, February 1992.

[60] E. Hagersten, S. Haridi, and D.H.D. Warren. The cache-coherence protocol of the Data Diffusion Machine. In *Cache and Interconnect Architectures in Multiprocessors*, pages 165–188. Kluwer Academic Publisher, Norwell, Mass, 1990.

[61] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–18, February 1988.

[62] M. Johnson. Super-scalar processor design. Technical Report CSL-TR-89-383, Stanford University, June 1989.

[63] Simon Kahan and Larry Ruzzo. Parallel quicksand: Sorting on the sequent. Technical Report 91-01-01, Department of Computer Science, University of Washington, January 1991.

[64] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, Inc., 1992.

[65] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy consistency for software distributed shared memory. In *19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.

[66] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *18th Annual International Symposium on Computer Architecture*, pages 43–53, 1991.

[67] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium on Computer Architecture*, pages 81–87, June 1981.

[68] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages*, July 1981.

[69] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[70] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[71] Edward D. Lazowska. personal communication.

[72] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *17th Annual International Symposium on Computer Architecture*, pages 27–37, 1990.

[73] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[74] Dan Lenoski, James Laudon, Luis Stevens, Truman Joe, Dave Nakahira, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *19th Annual International Symposium on Computer Architecture*, pages 92–103, 1992.

[75] Brian Lockyear. personal communication.

[76] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computers Systems*, 9(1):21–65, February 1991.

[77] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[78] Sang-Lyul Min. *Memory Hierarchy Management Schemes in Large Scale Shared-memory Multiprocessors*. PhD thesis, University of Washington, 1989.

[79] Haim E. Mizrahi, Jean-Loup Baer, Edward D. Lazowska, and John Zahorjan. Introducing memory into the switch elements of multiprocessor interconnection networks. In *16th Annual International Symposium on Computer Architecture*, pages 158–166, 1989.

[80] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetch in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[81] Brian W. O'Krafka and A. Richard Newton. An empirical evaluation of two memory-efficient directory methods. In *17th Annual International Symposium on Computer Architecture*, pages 138–147, 1990.

[82] Susan Owicki and Anant Agarwal. Evaluating the performance of software cache coherence. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, 1989.

[83] G. F. Pfister and et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *International Conference on Parallel Processing 1985*, pages 764–771, 1985.

[84] Ridge Computers. *Ridge 32 User's Guide*.

[85] Christoph Scheurich and Michel Dubois. Correct memory operations of cache-based multiprocessors. In *14th Annual International Symposium on Computer Architecture*, pages 234–243, 1987.

[86] Zary Segall and Larry Rudolph. Dynamic decentralized cache schemes for an mimd parallel processor. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[87] 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.

[88] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Real-Time Signal Processing IV*, volume 298, pages 241–248. Society of Photo-Optical Instrumentation Engineers, 1981.

[89] M. Upton, K. Samii, and S. Sugiyama. Integrated Placement for Mixed Standard Cell and Macro-Cell Designs. In *Proceedings of the 27th Design Automation Conference*, 1990.

[90] Mary K. Vernon, Edward D. Lazowska, and John Zahorjan. An accurate and efficient performance analysis technique for multiprocessor snooping cache consistency protocols. In *15th Annual International Symposium on Computer Architecture*, pages 308–315, 1988.

[91] David B. Wagner. Conservative Parallel Discrete-Event Simulation: Principles and Practice. Technical Report 89-09-03, Department of Computer Science, University of Washington, September 1989.

[92] David B. Wagner. The Design of an Object-Oriented Parallel Simulation Environment. In *SCS Multiconference on Object-Oriented Simulation*, 1991.

[93] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *16th Annual International Symposium on Computer Architecture*, pages 273–280, 1989.

[94] C. Wittenbrink, A. K. Somani, and C. Chen. Cache write generation for high performance parallel processing. Revised and submitted to *ACM Transactions on Computers Systems*.

[95] Philip J. Woest and James R. Goodman. An analysis of synchronization mechanisms in shared-memory multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 152–165, April 1991.

[96] Richard N. Zucker. A study of weak consistency models, March 1991. Ph.D. dissertation proposal, Department of Computer Science and Engineering, University of Washington.

[97] Richard N. Zucker and Jean-Loup Baer. A performance study of memory consistency models. In *19th Annual International Symposium on Computer Architecture*, pages 2–12, 1992.

[98] Richard N. Zucker and Jean-Loup Baer. A performance study of memory consistency models. Technical Report 92-01-02, Department of Computer Science and Engineering, University of Washington, March 1992.

Appendix A

Glossary

A.1 Initiated, Issued and Performed

These definitions come from Dubois et al. [41, 85].

Initiated - a request is *initiated* when a processor has sent the request and completion of the request is out of its control.

Issued - An initiated request is *issued* when it has left the processor environment (which includes the CPU and local buffers) and is in transit in the memory system.

Performed - A LOAD by processor I is considered *performed* with respect to processor K when issuing of a STORE to the same address by processor K cannot affect the value returned to processor I. A STORE by processor I is considered *performed* with respect to processor K, at a point in time when an issued LOAD to the same address by processor K returns the value defined by the STORE. An access by processor I is *performed* when it is performed with respect to all processors. A STORE is *globally performed* when it is performed with respect to all processors. A LOAD is *globally performed* if it is performed with respect to all processors and if the STORE which is the source of the returned value has been globally performed.

A.2 Processor Consistency, Original Definition

This is the original definition of *processor consistency* and comes from Goodman [54]:

> A multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program.

So operations of one processor, as seen by two other processors, may be in different orders.

Appendix B

Sufficient Conditions for DRF0 and DRF1

B.1 Conditions for DRF0

Adve and Hill [4] show that the five conditions below are sufficient conditions for hardware to be weakly ordered with respect to DRF0. However, some definitions are first needed. These definitions and the five conditions are taken verbatim from Adve and Hill [4].

> A *commit point* is defined for every operation as follows. A read commits when its return value is dispatched back towards the requesting processor. A write commits when its value could be dispatched for some read. A read-write synchronization operation commits when its read and write components are globally performed. We will say that an access is *generated* when it "first comes into existence".
>
> Hardware is weakly order with respect to DRF0 if it meets the following requirements.
>
> 1. Intra-processor dependencies are observed.

2. All writes to the *same* location can be totally ordered based on their commit times, and this is the order in which they are observed by all processors.

3. All synchronization operations to the *same* location can be totally ordered based upon their commit times, and this is also the order in which they are globally performed. Further, if S_1 and S_2 are synchronization operations and S_1 is committed and globally performed before S_2, then all components of S_1 are committed and globally performed before any in S_2.

4. A new access is not generated by a processor until all its previous synchronization operations (in program order) are committed.

5. Once a synchronization S by processor P_i is committed, no other synchronization operations on the *same* location by another processor can commit until after all reads of P_i before S (in program order) are committed and all writes of P_i before S are globally performed.

B.2 Conditions for DRF1

Adve and Hill [5] show that a set of conditions, which I refer to as the DRF1 Condition, are sufficient conditions to show that hardware obeys DRF1. The terminology used and conditions are explained in more detail in Section 5.2.2.

DRF1 Condition: Hardware satisfies Condition 2.1 (defined earlier in Adve and Hill [5]) and therefore obeys the data-race-free-1 memory model if for every execution, E_{drf}, of a program, *Prog*, on the hardware, there is an $\xrightarrow{xo}$, that satisfies the following conditions:

1. *Data* - Let Rel and Rel' be release operations and Acq and Acq' be acquire operations. Let Z be any operation. Let X and Y be conflicting operations such that at least

one of X or Y is a data operation.

(a) *Release-Acquire* -

i. If $Rel \xrightarrow{\text{so1}} Acq$, then $Rel\ (i) \xrightarrow{\text{xo}} Acq\ (j)$ for all i, j.

ii. If $Z \xrightarrow{\text{po}} Rel' \xrightarrow{\text{so1}} Acq' \xrightarrow{\text{po}} Rel \xrightarrow{\text{so1}} Acq$, then $Z(i) \xrightarrow{\text{xo}} Acq\,(j)$ for all i, j.

(b) *Post-Acquire* -

i. If $Acq \xrightarrow{\text{po}} Z$, then $Acq\,(i) \xrightarrow{\text{xo}} Z(j)$ for all i, j.

ii. If $X \xrightarrow{\text{po}} Rel \xrightarrow{\text{so1}} Acq \xrightarrow{\text{po}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i.

(c) *Intra-processor* - If $X \xrightarrow{\text{po}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i.

2. *Synchronization* - Let X, Y and Z be synchronization operations.

(a) If $Y \xrightarrow{\text{po}} Z$, then $Y(i) \xrightarrow{\text{xo}} Z(j)$ for all i, j.

(b) If X is a write operation, Y is a read operation that conflicts with X and $X \xrightarrow{\text{so0}} Y \xrightarrow{\text{po}} Z$, then $X(i) \xrightarrow{\text{xo}} Z(j)$ for all i, j.

(c) If X and Y are conflicting write operations, then either $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i or $Y(i) \xrightarrow{\text{xo}} X(i)$ for all i.

3. *Control* -

(a) Let read R control an operation X or determine the value that X writes (if X is a write). Then $R(i) \xrightarrow{\text{xo}} X(j)$ for all i, j.

(b) Consider any sequentially consistent execution, E_{sc}, of *Prog* and operations X and Y such that $X \xrightarrow{\text{po}} Y$ and either X and Y conflict, or X is an acquire, or Y is a release, or X and Y are synchronization operations in E_{sc}. Let operation X not be executed in E_{drf} and operation Y be executed in E_{drf}. Let read R control operation X in E_{sc} and let R be one of the reads in E_{drf} whose value determined that X would not be executed in E_{drf}. Then $R(i) \xrightarrow{\text{xo}} Y(j)$ for all i, j.

(c) If $X \xrightarrow{\text{po}} Y$ and X is an acquire, then $X(i) \xrightarrow{\text{xo}} Y(j)$ for all i, j.

(d) Let X and Y be synchronization operations. If $X \xrightarrow{\text{po}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(j)$ for all i, j.

Appendix C

Detailed Statistics

These tables contained more detailed statistics than those presented in Table 4.4.

Table C.1: Benchmark statistics for reads for SC1 for 16K and 64K caches
Reads are averages per processor and are in 1,000's.

Program	Reads	Hit Rate (%) by line and cache size					
		16K cache			64K cache		
		8 bytes	16 bytes	64 bytes	8 bytes	16 bytes	64 bytes
Gauss	1074	75.9	87.0	96.1	95.8	97.5	98.9
Qsort	1171	73.3	76.4	84.0	75.7	78.2	84.3
Relax	2132	87.2	93.4	98.1	88.1	94.0	98.4
Psim	1827	92.8	92.9	94.0	93.8	93.8	94.6

Table C.2: Benchmark statistics for writes for SC1 for 16K and 64K caches
Writes are averages per processor and are in 1,000's.

Program	Write	Hit Rate (%) by line and cache size					
		16K cache			64K cache		
		8 bytes	16 bytes	64 bytes	8 bytes	16 bytes	64 bytes
Gauss	314	21.1	59.2	88.3	96.4	97.1	98.2
Qsort	310	58.8	64.0	73.5	64.1	67.8	74.7
Relax	329	24.6	62.1	89.9	24.6	62.1	90.0
Psim	319	62.6	64.0	68.4	64.6	65.7	69.4

Table C.3: Read and write frequency for SC1 for 16K and 64K caches with 16 byte lines
Frequencies are the number of cycles between references on average.

Program	Cycles between references			
	16K caches		64K caches	
	Reads	Writes	Reads	Writes
Gauss	19.6	70.0	15.4	52.8
Qsort	16.1	59.5	15.5	57.6
Relax	12.8	83.5	12.7	82.9
Psim	16.0	92.0	15.8	90.8

Appendix D

Proof - Chen and Veidenbaum Obey DRF1

Chen and Veidenbaum's simulated system does obey DRF1, which will be shown in the next several paragraphs. The processor they simulated was a MIPS R3000/3010 [64], which does issue instructions in order. The reader should keep this in mind as this is an important feature which simplifies proving that their system obeys DRF1 as is the fact that all synchronization was done using barriers. I will show that their system obeys the DRF1 Condition, and hence, obeys DRF1.

It should be noted that Chen and Veidenbaum do not specify how barriers are implemented. I assume that there is a barrier instruction with an implicit release and then an acquire. If barriers are implemented in software using a counter or some other algorithm, then when showing that Chen and Veidenbaum's system obeys DRF1, the individual synchronization operations (which are not described) must be considered when the DRF1 Condition refers to an access as being a synchronization operation.

Data Requirement

1.a(i) is obeyed trivially because synchronization variables are not cached. There is only one copy of any synchronization variable, the memory copy, so there is no question of

when the other copies are updated.

1.a(ii): Z (i) completes for each i when the ith processor invalidates its cache before entering the barrier of which *Rel'*is a part. At that point in time, Z (i) becomes the memory copy, which is up-to-date since write-through caches are used. Once every processor has entered the barrier, Z is complete, which is before Acq.

1.b(i): There is no caching of synchronization variables, so once Acq is performed by the issuing processor, *Acq (i)* for all i is done. Since the processor will stall until an acquire completes, Z will not be issued until after the acquire is performed, and therefore if $Acq \xrightarrow{\text{po}} Z$, then *Acq (i)* $\xrightarrow{\text{xo}} Z(j)$ for all i, j.

1.b(ii): $X(i)$ for all i is true when each processor has invalidated its cache before entering the barrier. Since each processor will have entered the barrier before any processor can leave the barrier, X completes before Y can be issued.

1.c: Because of data dependences, the processor that executes X and Y will execute them in order. If one of X and Y is a read, then X (i) $\xrightarrow{\text{xo}}$ Y (i) for all i is only defined for i equal to the issuing processor and the condition holds. If both X and Y are writes, it is sufficient to show that it is not the case that Y (i) is completed and then X (i) completes for any i. Since the caches are write-through, the write of X to memory is completed before the write of Y (there is no bypassing in the network). That leaves only the cached copies of the location that X and Y reference. If a processor caches the value written by Y, it could only have done so by reading that location from main memory. If it read that location from main memory, Y must have completed at main memory, and there is no way for the value written by X to be subsequently cached by that processor.

Synchronization Requirement

2.a: Since synchronization variables are not cached, there is only one copy of each. Since a processor must complete one barrier before entering a second one, two barriers will always complete in program order.

[5] says that 2.b and 2.c are obeyed automatically by any system that does not cache

synchronization variables. So no separate proof of these two sub-conditions is necessary.

Control Requirement

3.a is obeyed because the processor used obeys uniprocessor dependencies. If R controls X, then there is a data or control dependency, and the processor must complete R before X is issued, and hence before X (j) is performed for any j.

3.b: For the same reasons that 3.a is obeyed, this condition is obeyed. There is a data dependence caused by the testing of the value that R reads, and hence R completes before X would be issued if it were executed, and hence, before Y is issued.

According to [5] Conditions 3.c and 3.d are met by the data and synchronization conditions of the DRF1 Condition. They were included for completeness since alternate data or synchronization conditions may not automatically obey 3.c and 3.d. So there is no need to show that the system obeys them.

Vita

Richard N. Zucker was born in New York City on March 31, 1960. After graduating from Stuyvesant High School, he attended Union College in Schenectady, New York. He received a Master of Science and Bachelor of Science degrees from Union in December 1982. He then worked as a software engineer for The BDM Corporation until 1986, where upon he started his doctoral studies at the University of Washington, which he completed in 1992.