On Synchronization Patterns in Parallel Programs

Jean-Loup Baer and Richard N. Zucker
Department of Computer Science and Engineering
University of Washington

Technical Report No. 91-04-01
April 1991

On Synchronization Patterns in Parallel Programs*

Jean-Loup Baer and Richard N. Zucker
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Abstract

Efficient synchronization is a key element in obtaining good speed-up from parallel programs. The overhead introduced by synchronization, especially lock manipulation, can sometimes remove any benefit from parallelizing programs. Techniques to efficiently obtain locks under high contention have been studied in the literature using artificial programs. We consider the impact of these techniques in a more realistic framework using a sample of real parallel programs running on a shared-bus multiprocessor system. Cycles lost to lock contention and the number of processors waiting to acquire a lock are the two principal metrics that we use. Trace-driven simulation experiments are performed for sequentially consistent and weakly consistent architectural models.

1 Introduction

Shared-memory multiprocessors have become much more prevalent in recent years. There are several commercially available systems with multiple CPU's such as those produced by Sequent, Encore, Silicon Graphics and Alliant. These machines can provide a significant performance increase for programs that have large portions of their execution that can proceed in parallel. Assuming that a program can be parallelized, there are still potential bottlenecks to achieving the maximum possible speed up on a parallel machine. One major source of the bottlenecks has to do with synchronization. While synchronization is most often required by the algorithm, the implementation of the synchronization primitives and, more specifically, the use of locks to guarantee mutual exclusion could be a determinant factor in the degradation of the speed-up.

The efficient implementation of locks in hardware (in the context of shared-bus systems) and software has been studied before [3],[12]. Locking schemes that can deal with high contention for locks very well, given the right hardware support, were developed. The efficiency of these locking schemes was tested using synthetic programs designed to have high amounts of lock contention.

*This work was supported by NSF Grants CCR-8904190 and CCR 8702915.

However, that research did not deal with real parallel programs. It is not clear, therefore, whether the extra hardware and/or software sophistication is justified. The purpose of this paper is to investigate the impact of lock manipulation in a more realistic environment.

We will therefore examine various aspects of the locking behavior of several parallel programs. Some of these aspects are architecturally dependent, e.g., the overhead incurred in the lock acquisition, while others, e.g., the number of processors waiting to acquire a lock, depend mostly on the algorithm and programming paradigm unless locks are implemented so poorly that their overhead overshadows that of contention for the lock. We study how these, and related, factors impact execution time and processor utilization in a shared-bus multiprocessor system. Two different locking schemes, a very efficient one (queuing locks [12]) and the more usual one (test-and-test-and-set [17]), will be tested. We will also perform experiments using a weakly consistent memory access model.

The paper is organized as follows. In section 2 we cover the methodology we used when collecting our information and discuss the benchmarks selected and their characteristics. In section 3 we present our results and discuss some of their implications. In section 4 we consider the impact of the memory access model. Concluding remarks are given in section 5.

2 Methodology

2.1 Traces

Our studies were done using trace-driven simulation. This performance evaluation tool allows us to contrast a number of different architectural variations (e.g., consistency model) as well as to assess the effect of changes in system parameters (e.g., bus and memory cycle times). Since the latter parameters did not modify the general trends of our results, we will not consider them further. A very important aspect of the trace-driven simulation, for the purposes of this study, is that we are able to analyze the "ideal" behavior of the traced programs, i.e., we can determine how long any section of the program would take given no interference from other programs or stalling due to cache misses.

We used traces of programs running on a Sequent Symmetry Model B with 20 Intel 80386 processors. These traces were collected using the MPTrace system [9]. 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. How locks are simulated is described below in section 2.4.

2.2 Model architecture

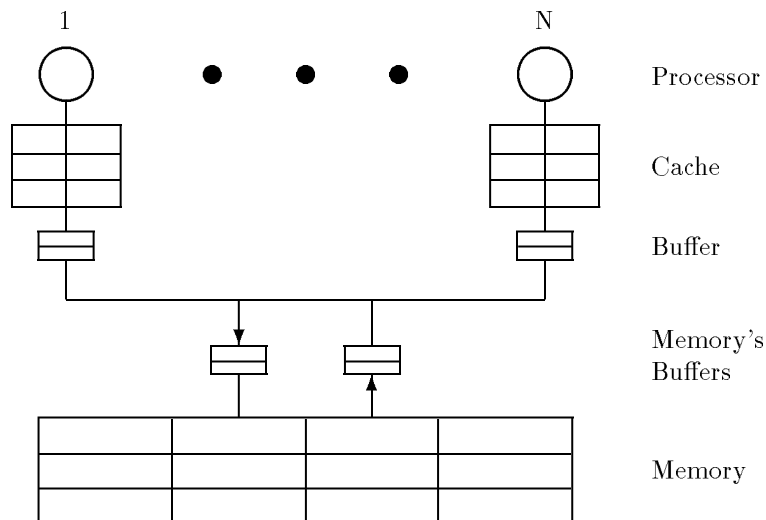

Figure 1: Model Architecture

We simulated a bus-based architecture similar to the Sequent Symmetry Model B (cf. Figure 1) 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 [4]. 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. The buffer depth was initially set to four which is larger than the usual write buffer in a system with a write-back cache. This was in anticipation of the larger buffer requirements of a weakly consistent architecture [7]. The wisdom of such a choice is discussed in section 4. 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 may arrive at the memory while a previous request is being processed. Hence our 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 we have a two element buffer at the memory output.

The memory has an access time of three cycles. So, 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 Benchmarks

The programs that we simulated include VLSI CAD tools and scientific programs written in either C or C++. The C++ programs (the first three in table 1), were written using the Presto [5] 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 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 [11]. The program trace ran for three timesteps of evolution for a system of 2000 stars. Pdsa [18] does topological optimization using simulated annealing. FullConn is a run of a Synapse [19] distributed simulation of a fully-connected processor network.

The three C programs are Pverify, Qsort, and Topopt. Pverify [8] 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 [8] 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 [13] 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 1 and 2 list "ideal" (in the sense given at the beginning of this section) 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 is allocated as shared even when it need not be. This is reflected in the numbers given. 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 with no change in the basic results we present.

Program	# of Proc.	Work Cycles	References		
			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 1: Benchmark Ideal Statistics
Cycles and references are averages per processor and are in 1000's.

Program	Lock Pairs	Nested Locks	Avg. Held	Total Held	% of 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

Table 2: Benchmark's Ideal Lock Statistics
Lock pairs and nested locks are averages per processor.
The average held and total held are in cycles.

In table 2 the "ideal" data on locking patterns as taken from the traces is given. The column "Nested locks" refers to when a lock is locked while another lock, the 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. Therefore, for Grav, Pdsa and FullConn we 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.

2.4 Locking

We want to study the effect of locking algorithms on real programs from two different, and complementary, viewpoints: the effect of lock implementation on the overall system and the pattern of locking in parallel programs. To remove as many artifacts as possible for the study on locking patterns, we need a locking algorithm that causes minimal interference with processors doing useful work. For this reason, we have chosen to simulate a scheme similar to queuing locks [12].

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. In the scheme implemented on our simulator, when a processor wants to acquire a lock, a memory access is made. When the result of that access returns to the processor, it sees whether or not it has the lock. If so, it enters the critical section. Otherwise it 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 we are 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. We used the slightly more efficient scheme to minimize the implementation constraints. With the results that we have generated so far, we believe that the two missing bus transactions have no impact on the validity of our results as applied to queuing locks. We are currently modifying our simulator to verify this assumption.

In order to assess the importance of a sophisticated implementation of lock manipulation, we simulated the same traces using the more mundane test-and-test-and-set (T&T&S) primitive. In this scheme the value of the lock variable is read. If it is locked, then the processor spins by reading this value until it is free. Since a copy of the lock variable is in the processor's cache, the spinning does not consume any bus bandwidth. When the lock is free, the change in value in the lock variable is seen by the waiting processor via hardware enforced cache coherence. The processor then does an atomic test-and-set to get the lock. If several processors are spinning, there will be a burst of traffic as all the processors try to get the lock after it has been freed. The ones that fail will still cause a great deal of bus activity, which may slow down the processor that has the lock [3],[12].

3 Results

3.1 Queuing Lock Implementation

In this section we present the results that we found when we examined the behavior of our benchmarks using our approximation to queuing locks. We are 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. We would also like to see which "ideal" statistics (e.g., number of lock pairs, the times locks are held) are good predictors of lock contention.

The basic statistics that we collected from the simulation of our benchmarks are summarized in table 3. The run-time for Topopt is somewhat skewed compared to the numbers in table 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 3, the programs with the largest number of lock acquisitions (cf. table 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

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

Table 3: Benchmark Runtime Statistics: Queuing Lock Implementation
The stall causes are the percent of stalls caused by that event.

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 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 4. We give 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 time. The percentage of time that locks are held is not a predictor of locking behavior. 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.

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

Table 4: Lock Contention Statistics: Queuing Lock Implementation
Lock holding times are averages in cycles.

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 [2]. 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. Grav and Pdsa, with their poorer behavior, were written as part of a ten week seminar.

3.2 Importance of the Lock Implementation

Tables 5 and 6 show the same statistics as tables 3 and 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.

Naturally, there won't be any major difference in the behavior and run-time of programs with low lock acquisitions, i.e., the last four 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 T&T&S than when using queuing locks and Pdsa takes 8.1% longer. In looking at table 6 we see that locks in those two programs are not held significantly longer than when using queuing locks nor are there

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 5: Benchmark Runtime Statistics: T&T&S
The stall causes are the percent of stalls caused by that event.

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

Table 6: Lock Contention Statistics: T&T&S
Lock holding times are averages in cycles.

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 vs. 1.2-1.5 cycles for the queuing lock scheme that we 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 T&T&S 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 T&T&S see [3]). However, the increased bus contention does have an overall impact. The bus utilization for Grav doubled when using T&T&S 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 significantly reduce 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.

4 Weak Memory Access Model

As the speed of processors increases faster than the speed of access to main memory, the relative cost of a cache miss increases. This is especially true in large scale multiprocessors that use a multi-stage interconnection network between the processors and memory. Several ideas have been put forward to reduce the frequency with which the full penalty of a cache miss must be paid. These ideas have focused on ways to change the programmer's model of memory by relaxing the hardware constraints of having to enforce sequential consistency [16]. These alternative memory models include weak ordering [7]. It is important to know whether the additional hardware features required for an efficient implementation of weak ordering, e.g., buffers, are warranted in terms of improved execution time. To that effect, we simulated a machine similar to the one presented in Section 2 but where the memory model implemented by hardware is that of weak ordering. In subsection 4.1 we describe what weak ordering is. In subsection 4.2 we present our simulation results.

4.1 Weak Ordering

Most multiprocessor systems are sequentially consistent. Lamport [16] says:

> [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.

Thus, in a sequentially consistent system the end result of the program must be as if each instruction had finished executing before the next instruction is started and the instructions are executed in program order. If this is not the case, then certain algorithms, such as Dekker's Algorithm for mutual exclusion [6] may not work correctly (an example of what could happen can be found in [7]).

Weak ordering is defined as [7]:

> 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 [1]). Performed is defined formally in [7]. 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 that the value to be returned by a LOAD has been set and cannot be changed.

In a weakly ordered system, the only restrictions on the access of normal, i.e., non-synchronizing, memory references is that imposed by the program's own data dependencies [15]. Otherwise 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 precedence 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. However, according to the second and third rules of weak ordering, whenever a synchronization operation is performed all references to shared data must complete and no new references to shared data may be made until the synchronization has completed. This means some

explicit synchronization operation visible to the hardware, such as a Test&Set, must be used to obtain a lock.

In our simulation we model the bypassing of accesses in the buffers between the caches and the bus. 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. Those references are loads and instruction fetches (this could also be done in a sequentially consistent system for the instruction fetch, but it is more complicated). 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 our model we neither prefetch, nor do out of order instruction issue or completion, nor delay 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, the prefetching strategy, either hardware or software based, is fairly elaborate and cannot be included in our trace information. Similarly, modeling out of order instruction issue or completion is not possible given the traces we have. No instruction types are given, only the number of cycles for the instruction to complete. A different methodology would have to be used for assessing the performance of superscalar or multi-functional units processors. This is outside the scope of this paper.

The delaying of invalidation signals would work if we had a single-word line size since both processors would be writing to the same word. The first invalidation 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 first written to be lost. Since we use a multi-word line size in our model, we do not delay 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.

4.2 Weak Ordering Results

Tables 7 and 8 summarize the results from running our benchmarks assuming a weakly ordered memory system. The column difference (%) shows the percent decrease 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 we model, 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 7, 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 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 3) and are much more frequent than write misses since the reads almost always precede the exchanges (writes) of the same lines.

As can be seen by comparing table 8 with table 4, there is no significant difference in the patterns of locking using the two memory models. Although the overall runtime was not significantly different, there was a possibility that locking would be different. In a weakly ordered system a processor must stall at every synchronization point to let all shared accesses complete. We found that there were almost never any uncompleted shared accesses when a lock or unlock was done. Therefore, it is debatable whether cache-bus buffers should be as deep as those we simulated.

Given these results it does not seem as if weak ordering is worth implementing in a shared-bus system such as the one we are simulating. One cannot forget that there would be extra costs involved in implementing weak consistency. 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 [14] 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.

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 7: Weak Ordering Runtime Statistics
Difference is the difference between these numbers and those in table 3

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

Table 8: Weak Ordering Lock Contention Statistics

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 [10]. 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 we have already seen that there are not enough writes to make a significant difference.

5 Conclusions

Efficient synchronization is a key element in obtaining good speed-up from parallel programs. In this paper we have shown that the number of lock acquisitions in the "ideal" analysis is the best

predictor of the level of contention to get a lock. The percentage of time that locks are held during the running of the program is inconsequential. Also, as has been shown with artificial programs, the choice of a better lock algorithm like queuing locks can make a significant difference in program performance.

Given the shared-bus architecture used in these studies, a weakly ordered model of memory does not provide any significant performance benefit over a sequentially consistent one. It may well be that the overhead of a weakly ordered system would outweigh its benefit. However, studies need to be done on multi-stage interconnection networks. Superscalar architectures may also show a benefit from weakly ordered memory systems.

Acknowledgements

We would like to thank Eric Koldinger for supplying most of the traces and answering innumerable questions about them and MPTrace. Also, Raj Vaswani explained the inner workings of Presto as did Ed Felten for Grav, and Simon Kahan for Qsort.

References

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition and Some Implications. Technical Report #902, Department of Computer Science, University of Wisconsin, Madison, December 1989.

[2] 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. Technical Report 90-04-02, Department of Computer Science, University of Washington, April 1990.

[3] 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.

[4] 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.

[5] 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.

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

[7] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[8] S. J. Eggers and R. H. Katz. Evaluating The Performance of Four Snooping Cache Coherence Protocols. In *16th Annual International Symposium on Computer Architecture*, pages 2–15, 1989.

[9] 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.

[10] K. Gharachorloo et al. Memory Consistency and Event Ordering In Scalable Shared-memory Multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[11] E. Felten. personal communication.

[12] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared Memory Multiprocessors. *Computer*, 23(6):60–70, June 1990.

[13] 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.

[14] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *8th Annual International Symposium on Computer Architecture*, pages 81–87, June 1981.

[15] 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.

[16] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transaction on Computers C-28*, C-28(9):690–691, September 1979.

[17] Z. Segall and L. Rudolph. Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[18] 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.

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