A Performance Study of Memory Consistency Models

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

Technical Report No. 92-01-02
January 1992

A Performance Study of Memory Consistency Models*

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

Abstract

Recent advances in technology are such that the speed of processors is increasing faster than memory latency is decreasing. Therefore the relative cost of a cache miss is becoming more important. However, the full cost of a cache miss need not be paid every time in a multiprocessor. The frequency with which the processor must stall on a cache miss can be reduced by using a relaxed model of memory consistency.

In this paper, we present the results of instruction-level simulation studies on the relative performance benefits of using different models of memory consistency. Our vehicle of study is a shared-memory multiprocessor with processors and associated write-back caches connected to global memory modules via an Omega network. The benefits of the relaxed models, and their increasing hardware complexity, are assessed with varying cache size, line size, and number of processors. We find that substantial benefits can be accrued by using relaxed models but the magnitudes of the benefits depend on the architecture being modeled, the benchmarks, and how the code is scheduled. We did not find any major difference in levels of improvement among the various relaxed models.

1 Introduction

Recent advances in technology are such that the speed of processors is increasing faster than memory latency is decreasing. Therefore the relative cost of a cache miss is becoming more important, since processors are left idle for a greater number of cycles. This performance loss is even more noticeable in multiprocessors because of the higher miss rates [11] and because memory access times are increased by the latency of and contention for the interconnect that links processors and memory modules.

We can alleviate the high cost of cache misses by reducing the frequency with which the processor must stall on a cache miss. In the case of multiprocessors, one approach is to use relaxed models of consistency. Under such models, the hardware is not required to stall every time there is an outstanding memory reference. However, for correct executions parallel programs must not exhibit data races and primitive synchronization operations must be visible to the hardware. In return the system appears sequentially consistent [21] and is generally faster than a truly sequentially consistent machine.

In this paper we present the results of simulation studies of shared-memory multiprocessors that indicate the relative performance benefits of various implementations of systems that are sequentially consistent, weakly ordered, and release consistent. Our studies show that relaxed models of consistency provide performance improvements from 1% to 36% over sequentially consistent systems. The magnitude of improvement depends mostly upon the benchmark and cache parameters (line and cache sizes). The single best predictor of the level of benefit provided by the relaxed models is the cache hit rate, but it is not always an accurate

*This is a more complete version of a paper that appeared in the 19th International Symposium on Computer Architecture.

†Supported by a DARPA/NASA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland

This work was supported in part by NSF Grants CCR-9101541, CCR-8904190 and CCR-8702915.

predictor. We also observed that programs may need to be written or compiled differently to obtain the highest performance on machines with different memory models.

Section 2 defines the various memory models, gives the rules that these models impose on programmers, and indicates how more freedom is allowed in the sequencing of memory accesses as the models become more relaxed (for a more in depth explanation of memory models, see [2]). The simulated architecture, the choices made in the implementations of the various models, and the selected benchmarks are described in section 3. In section 4 we present the results of our study. We analyze both the characteristics of the benchmarks for which there is clearly a performance gain in using a relaxed memory model and the hardware features that yield the most benefits. In section 5 various architectural variations are considered and results presented as well as compile-time improvements. Section 6 compares our results to a previous study of relaxed consistency models based on a simulation of DASH [14]. As will be seen, our results are qualitatively in agreement with those of that previous study. The quantitative differences can be explained by the architectural features being simulated. Section 7 concludes with an assessment of the performance gains that can be exploited by building shared-memory multiprocessors with relaxed models of memory consistency.

2 Consistency Models

2.1 Sequential Consistency

Memory models impose rules on programs which must be observed in order to ensure correct executions. On a uniprocessor, the implicit model is that of sequential consistency (SC), which is so intuitive that programmers who implicitly follow it do not even realize that they are adhering to a model's rules. SC is defined as [21]:

> [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.

In a sequentially consistent multiprocessor the result of the program must be the same as if each shared access had completed before the next shared access is started; these accesses are executed in program order. If this order is not maintained, then communication between processes, that is, synchronization, using loads and stores (e.g. Dekker's Algorithm [9]) may not work correctly.

Until recently sequential consistency had been the usual model of memory access for multiprocessors. However, SC's strict ordering rules can lead to performance inefficiencies. For example, there is (very) limited use of write buffers and the execution of memory accesses must be in program order. Because of SC's restrictions, other memory access models have been proposed. In the following sections, we review those models that we will consider in our performance evaluations.

2.2 Weak Ordering

In systems using relaxed memory models, not all memory accesses need to be sequentially consistent; only the synchronization operations do, which allows for greater freedom in the ordering of memory accesses. The first proposed relaxed memory model was weak ordering, which is defined in [10] as:

1. accesses to global synchronizing variables are strongly ordered and
2. no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed and
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 are examined in [1]). The formal definition of "performed" is given in [10]. Basically a store is performed when the value stored by the processor executing the instruction can be seen by all other processors. A load is performed when the value to be returned by the load has been set and cannot be changed.

Under this weak ordering model, writing parallel programs, already a difficult task, becomes even more difficult. However, it has been shown that weakly ordered hardware can appear sequentially consistent to programs with minimal constraints [2], only two of which are significant: programs must not contain data races and all synchronization operations must be visible to the hardware. The first condition is not much of a restriction since a data race is usually an error in a sequentially consistent program (chaotic relaxation is an exception). The second condition is not a significant restriction either since most multiprocessors already provide hardware support for synchronization (e.g., Test-and-Set), and these primitives are those generally used for synchronization.

In a weakly ordered system there are minimal restrictions on on the order in which the hardware can perform non-synchronizing memory references. The access order must obey the program's own data and control dependencies. Also, the second and third rules of weak ordering say that whenever a synchronization operation is performed, all references to shared data must be performed and no new references to shared data may be issued until the synchronization has completed (this explains why in [2] it was said that programs on weakly ordered machines must use synchronization operations that are visible to the hardware). Except for these restrictions, the accesses may be performed in any order. Thus an order that maximizes system performance may be used. This relaxation of access order lets reads take precedence over writes, memory references overlap, data be prefetched, instructions issue or complete out of order, and cache coherence invalidation signals be delayed until after the write to a line in the cache has been performed.

2.3 Release Consistency

In [16], Gharachorloo et al. proposed the release consistency model. As in weak ordering, the release consistency model has minimal restrictions on the ordering of the performing of normal accesses. There are some restrictions on the ordering of synchronization accesses.

In release consistency synchronizing accesses are categorized as either acquire or release operations. An acquire indicates that a processor needs to "see" the latest values for the next part of its program. A release indicates to the system that some values have been updated and can now be "seen" by other processors. For example, a lock/unlock pair corresponds to an acquire/release operation; a barrier is a release followed by an acquire. The processor treats acquire and release operations differently. An acquire causes the processor to stall until the acquire has completed, since the processor must be certain that it is "seeing" the new values before it continues. However, unlike a weakly ordered system, it is not necessary that all of its previous accesses have been performed before the acquire is issued. A release cannot be performed until all outstanding accesses have been globally performed. However, the processor does not need to stall until the release is completed. Only the release operation itself needs to be delayed, so that it can be certain that other processors see the new values. The final difference between weak ordering and release consistency follows from this. There can be an outstanding release operation when an acquire is issued (assuming no data dependency). Gharachorloo et al. also showed that under program restrictions similar to those mentioned above for weak ordering, a release consistent system behaves in a sequentially consistent manner [13].

3 Methodology

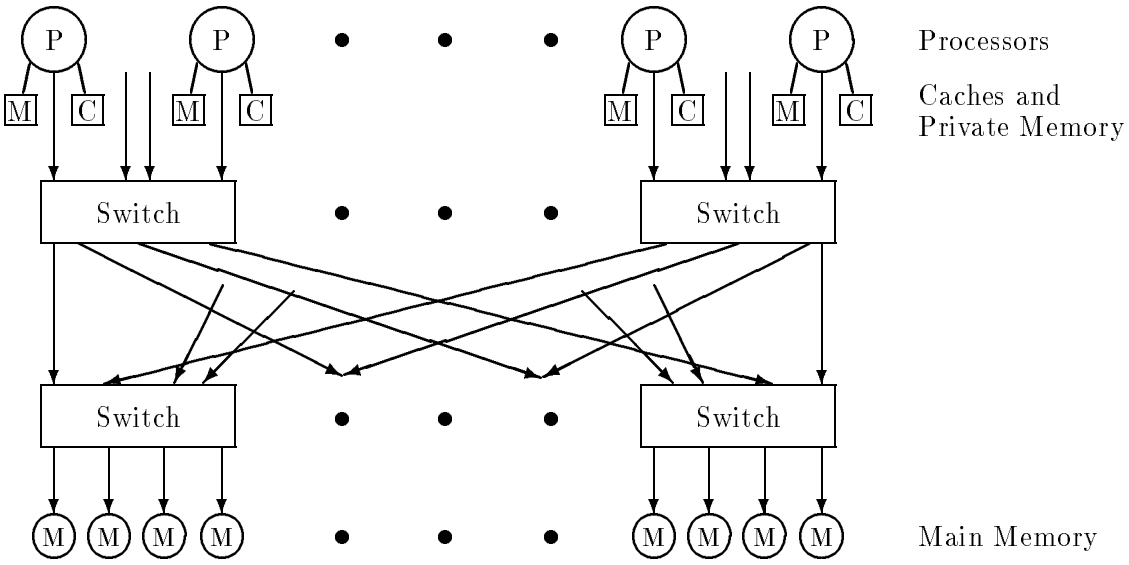

Figure 1: Model Architecture

3.1 Base Architecture

The architecture that we simulated is a typical "dance-hall" architecture (see Figure 1). 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 [6].

We used the Cerberus instruction-level simulator [5] for our simulation studies. The processor it simulates is a RISC processor similar to the Ridge 32 [24]. The caches are two-way set associative and use a write-back write-allocate policy; we experimented with a range of cache and line sizes. The caches are only for shared data. We assume that there are no instruction cache misses and that the private data can be accessed from local memory. We configured the simulator 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.

3.2 Model Implementations

We simulated five system types. Two are sequentially consistent, two are weakly ordered, and one is release consistent. A summary is given in Table 1.

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 oustanding accesses at an acquire

Table 1: Summary of Implementation Features

Sequentially Consistent 1 - SC1 SC1, a sequentially consistent implementation, is our baseline system. The programmer has complete freedom for the scheduling of inter-processor communication and synchronization. The hardware does not allow a subsequent memory access[1] if one access is already outstanding. However, we allowed one optimization, 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 [19] 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 (Our simulator forced us to use a (arguably long) delay of four cycles. We have since repeated the experiments with a smaller delay and found the absolute benefits of the models to be roughly the same. See section 5.3). 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 in [15], 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 we simulated obeys the definition for weak ordering given in section 2. With a WO1 implementation the programmer is limited to synchronization that is visible to the hardware. 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

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

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) [19] (we simulate five MSHR's). 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 5.1 we use blocking loads, and 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 in relaxed consistency systems 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 our 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 we had two FIFO buffers, one for loads and one for stores and a priority scheme always favoring the former. We will discuss in section 4.2.3 why our results are still valid despite our 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 our 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.

3.3 Benchmarks

The benchmark programs used to compare the various memory models are all written using PCP [4], 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.

Program	References (1,000's)		Hit Rate (%) by line and cache size					
			16K cache			64K cache		
	Reads	Writes	8 bytes	16 bytes	64 bytes	8 bytes	16 bytes	64 bytes
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

Table 2: Benchmark Statistics for SC1 for 16K and 64K caches
References are averages per processor and are in 1,000's.

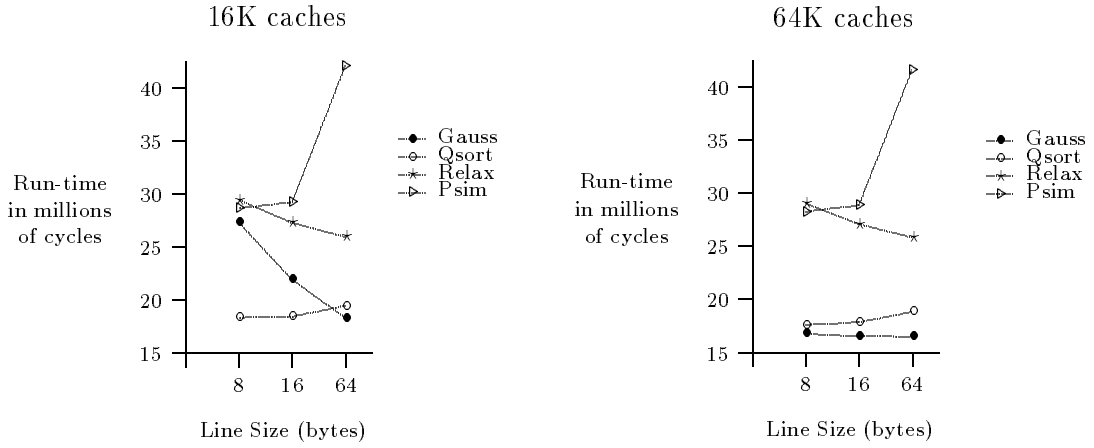

Figure 2: Performance by Line Size for SC1

Our four benchmark programs are described below. Table 2 shows statistics on the benchmarks for 16 processors for various cache and line sizes under SC1 and their run-times are given in Figure 2 (there are more detailed statistics in Appendix A). The exact number of memory references can change based upon the consistency model, and this can impact the hit rates.

Gauss *Gauss* [8] 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 2 and Figure 2. The larger line sizes result in large performance gains: the 64 byte line configuration shows a speed-up over 8 byte lines of almost 50% and over 16 byte lines of almost 25%. 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* [18] 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 [18]. In one case, when we changed the implementation from WO1 to WO2, we found that the number of synchronization operations increased by a third, thereby demonstrating the effect of dynamic scheduling. This natural variability prevents us 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, we have not yet had the time to repeat this experiment with different seeds and to compute appropriate confidence intervals. We will not discuss the generally small variations in performance we observed, since we cannot isolate the likely causes due to the dynamic nature of the program.

As shown in Table 2, 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, we see 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.

At first we found the low write hit ratios we observed curious. In our previous study [3] we found that *Qsort* had a write hit ratio of close to 100%. This is because before values are swapped, they must be compared. So, they are always read before being written, and hence, are already in the cache. However, in this study we found write hit ratios of around 70-80%. We finally realized that this was because we changed the way write hits are counted due to the different cache coherence protocol. In our earlier study we simulated a bus-based 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 still counts as a write-hit. However, in our 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. So, this is 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. 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 so that each processor references all the locations in a given cache line (for the shared-bus system for which the program was written [18], the overhead of doing this was found to remove any benefit it would have otherwise generated. However, this may be different in the case of a higher latency system such as the one we are simulating). Since there is a great deal of sharing during this phase, it would not be useful to do a read with ownership, which might not be needed if there was no swapping done by that processor. Later, when each processor is sorting its own partition, then it is worthwhile.

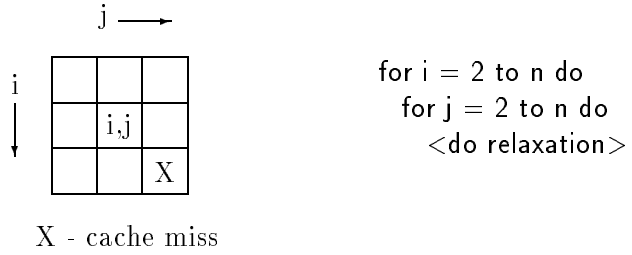

Figure 3: Relax access pattern

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 3. 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 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 [20]). 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 element 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. Our compiler does schedule all the loads before all the additions (this is discussed further in section 4.1.3).

This computation pattern 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 we predicted. As long as the cache is large enough to hold three 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 [20] 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).

There was a case overlooked in what was mentioned above about the cache hit pattern. 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. So, 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. We observe that 70% of its cache misses are invalidation misses, which is indicative of a high level of sharing. This increases the traffic on the Omega 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 2, can have a major performance impact. The data set of *Psim* is fairly small and fits into a 16K cache. So, 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 Simulation Results

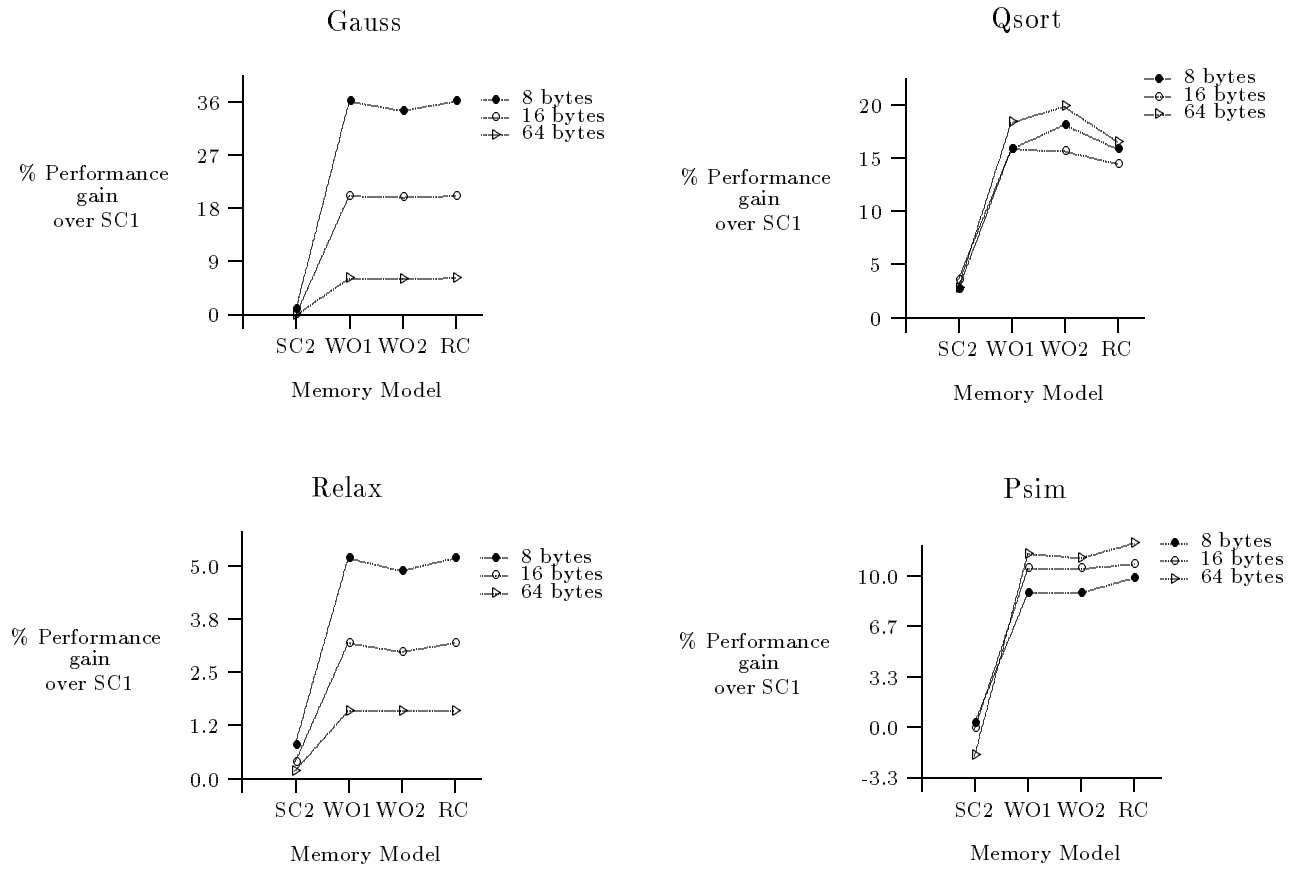

Figure 4: 16 processors, 16K caches

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

Our experiments were instruction-level simulations of 16 processor systems for all benchmarks and of 32 processor systems for *Gauss* only because of time limitations. We ran the simulations 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, the results for 16 processors and 64K caches are shown in Figure 5, and the results for *Gauss* with 32 processors are shown in Figure 6. In these figures, each graph has plots for each

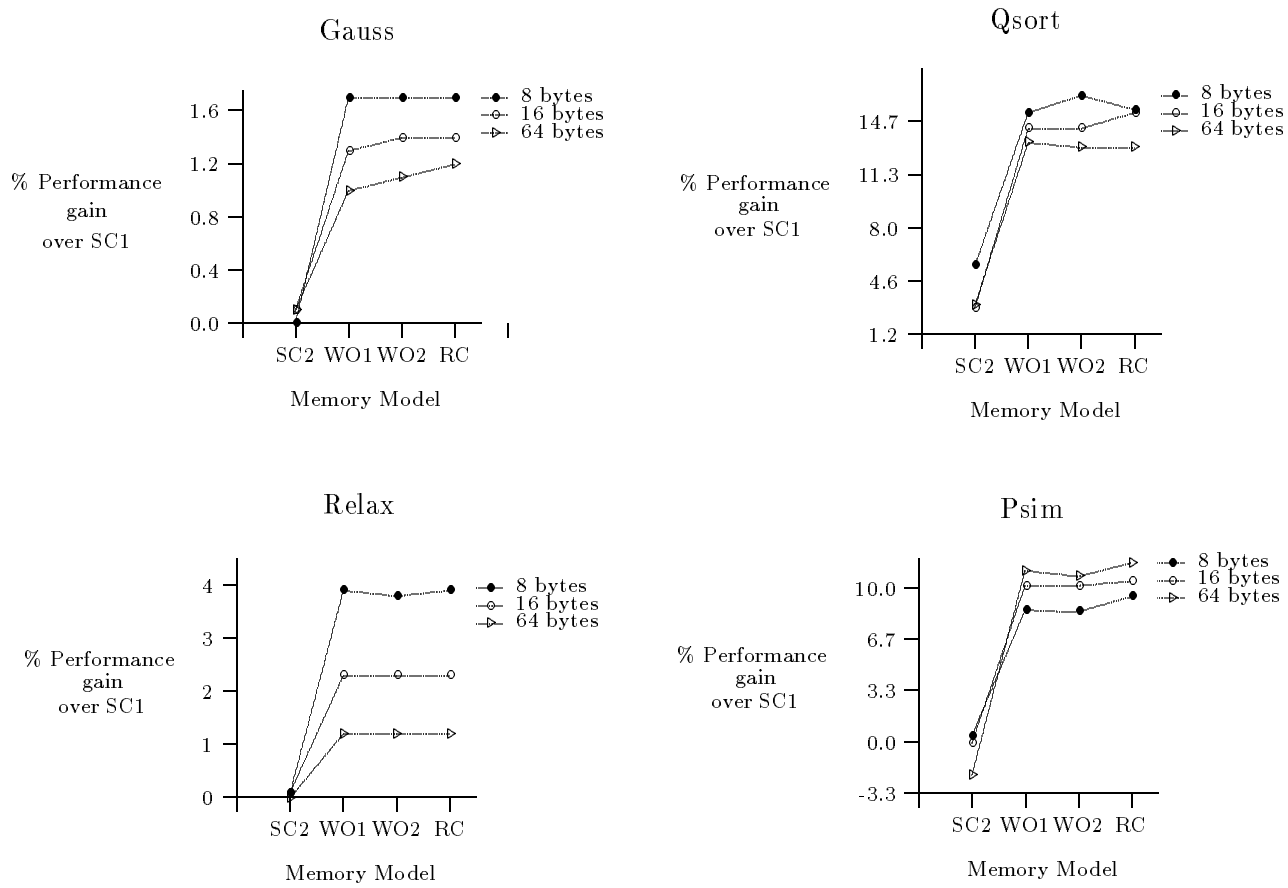

Figure 5: 16 processors, 64K caches

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

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 we discuss the results for each benchmark and then compare the results for the various memory models. This will allow us to draw some conclusions on the relative merits of each hardware addition.

4.1 Analysis of Benchmarks

4.1.1 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.

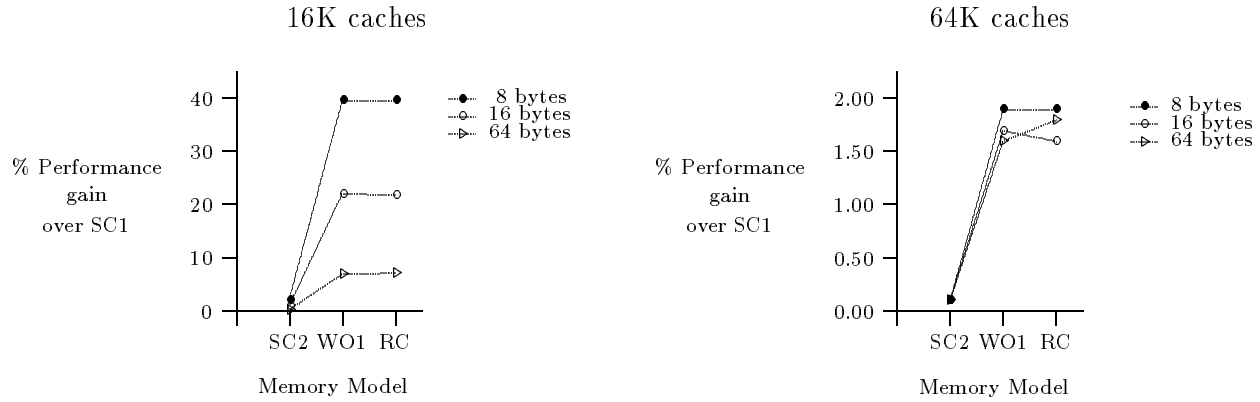

Figure 6: 32 processors, Gauss

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

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 data set still does not fit in the cache when 16K caches are used. 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 speed-up than SC1 did when using the same cache structure. We did not do studies of 32 processors for WO2. The reason is explained in section 4.2.3.

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. A 250×250 matrix of doubles occupies only half a megabyte of memory. Gaussian elimination on such a matrix takes about 10 CPU seconds on a modern workstation. The problems we expect to 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.

4.1.2 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 has the capacity to store the program's working set) accompanied by memory access patterns that allow overlapping. As mentioned in section 3.3, the variability of results for the various line sizes and models is caused by dynamic scheduling effects.

4.1.3 Relax

As can be seen in Figures 4 and 5, *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 3.3, 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 that causes a stall almost immediately when the destination register is used as described in section 3.3

This analysis of *Relax* made us 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 3.3) should be done first among the nine loads, thereby allowing the main memory access to be done while loading the other eight registers from cache. For larger lines with high hit rates (over 90%), we expect the effect of this optimization, for this size data set, to be minimal. Note that 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 the 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 our 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. We conducted some experiments where we manually scheduled the code taking this effect into account. The results are in Section 5.2.

4.1.4 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 *Relax*.

4.2 Relative Performance of the Consistency Models

4.2.1 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 (as in SC1) or store 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. We further discuss the general benefits of the relaxed models in section 6.2.

4.2.2 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 it 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.

4.2.3 WO2 - Bypassing

Overall, it appears that bypassing of stores by loads is not worthwhile with lockup-free write-back caches. Recall that our 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 and 5 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 our 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, we do not see any reason to consider implementing bypassing in a release consistent system or studying it with 32 processors.

4.2.4 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 we 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.

5 Architectural Variations and Results

5.1 Blocking Loads

With our implementations of the relaxed models, both read and write latencies are overlapped with computation. However, we do not know how much of the hidden latency is due to reads and how much is due to writes. Therefore, we modified two of our implementations to use blocking loads and compared our results with those we already had. With blocking loads, when there is 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 we 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 7 and 8.

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 section 5.2).

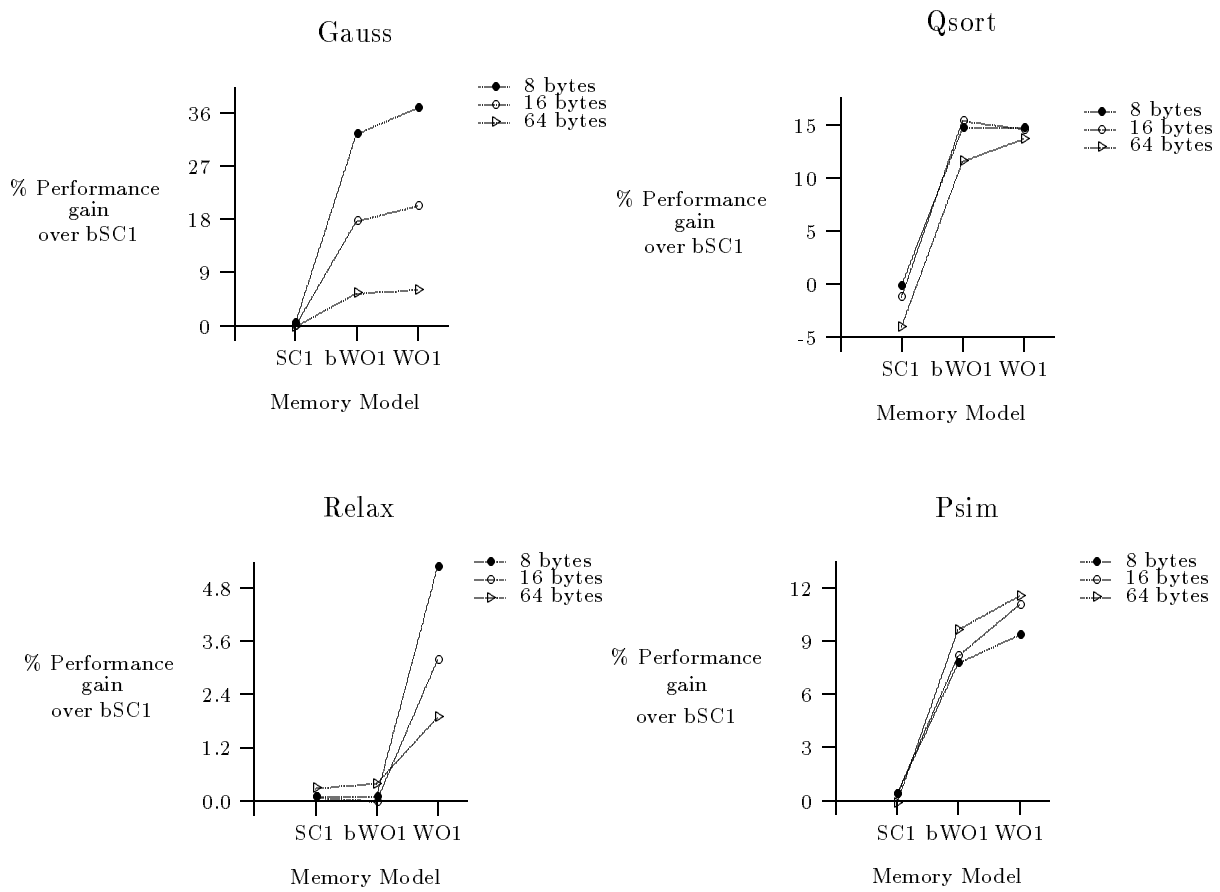

Figure 7: 16 processors, 16K caches, blocking loads

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

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. There is a significant difference however for WO1 for which the non-blocking loads are important. Because of the structure of the *Relax* program, it is clear 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 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 7 is 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 8). However, the differences are actually so small as to be unimportant.

5.2 Hand Scheduled Relax Code

In section 4.1.3 we described how simply moving all the loads to before all the floating point adds, as our 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

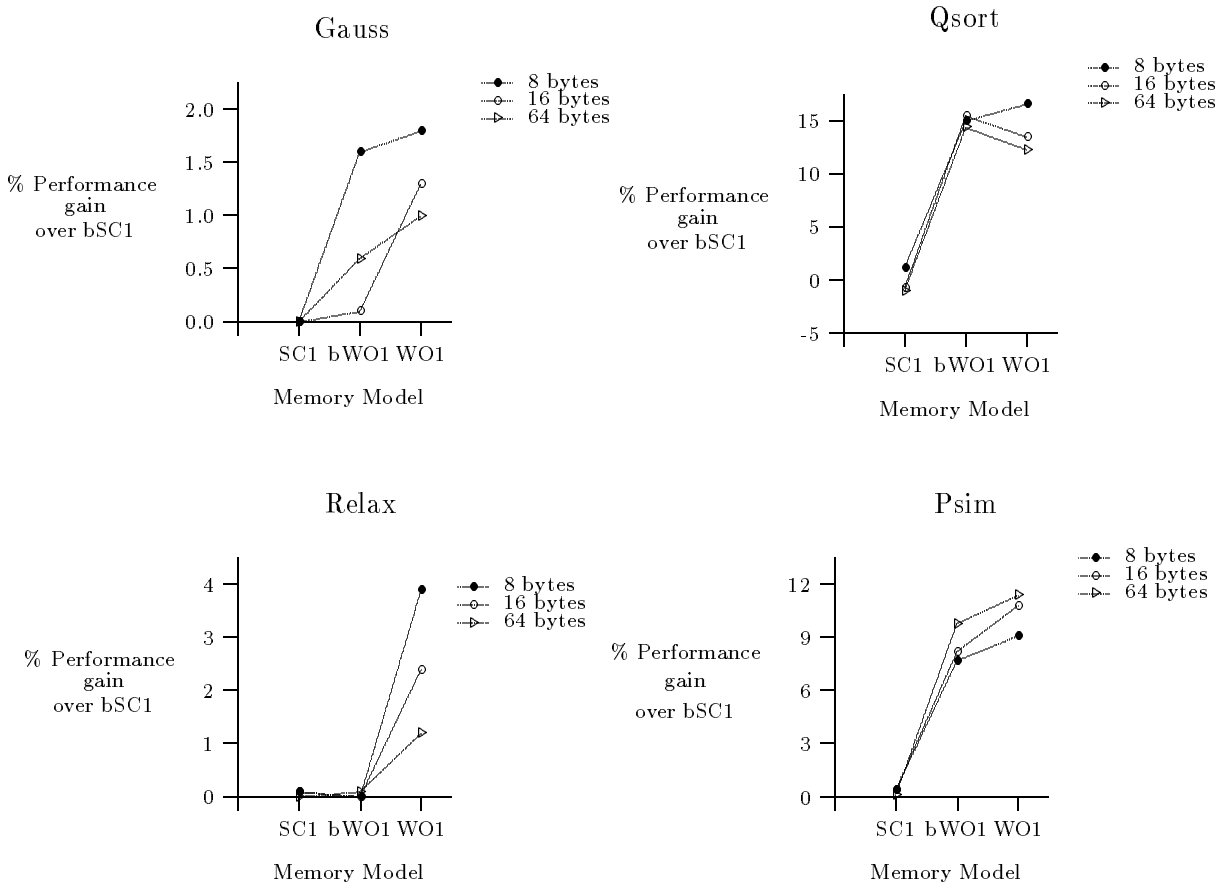

Figure 8: 16 processors, 64K caches, blocking loads

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

the loads, we manually scheduled the code in a more efficient order, with one schedule for the SC systems and one for the WO systems. As we described in section 4.1.3, we 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). We also manually scheduled the code in such a way to produce a deliberately bad schedule given the knowledge of which load would miss. These schedules were equally likely to have been produced by the compiler given its lack of knowledge of the architecture.

In Figure 9 we show the relative performance of the codes manually scheduled for good and bad performance compared to the optimizer's default schedule. We see that there is a noticeable difference (up to 8%) in performance. Normally the manual scheduling of the code will be impractical. However, compiler scheduling can perform some of these optimizations [7].

5.3 Two Cycle Load and Branch Delays

Due to limitations of our simulator, our 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). We duplicated our studies of SC1 and WO1 with load and branch delays of two cycles (this includes the load

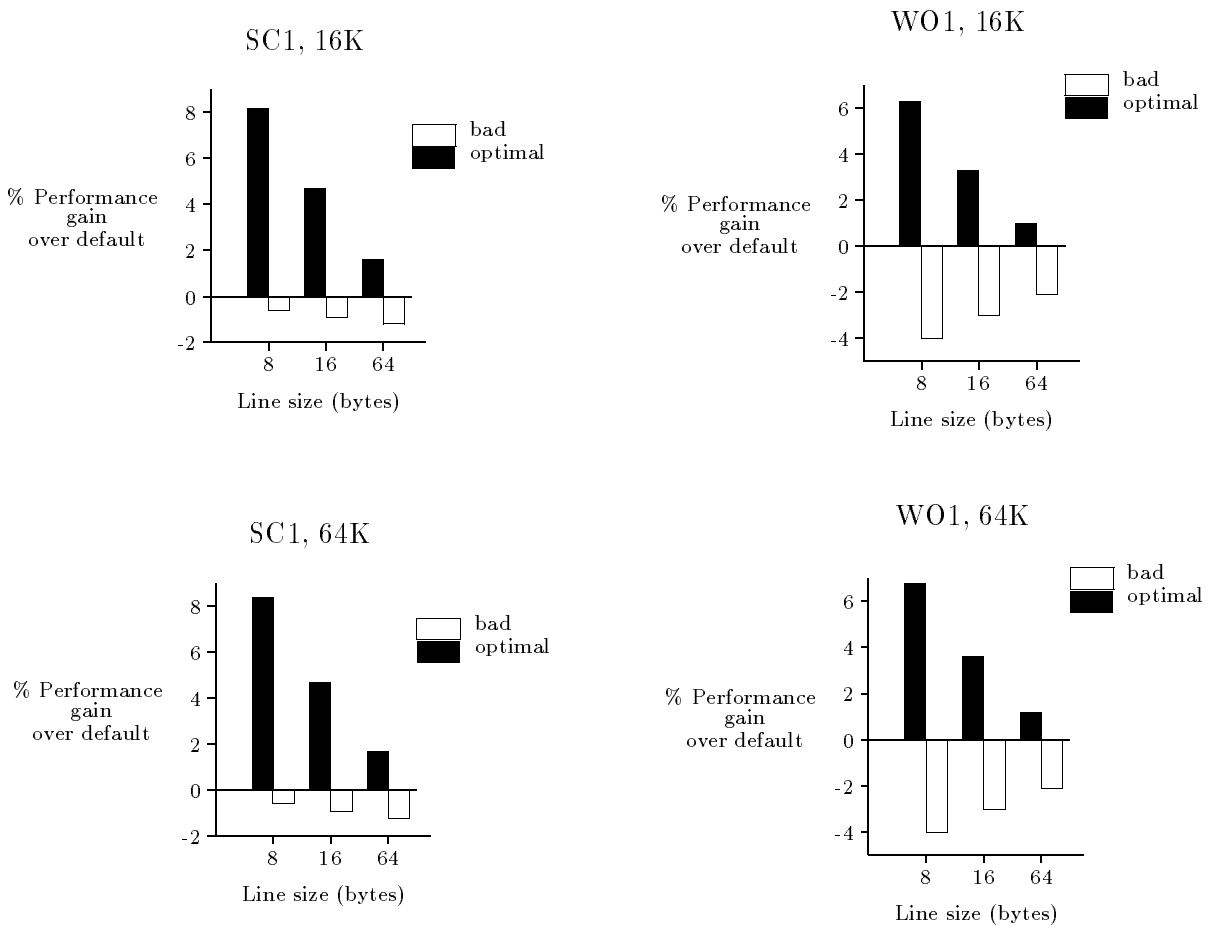

Figure 9: Effect of improved code scheduling

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

delay for loads of private data). The performance of the systems with this new parameter is shown in Tables 3 through 6. These results are consistent with those obtained with a four cycle delay and do not bring any further insight.

6 Related Work and Comparisons

6.1 Methodology

Simulation performance studies of relaxed consistency models have been done for two different architectures: a shared-bus multiprocessor [3] and a mesh connected multiprocessor based on DASH, an experimental system being built at Stanford [14, 23]. Our previous study [3] should not be compared to this one due to the modest number of processors (nine to twelve), the low memory latency, and the limitations of its trace-driven nature. However, the Stanford study, which used instruction-level simulation, is similar in scope to our current study but their architectural framework and memory model implementations are significantly different.

The Stanford study is based on a variant of the DASH [22] architecture with 16 processors connected

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

Table 3: Gauss, absolute and relative benefits

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

Table 4: Qsort, absolute and relative benefits

in a mesh-like fashion. Memory is globally distributed with a portion of global memory being 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 [12]. 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. [14] 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

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

Table 5: Relax, absolute and relative benefits

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

Table 6: 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.

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.

6.2 Results and Comparisons

Our 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 Stanford study were the same as ours, 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 Stanford study are much larger, thus giving a greater advantage to relaxed memory models. Second, 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 our 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, we 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 our 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. We believe that there are three main reasons for the difference in the level of improvement: (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), and (iii) even though the write hit rates are roughly the same, the number of write misses per

unit time of *Pthor* is over double that of *Psim*, thereby increasing the chances of a write being outstanding at a release. All three factors provide RC with a greater opportunity for a benefit over WO1.

7 Conclusions

We have studied the performance benefits of several implementations of models of relaxed memory consistency in shared-memory multiprocessors where each processor has a private cache and the interconnect is a multistage network. Using instruction-level simulations, we have shown that under different and more aggressive architectural choices the performance benefits that might arise from using relaxed models of consistency depend significantly on the cache structure and on the program characteristics and much less so on the sophistication of the model implemented.

Our results are qualitatively similar to those presented in a simulation of the DASH architecture. Programs with high cache hit rates, little sharing, and good locality will not benefit much, as expected, from relaxed consistency. The line and cache sizes, which are very important factors in determining the hit rates, will also greatly influence the utility of relaxed consistency. When the hit rates are lower, relaxed models will usually be advantageous. However, our study shows that the gains are proportional to the memory latency and the rate of misses per unit time as well as being dependent upon the ability of the compiler to schedule the code to take advantage of the relaxed models.

The main hardware features that allow relaxed models to be beneficial are non-blocking loads and lockup-free caches. Prefetching of one line when there is already an outstanding miss, and extra flexibility to prevent the processor from stalling at some synchronization points had only minimal secondary effects. As memory latency increases, it will clearly be worthwhile to develop multiprocessors with lock-up free caches and hardware visible synchronization points so that some form of relaxed consistency can be implemented. Relaxed consistency should be implemented in conjunction with other memory latency reducing techniques such as more sophisticated prefetching, speculative execution, fast context-switching or multithreaded architectures [17].

The effect of relaxed consistency on the programmer and the compiler needs to be investigated. We have shown, through an example, how the optimal code reordering depends upon the model of consistency implemented. Although we know what the constraints of relaxed consistency models are to the writer of correct parallel programs, the impact of these relaxed models on programming for efficiency is an open area of study.

We could not close this paper without a word of caution. Both our study and the Stanford study used time-consuming instruction-level simulations which limited the size of the benchmarks. It is unclear what the performance of a full size data set would be. It is also unclear what other methodology should be used, if any, to avoid the overbearing consumption of CPU resources. For example, limiting the cache size proportionally to the data set size might be an approach but we have shown that not all benchmarks (e.g., *Qsort*) would have been sensitive to that effect. This methodological question is one that certainly deserves further study.

Acknowledgements We would like to thank the MPCI Project of Lawrence Livermore National Labs for hosting the first author and providing the Cerberus Simulator and some of the benchmarks. Simon Kahan wrote *Qsort* and was of great help in explaining its intricacies. We thank Craig Anderson and Tien-Fu Chen for various forms of assistance, the UW architecture lunch, and Cathy McCann in particular, as well as Sarita Adve and Kourosh Gharachorloo, for numerous and excellent comments on earlier versions of this paper. Diane Gorenberg provided assistance with the writing style and overall readability.

References

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and preloading caches, 1992. submitted to Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.

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

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

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

[11] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, 1989.

[12] Kourosh Gharachorloo. personal communication.

[13] 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), June 1992.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

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

A Read/Write Statistics

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 7: Benchmark statistics for reads for SC1 for 16K and 64K caches
Reads 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 8: Benchmark statistics for writes for SC1 for 16K and 64K caches
Writes are averages per processor and are in 1,000's.

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

Table 9: 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.