Design and Evaluation of a Subblock Cache Coherence Protocol for Bus-Based Multiprocessors

Craig Anderson and Jean-Loup Baer

May 11, 1994

Email addresses: craig@cs.washington.edu, baer@cs.washington.edu
Telephone: (206) 543-7798
FAX number: (206) 543-2969

Abstract

Parallel applications exhibit a wide variety of memory reference patterns. Designing a memory architecture that serves all applications well is not easy. However, because tolerating or reducing memory latency is a priority in effective parallel processing, it is important to explore new techniques to reduce memory traffic.

In this paper, we describe a snoopy cache coherence protocol that uses a large sized transfer block (to take advantage of spatial locality) while using a small coherence block in order to avoid false sharing. To further illustrate the protocol, we present an example of its workings. We then present the results of simulating our protocol on 5 applications that exhibit a variety of reference patterns. We find that our protocol effectively takes advantage of spatial locality while avoiding the increase in false sharing that often occurs when using large line sizes.

Keywords: cache coherence protocol, multiprocessor

This work was supported by the National Science Foundation under Grants CCR-91 01541 and CCR-87 02195

1 Introduction

There is now a large and still expanding body of literature on the behavior of parallel programs. It has become quite clear that parallel applications exhibit a wide variety of memory reference patterns. Because of these differing reference patterns, it is difficult to design memory architectures that will serve all applications well. However, since tolerating or reducing memory latency is one of the main challenges to effective parallel processing, new techniques are continually under investigation to decrease memory traffic.

In this paper we investigate one such technique in the context of a cache coherent shared-memory multiprocessor architecture. Our main focus is on the relationship between the coherence protocol and the cache block size. Coherence protocols are either invalidation based – the write invalidation (WI) protocols – or update based – the write update (WU) protocols. It is well known that depending on the application, either WI or WU performs best[EK88]. Similarly, some applications run better when small cache block sizes are used because of the presence of migratory data or because false sharing is avoided, while others execute more quickly with larger block sizes because they exhibit good spatial locality.

Most often, an architecture is implemented with a given protocol and a block size used for both transfer (memory to cache or cache to cache) and coherence. Protocols with varying block sizes[DL92] or with partial block invalidations[CD93] have been proposed in order to reduce the number of invalidations or the amount of bytes transferred between the various components of the memory hierarchy. The choices can be made either statically (compile-time) or dynamically (run-time). For example, some architectures[MIP91] allow the choice of a coherence protocol to be determined on a page per page basis. Competitive snooping[KMRS88] and hybrid protocols such as the one implemented on the DEC Alpha [TCS92] allow dynamic switching between WI and WU protocols. Compiler optimizations have been advocated for the choice of a protocol at compile time[VF92] on a block per block basis.

In this paper we present a dynamic technique, namely a snoopy cache coherence protocol, that uses different block sizes for transfer and coherence. The goal of the protocol is to obtain the advantages of using large block sizes for those programs that have good spatial locality while avoiding many unnecessary invalidations or updates that result from migratory data and false sharing. In Section 2 we present our model architecture, in particular a sector cache organization that allows subblocks. Section 3 introduces a cache coherence protocol for sector caches. Section 4 describes our evaluation methodology with Section 5 giving simulation results. Section 6 is a brief summary of related work. Suggestions for improvements and further study are given in the concluding section.

2 Architectural Model

Our base architecture is a shared-bus multiprocessor. The base architecture can be seen as a cluster in a hierarchical multiprocessor. Our investigation at this point explores intra-cluster effects but the techniques could be extended to inter-cluster optimizations. Each processor has a private cache and coherence is maintained via a snoopy protocol (cf. Section 3). The shared bus is a split-transaction bus.

We consider two types of caches: "usual" caches and "sector" caches. Usual caches have a capacity C, a line size L, and an associativity k. Corresponding sector caches have the same

(C, L, k) characteristics but in addition the lines are divided into subblocks of size b. The units of coherence and transfer are the same for the usual caches, namely L, while these units can be L or b depending on specific protocols in the case of sector caches. In the sector caches, both lines and subblocks have states. A sector cache (C, L, k, b) will therefore require $2L/b$ extra state bits/line, assuming that subblocks have a maximum of 4 states. Thus an $(C, L/2, k)$ usual cache and a (C, L, k, b) sector cache require approximately the same number of tag and state bits for L between 32 and 128 bytes and $b = 8$ bytes. Note also that a usual cache with $L = b$ is more memory expensive than a sector cache with L and b as above.

Systems with sector caches also require more bus lines to transmit bitmasks corresponding to the status of the subblocks in a particular line. However, the number of additional lines will be comparatively small, since the number of extra lines required is equal to L/b, the number of subblocks in a line.

3 Coherence Protocol

The motivation behind sector caches is to design a coherence protocol that takes advantage of applications' spatial locality by fetching large sized lines while avoiding invalidations and migrations of falsely shared data by having a smaller coherence unit, namely a subblock. The protocol is snoopy-based[AB86]. It incorporates features from the Illinois protocol [PP84] (the Illinois protocol will be the basis for comparisons with usual caches) and from protocols or write policies with subblock (in)validations [CD93, Jou93].

The basic philosophy behind the protocol is as follows. As much as possible, we favor cache to cache transfers. On read misses, we transfer as many valid subblocks in the line as possible. On writes to clean subblocks and write misses, we invalidate only the subblock to be written. We also implemented read broadcasting (also called snarfing): when a cache or main memory responds to a read request, all other caches use the data on the bus to update an invalid subblock (or subblocks) if the address of the invalid subblock matches that of the data on the bus[SR84, EK89]. Unlike Eggers' study, we assumed that read-broadcasting would not lock out the processor from accessing the cache.

In contrast to the Illinois protocol, dirty subblocks that are transferred on read misses are not copied to memory. An early version of our protocol implemented this operation but we discovered that the memory queues were becoming unduly large. Instead, we use subblock states to encode a form of ownership. Our final protocol has four states for the cache line and four states for each subblock. In the following descriptions, line states will be in **bold** type, while subblock states will be in *italic* type. The line states are as follows:

INVALID All subblocks are *Invalid*.

VALID EXCLUSIVE All valid subblocks in this line are not present in any other cache. All subblocks that are *Clean Shared* may be written without a bus transaction. Note that any (or possibly all) subblocks in the line may be *Invalid*. There also may be *Dirty* blocks which must be written back upon replacement.

In the final version of the paper we plan to quantify the benefits of using read broadcasting on the subblock protocol.

CLEAN SHARED The line contains subblocks that are either *Clean Shared* or *Invalid*. The line can be replaced without a bus transaction.

DIRTY SHARED The subblocks in this line may be in any state. There may be *Dirty* blocks which must be written back on replacement.

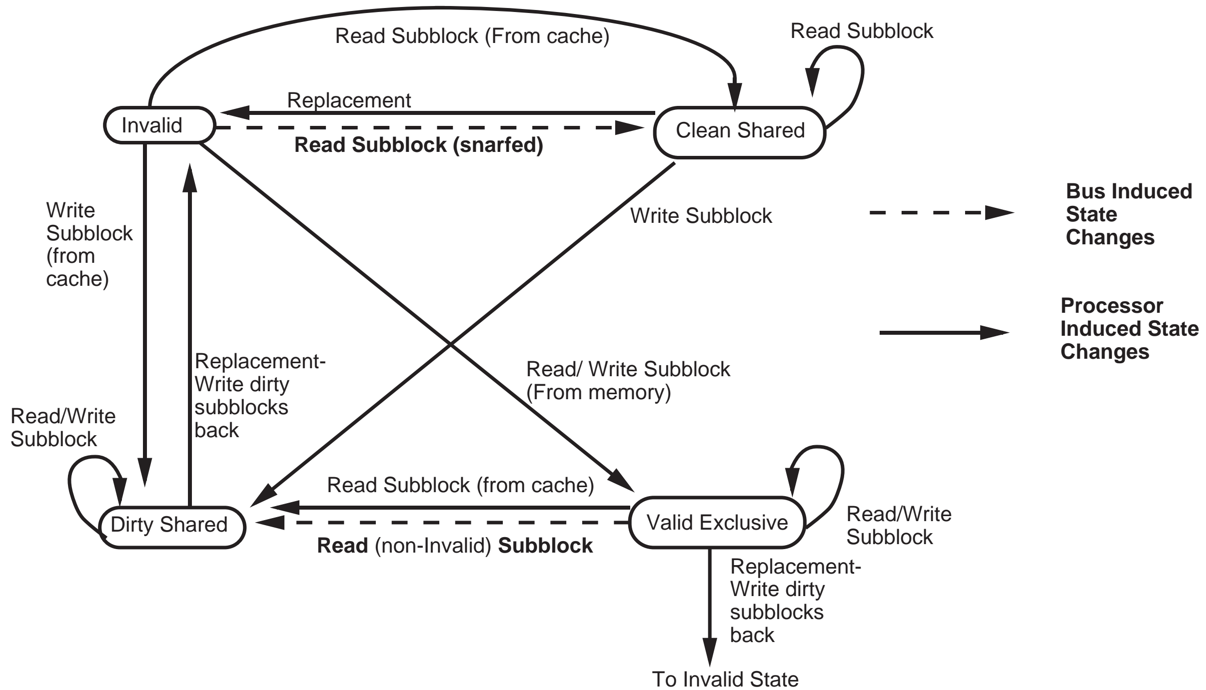


Figure 1: Line Protocol

The subblocks states are as follows:

Invalid The subblock is not valid.

Clean Shared A read access to the subblocks will succeed. Unless the line the subblock is a part of is in the **VALID EXCLUSIVE** state, a write to the subblock will force an invalidation transaction on the bus.

Dirty Shared The subblock is treated like a *Clean Shared* subblock, except that it must be written back on replacement. At most one cache will have a given subblock in either the *Dirty Shared* or *Dirty* state.

Dirty The subblock is exclusive to this cache. It must be written back on replacement. Read and write accesses to this subblock hit with no bus transactions.

Table 1 shows the states a subblock may be in given a particular line status.

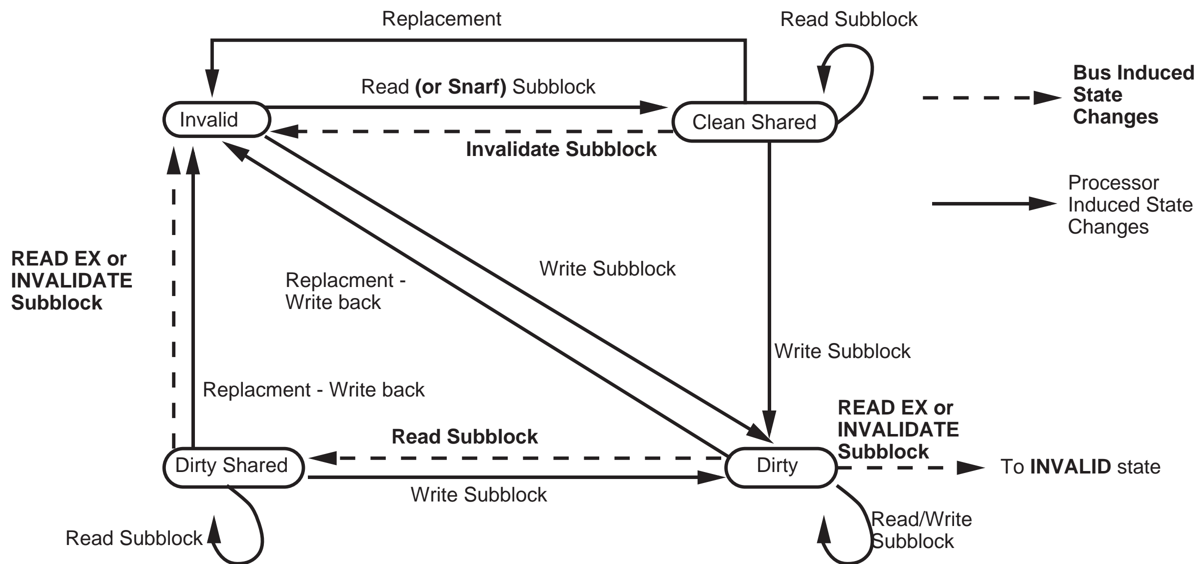


Figure 2: Subblock Protocol

Line Status	Subblock Status			
	Invalid	*Clean Shared*	*Dirty Shared*	*Dirty*
INVALID	OK	Error	Error	Error
VALID EXCLUSIVE	OK	OK	Error	OK
CLEAN SHARED	OK	OK	Error	Error
DIRTY SHARED	OK	OK	OK	OK

Table 1: Allowable states for subblock

Figure 1 gives the state transitions for cache lines. Note that there is only one transition (from **VALID EXCLUSIVE** to **DIRTY SHARED**) that is induced by requests from other processors that the cache observes. All other bus induced state changes change subblock states only. Figure 2 shows the state changes at the subblock level. For a complete description of the protocol, see Appendix A.

To illustrate the protocol, we give an example of its workings on a particular cache line. A complete description can be found in an upcoming technical report. In Figure 3 we show the status of a cache line, say L, which contains 4 subblocks. The subblocks will be referred to as a through $a+3$. Initially, the line is invalid in all three caches of the processors P1, P2 and P3.

At step one, P1 reads subblock $a+2$. Since no other cache has the block, the request goes to memory. Memory returns all subblocks in the line. The state of the line is **VALID EXCLUSIVE**, which signifies that all valid blocks in L exist only in this cache. The four subblocks are loaded in the *Clean Shared* state.

At step two, P2 writes subblock $a+1$. An invalidation message is sent over the bus, which invalidates only subblock $a+1$ in P1's cache. Note that P1 remains **VALID EXCLUSIVE** because all of its remaining subblocks are still unique to it.

On the third step, P3 reads subblock a. P1 supplies subblock a as well as all other valid, non-dirty subblocks it possesses. The state of the line in P1's cache is changed to **DIRTY SHARED** because other copies of L's subblocks exist in the system. Note that P2's cache snarfs the 3 valid blocks.

The next operation is a write to subblock $a+3$ by P1. This operation invalidates the subblock in P2's and P3's caches.

Following this operation, let us assume that a request from P1 induces a replacement of L in P1's cache. A bitmask is transmitted with the write back request so that only subblock $a+3$ is updated in memory.

In the final operation, P3 reads subblock $a+3$. When the request is broadcast over the bus, each cache responds with a bitmask indicating which subblocks of L (if any) it has stored in a valid state. These bit-masks are OR'ed together to give a complete view of which subblocks of this particular line are cached – and valid – in the system. Because no other cache has the subblock $a+3$, the request must go to memory. The memory unit returns the entire line, but only those subblocks which are not cached elsewhere in the system are marked valid in the requester's cache (in addition to those subblocks, like a and $a+2$ that were already valid). This is to prevent stale data from memory from being marked valid in a cache. In our example, the $a+1$ block is not marked valid in P3's cache, because it is cached and dirty in P2.

4 Methodology

4.1 Benchmarks

We modified the Cerberus multiprocessor instruction-level simulator [Bro89, AB93] to evaluate the sector cache protocol. The basis for comparison with usual caches is the Illinois protocol. The use of an instruction-level simulator not only allowed us to evaluate the merits of the new protocol but also to check its correctness by comparing the results of application runs on the simulator against runs on the native workstation. Instruction references and references to non-shared locations are assumed to always hit in a private cache. The width of the bus in all simulations was 64 bits. The

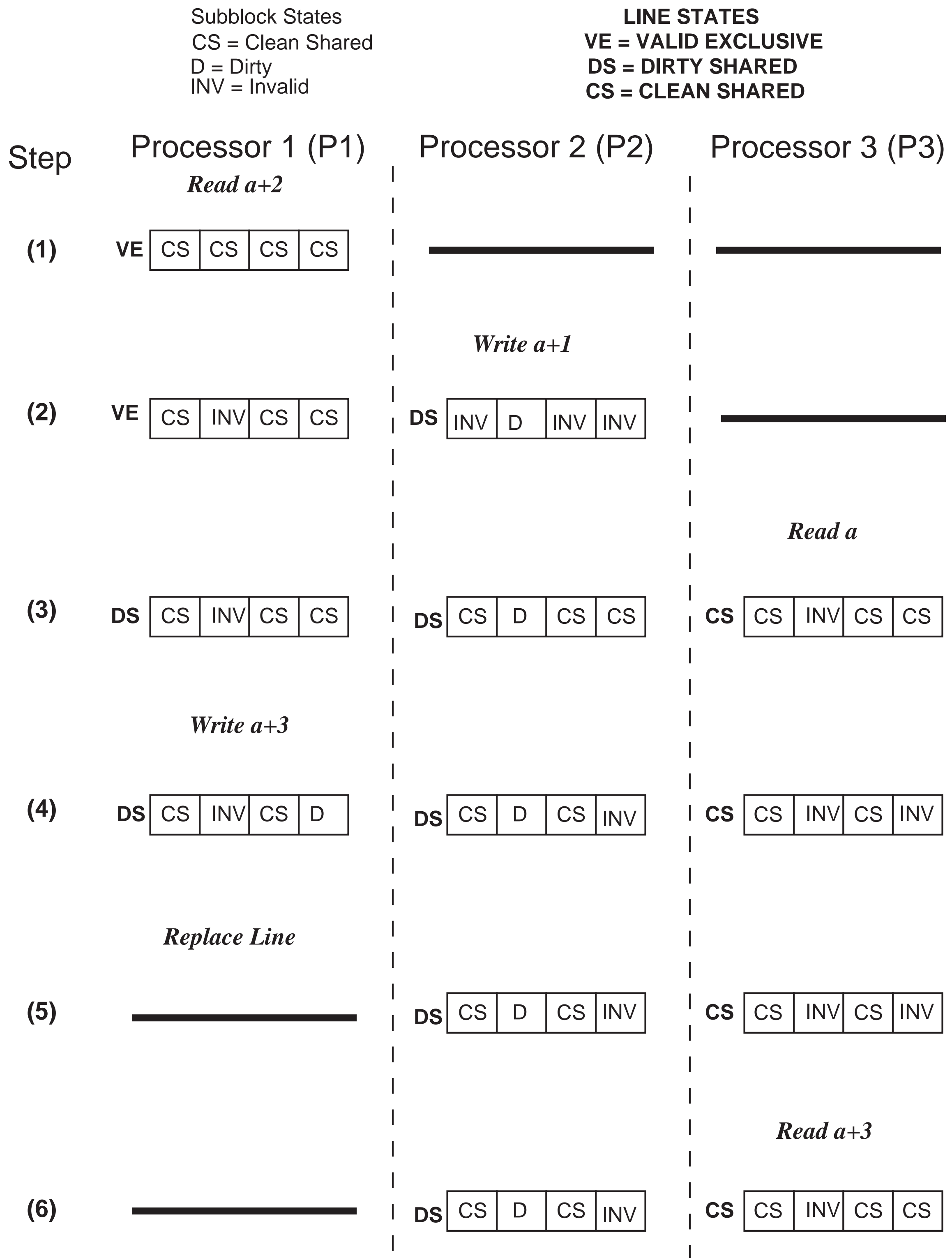


Figure 3: Protocol example

latency of main memory was assumed to be 10 cycles.

We picked five applications to evaluate our protocol. The applications' reference behaviors fall into three categories:

- Applications that have very little sharing and exhibit high cache hit rates (Gauss, Relax, Cholesky)
- Applications that exhibit true sharing (MP3D)
- Applications that exhibit both true and false sharing (Topopt)

Gauss is a gaussian elimination program to solve a system of linear equations. The first phase, which is the reduction of a square matrix to upper triangular form, lends itself well to parallelism [Dar88]. The second phase is back substitution that takes much less time than the triangularization. We ran our simulation on a 250x250 matrix, comprising approximately 161 million instructions on a single processor.

Relax is a successive over-relaxation program on a 256x256 array of 8 byte floating point numbers. For our study, we executed 10 iterations of the algorithm, gathering statistics for only the final 5 iterations to eliminate cold-start effects. This represents the simulation of 62 million instructions.

MP3D is a 3-dimensional particle simulator from the SPLASH suite [Sin92]. The program evaluates the positions and velocities of molecules over a sequence of time steps. Although it appears that the program is well suited for parallelization, speed-ups are hampered by changes in localities when molecules move from one "space" cached in one processor to another. As a consequence, most of the cache misses are coherence misses. We ran MP3D on 50000 molecules for 50 steps, approximately 21 million instructions.

Cholesky is a program from the SPLASH benchmark suite for sparse matrix factorization[Sin92]. Columns of the matrix are grouped into supernodes to obtain adequate granularity. Each free processor takes a supernode off the task list and modifies the appropriate parts of the main matrix. Processors interfere with one another when attempting to take work off the queue, and when making modifications to the same destination column. Our simulation used the BCSSTK14 input file, which ran for 65 million instructions.

Topopt performs topological optimization on VLSI circuits using a parallel simulated annealing algorithm[DN87]. This application exhibits a fair amount of both true and false sharing[EJ91]. We used the cpla1.lomim input file, approximately 1.7 billion instructions.

4.2 Simulated architectures

Our experiments simulated a variety of usual and sector caches. We varied C, the cache capacity, and L, the line size. In all experiments the subblock size b was 8 bytes and caches were two-way set-associative ($k = 2$).

Caches were either small ($C = 32K$) or large ($C = 128K$). The line size was either 8, 32 or 64 bytes. For the usual caches we used the Illinois protocol. For the sector caches we simulated the protocol described previously (of course there was no simulation of sector caches with a line size of 8 bytes).

Finally, we simulated the applications on systems with 1, 4, 16, and 32 processors.

4.3 Metrics

The principal metric we used to evaluate the protocols was execution time in simulated clock ticks. For all programs, all measurements were done only on the parallel section of the application; for the programs studied, the execution time of the serial section of the code is negligible compared to the time spent in the parallel section.

We also gathered other performance metrics that allowed us to interpret the simulation results. Among these, we paid particular attention to the read/write miss rates since we cannot expect to see improvements in the coherence protocol to be of much value if the miss rates are very low. We also report on the number of cycles the bus was busy. This metric is especially important because bus saturation limits the number of processors that can effectively be used to speed up a program. For some applications, increasing the cache line size may reduce the cache miss rate, yet the program may run slower because the additional time to transmit larger lines increases both bus utilization and the overall time it takes to satisfy a cache miss.

5 Results

In this section we present the results of our simulations. In Figures 4 through 11, we show each application's execution time when run using the large line sizes (32 and 64 bytes) normalized to the running time of the program using the Illinois protocol using an 8 byte line size. We give more detail about two applications: Gauss, because it is representative of the applications with good locality; and MP3D, an example of a program with a great deal of sharing. For these two programs, we show the overall speedup versus the single processor, 8 byte line size Illinois protocol. We also show the number of bus busy cycles during the program's run.

5.1 Gauss

Gauss has a high degree of spatial and temporal locality and little sharing between processors. Because of this, miss rates for both reads and writes were low. Write miss rates across all studied architectures and both protocols were 1% or lower. However, there were significant differences in the read miss rates between runs using the small line size Illinois protocol and those using the larger line sizes. This is due to the fact that satisfying a cache miss when the line size is large also brings in additional matrix elements. First accesses to these elements hit in the cache, whereas such accesses are misses when small line sizes are used. Read miss rates using small line sizes ranged from 10% to 14%, while miss rates for larger line sizes varied between 2% and 4%. These higher miss rates did not affect execution times much when few processors were used. When bus and memory contention become high, though, using larger line sizes produced much smaller execution times, as can be seen in Figures 4 through 7 and Figure 12. On 16 and 32 processors, Gauss runs 2.5 to 3.5 times faster using 32 or 64 byte lines. While the Illinois protocol with 8 byte lines reaches saturation at 4 processors (cf. Figure 12), speedups still increase after 16 processors with larger line sizes. Using larger line sizes also substantially reduces the amount of time the bus is busy, as shown in Figure 13. Because the miss rates were so low for the large line size Illinois protocol, there was little room for improvement on the part of the subblock protocol. Nonetheless, in all cases

Bus cycles are reported for the entire program run; this will be changed in the final version of the paper.

Gauss executed as fast (and sometimes a bit faster) using the subblock protocol as it did using the large line size Illinois protocol.

5.2 Relax

Relax's memory reference patterns are similar to those found in Gauss. The relax application exhibited very low read miss rates; across all simulation runs, the rate was 3% or below. This is not surprising, since each element in the matrix is read 9 times each iteration of the main loop. However, write miss rates varied a great deal among the simulation runs. For a given number of processors, write miss rates (and execution time) for large caches (128K) varied little when the line size under the Illinois protocol was varied from 8 to 32 to 64 bytes, or when the subblock protocol was used (cf. Figures 6 and 7). However, when small caches were used (32K), write miss rates for the Illinois protocol were high (from 50% to 75%) when small line sizes were used. Miss rates were much lower (10% to 20%) with larger line sizes because there is a great deal of spatial locality for write accesses. Since the cost of write misses is low when there are few processors in the system (because of a lack of bus and memory contention), there was little difference in execution time in the 1 and 4 processor systems. However, when bus contention is high due to the large number of processors, using large line sizes substantially reduced execution times (cf. Figures 4 and 5 which show execution time improvements between 1.5 and 2.5). There was little difference between the performance of the subblock protocol and the Illinois protocol since there is relatively little sharing in the application.

5.3 Cholesky

Because there is not a great deal of data sharing in Cholesky, increasing the line size in the application decreased execution times. In all cases except one, the execution time of the subblock protocol was within 3% of the time of Illinois protocol on the same line size (the biggest difference was 5%). Because there are many locking operations done in Cholesky, the addition of the *Dirty Shared* subblock state to the subblock protocol reduced the latency for the memory unit, because otherwise the subblock would have to be written back to memory after every lock or unlock operation (and subsequent read by another processor) just as in the Illinois protocol. This allowed the performance of the subblock protocol to stay competitive with the large line size Illinois protocol, even though the subblock protocol had more misses (and bus busy cycles) than the large line size Illinois protocol. Using large line sizes gave an improvement of from 5% to 38% for small caches (cf. Figures 8 and 9) and from 3% to 26% for large caches (cf. Figures 10 and 11).

5.4 MP3D

MP3D has a great deal of true sharing in one of its principal data structures. In practice, this sharing strongly limits the amount of speedup that can be obtained. Because of this sharing behavior, we would expect that increasing the line size under the Illinois protocol would strongly degrade application performance when compared to using 8 byte line sizes. As seen in Figures 8 through 11, using large line sizes with 1 processor actually increased performance by up to 10% over the small line size cases. But as the number of processors increases, the advantage of using large line sizes turns into a disadvantage, so much so that in the worst case (32 processors, 128K caches, 64 byte line size) the large line size case takes almost twice as long to execute as the 8 byte

case. In contrast, the subblock protocol took only 4% longer to execute (cf. Figure 11). Though the large block Illinois protocol had miss rates, about 6% that were 10% to 20% lower than those of the subblock protocol, a cache miss on the Illinois protocol caused much more data to be transferred per miss than the subblock protocol. Indeed, the 8 byte Illinois protocol had the highest miss rates of all, about 8%, yet performed better when compared to using larger line sizes under the same protocol because each miss required fewer bus cycles to satisfy. The overall reduction in bus cycles can be seen in Figure 13 which shows that at equal line sizes the number of bus busy cycles under the subblock protocol is less than half that of the Illinois protocol. As expected, the number of bus busy cycles went up markedly as the number of processors increases. Because of these additional bus cycles, performance for the large line size Illinois protocol actually decreased as the number of processors is increased from 16 to 32, while the subblock protocol and the 8 byte Illinois protocol actually posted small increases in performance (see Figure 12). Thus using the subblock protocol with large line sizes produces a dramatic gain in performance over the standard Illinois protocol.

5.5 Topopt

As previously discussed, Topopt exhibits a great deal of true and false sharing. As seen in Figures 8 through 11, Topopt's performance under the Illinois protocol substantially decreased when the line size was increased from 8 bytes to 32 and 64 bytes. In the 16 processor case, performance of the 32 byte line size dropped 9% from the 8 byte Illinois protocol, while using 64 byte line sizes performance decreased by 27%. In contrast, the performance of the subblock protocol decreased very little (if at all) from the 8 byte Illinois protocol across the various configurations studied. The source of the difference between the subblock protocol and the Illinois protocol for large line sizes is the additional misses caused by sharing. These misses generate many more bus busy cycles for the Illinois protocol than the subblock protocol. The delay in satisfying these misses substantially increased program execution time.

5.6 Results summary

From the above results, we can see that using small coherence blocks with large transfer blocks gives good performance for both applications with very little sharing (e.g. Gauss et al.) as well as applications with a great deal of sharing (MP3D and Topopt). Using large line sizes, the subblock protocol (along with the Illinois protocol) sped up Gauss by 2.5 to 3 times over using small line sizes with the Illinois protocol. At the same time, the subblock protocol performed about as well as (and sometimes better than) the 8 byte Illinois protocol for those applications with a great deal of sharing, cases in which using a large line size with the Illinois protocol substantially reduces performance. The number of cache state and tag bits required by the subblock protocol (used with large line sizes) is actually less than that needed by the Illinois protocol for an 8 bit line size (for a given cache size), since the additional bits needed for the subblock states are more than matched by the savings in tag bits. Though a cache that implements the subblock protocol for a given line size does require a bit more memory than a cache that supports the Illinois protocol, the cost is minimal given the performance advantages of the subblock protocol.

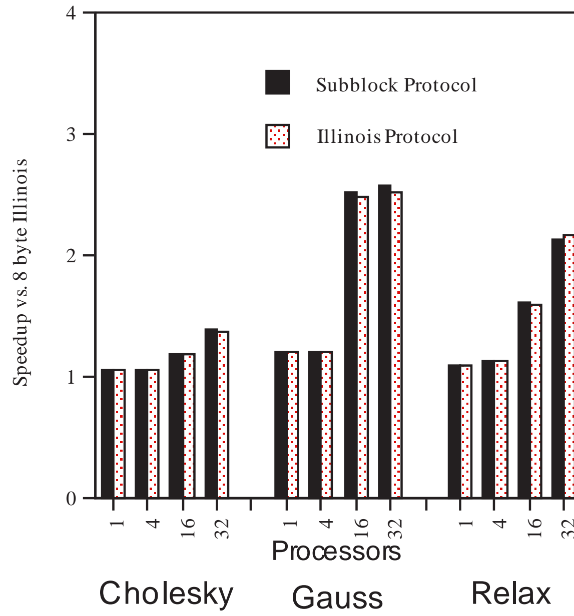


Figure 4: Speedups for 32K cache, 32 byte line size

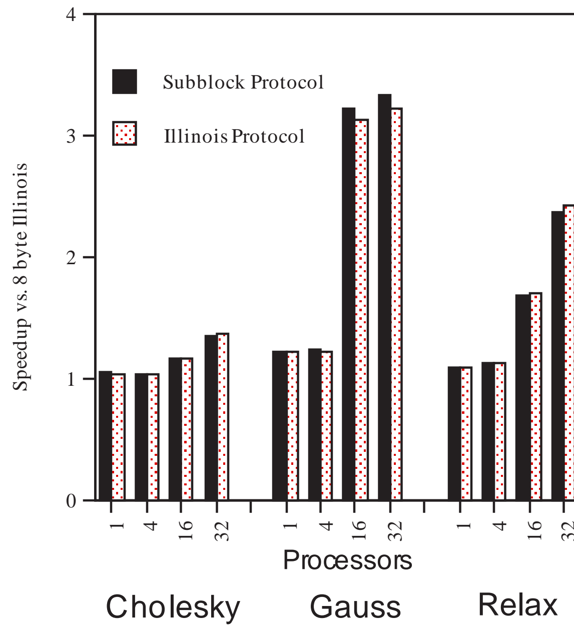


Figure 5: Speedups for 32K cache, 64 byte line size

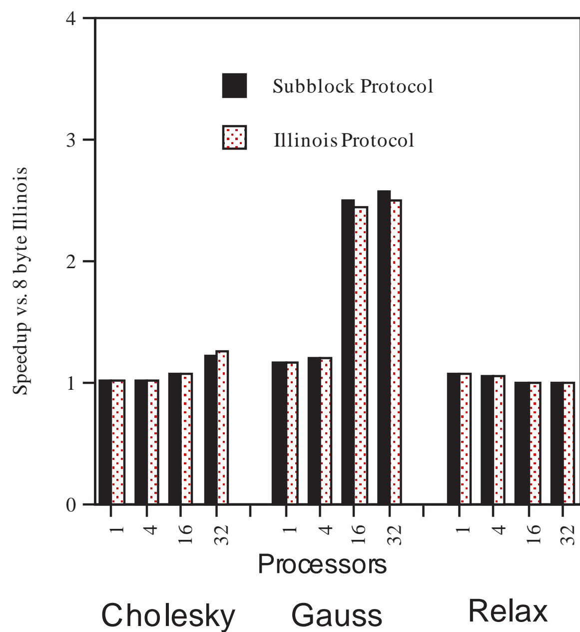


Figure 6: Speedups for 128K cache, 32 byte line size

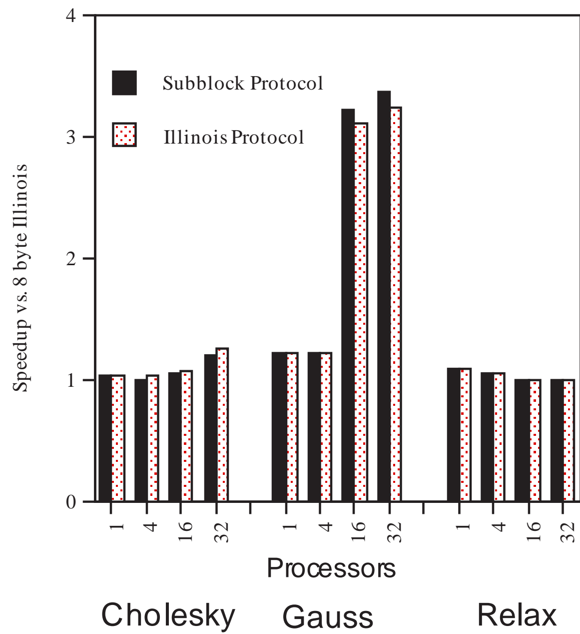


Figure 7: Speedups for 128K cache, 64 byte line size

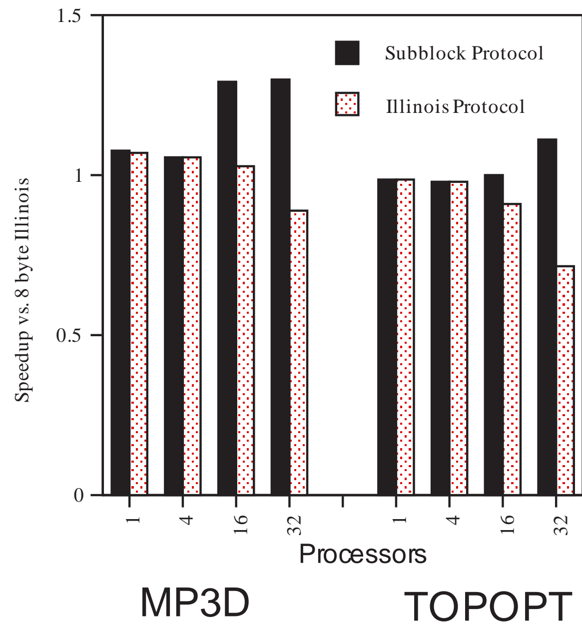


Figure 8: Speedups for 32K cache, 32 byte line size

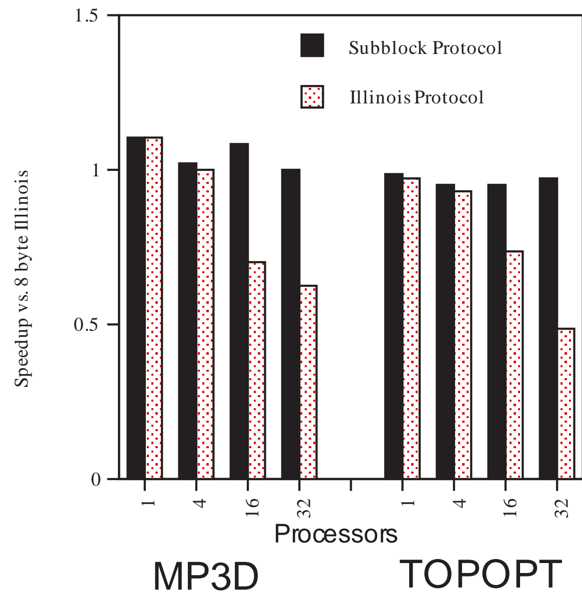


Figure 9: Speedups for 32K cache, 64 byte line size

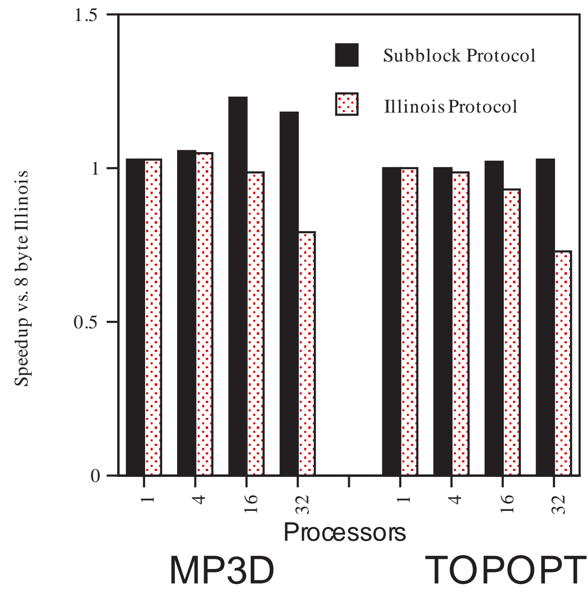


Figure 10: Speedups for 128K cache, 32 byte line size

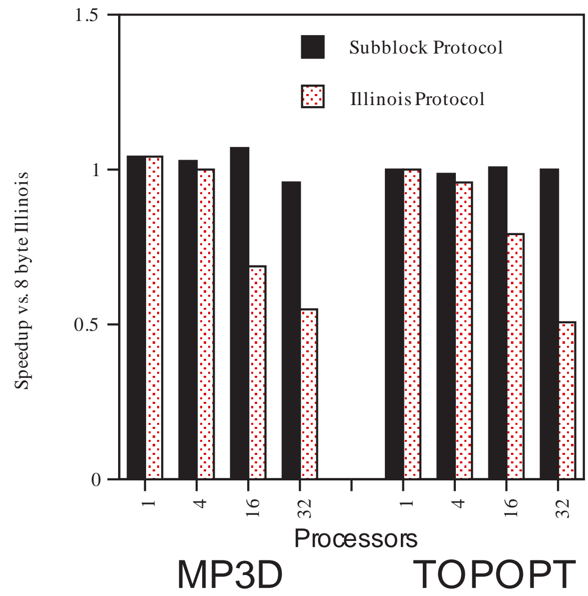
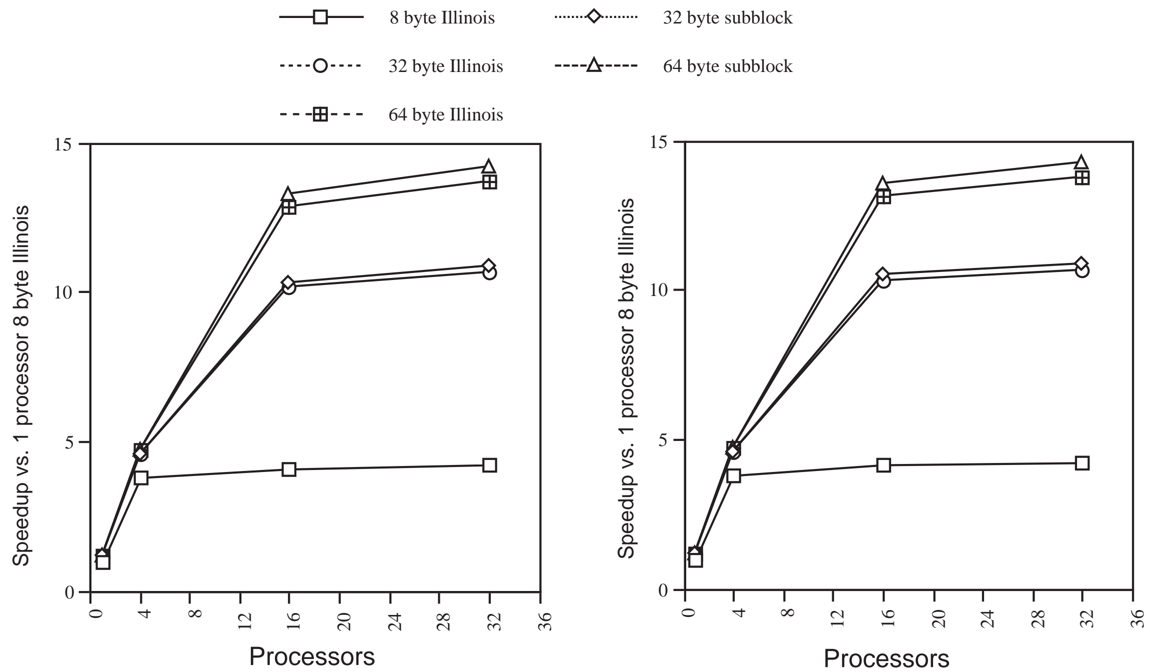
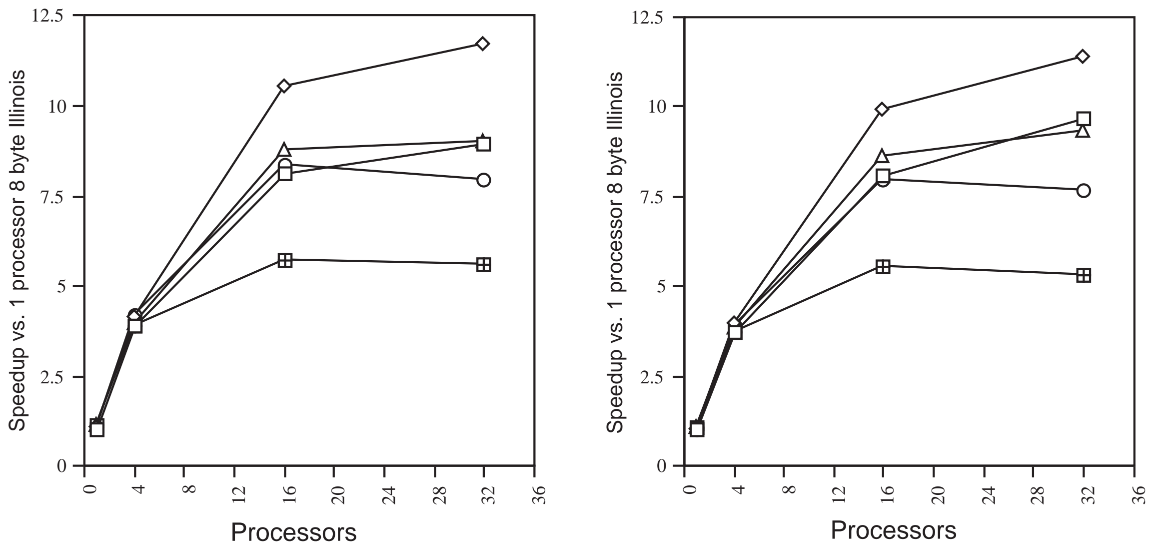


Figure 11: Speedups for 128K cache, 64 byte line size

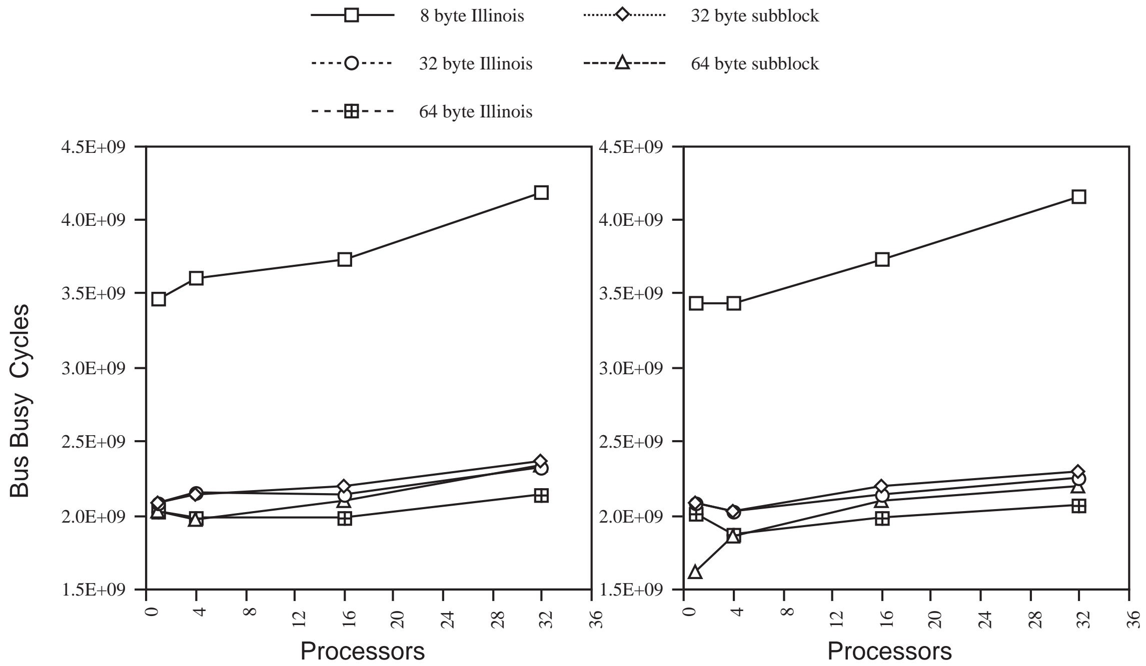


Gauss Speedups: 32K (L) and 128K (R) Caches

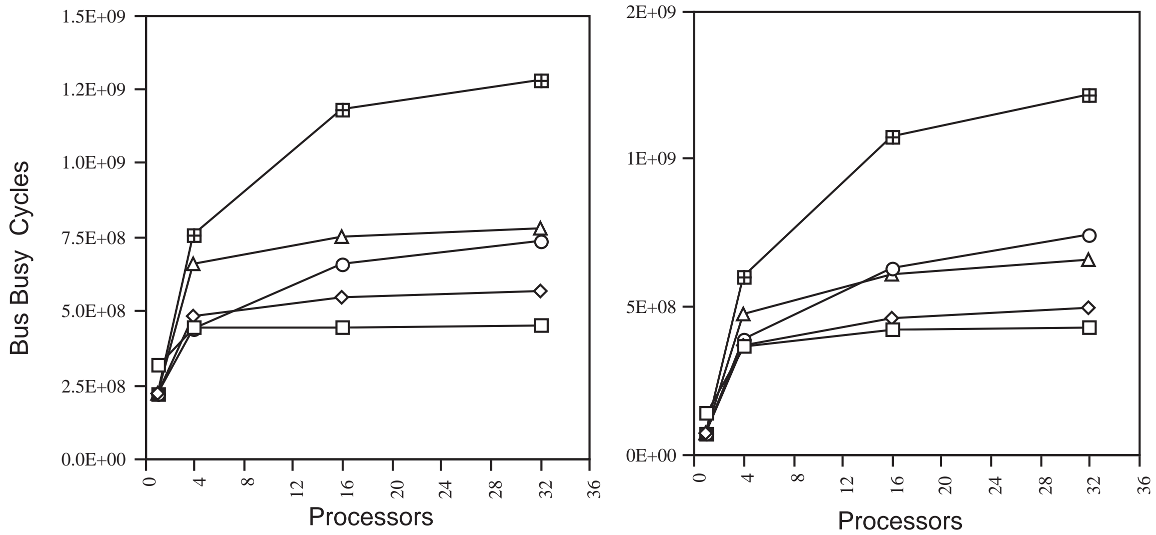


MP3D Speedups: 32K (L) and 128K (R) Caches

Figure 12: Speedups for Gauss and MP3D Relative To Single Processor



Gauss Bus Busy Cycles: 32K (L) and 128K (R) Caches



MP3D Bus Busy Cyles: 32K (L) and 128K (R) Caches

Figure 13: Bus busy times for Gauss and MP3D

6 Related Work

Goodman introduced the concept of a *coherence* block which can be different in size from either an address block (the amount of storage associated with a cache address tag) or a transfer block, which is the amount of data transferred from memory on a miss [Goo87]. Goodman advocates using a large size for the coherence block to reduce the number of coherence operations that must be done to read or write a given amount of data. However, at some point increased traffic from false sharing misses will cause an overall increase in memory latency and bus traffic.

Other authors have proposed *directory* based schemes in which the unit of coherence is smaller than an address block. Chen and Dubois [CD93] describe an extension to Censier and Feautrier's full-map directory protocol in which a valid bit is associated with each invalidation block. They show a substantial decrease in both miss rates and memory traffic when partial invalidations are used.

Dubnicki and LeBlanc [DL92] propose adjustable block size caches which dynamically grow or shrink the size of an address block in response to various patterns of write sharing. A counter is associated with each cache line. The counter is incremented if only half of the block was accessed; if both halves were accessed, the counter is decremented. A block is split when a processor requests part of a block which has been modified in another processor, and the split/merge counter indicates that processors are accessing only parts of the block. Two blocks is merged when both are owned by the same node and both split/merge counters indicate that the blocks should be merged. Unlike Chen's protocol, Dubnicki's protocol is not easily adaptable to non-directory based coherence protocols.

7 Conclusion

In this paper, we have highlighted a cache coherence protocol that attempts to combine the best features of using large and small line sizes. The coherence protocol uses large block sizes for data transfers, yet uses smaller block sizes when doing invalidations in order to avoid false sharing effects. We also used read broadcast to reduce the number of data re-reads caused by invalidations.

We ran this subblock protocol on an instruction-level simulator, to both test its correctness and evaluate its performance. We found that the subblock protocol did as well as the large line size Illinois protocol on applications that exhibit good spatial locality and little sharing. At the same time, the subblock protocol was competitive with the small line size Illinois protocol on the two applications that have large amounts of sharing. We believe that the relatively small amount of additional hardware needed by the subblock protocol (versus using the same line size with a conventional protocol) is justified by the fact that it performs well both for applications that exhibit good spatial locality and for those that have true and false sharing. This is in contrast with protocols with a single line size for coherence and transfer that perform well for one type of applications and poorly for the other.

In future work we will modify the protocol to work in a hierarchical bus system. We plan to evaluate the effectiveness of using different transfer and coherence sizes at different levels in the system. We will also investigate incorporating the detection of migratory sharing into our subblock protocol[SBS93].

References

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM TOCS*, 4(4):273–298, November 1986.

[AB93] Craig Anderson and Jean-Loup Baer. A multi-level hierarchical cache coherence protocol for multiprocessors. In *Proc. of 7th Int. Parallel Processing Symoposium*, pages 142–148, 1993.

[Bro89] Brooks III, E. D., T. S. Axelrod, and G. H. Darmohray. The Cerberus multiprocessor simulator. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 384–390. SIAM, 1989.

[CD93] Yung-Syau Chen and Michel Dubois. Cache protocols with partial block invalidations. In *7th International Parallel Processing Symposium*, pages 16–24, 1993.

[Dar88] Darmohray, G. A. Gaussian techniques on shared-memory multiprocessors. Master's thesis, University of California, Davis, April 1988.

[DL92] Czarek Dubnicki and Thomas LeBlanc. Adjustable block size coherent caches. In *Proc. of 19th Int. Symp. on Computer Architecture*, pages 170–180, 1992.

[DN87] S. Devadas and A. R. Newton. Topological optimization of multiple level array logic. *IEEE Transactions on Computer-Aided Design*, November 1987.

[EJ91] Susan Eggers and Tor Jeremiassen. Eliminating false-sharing. In *Proc. of Int. Conf. on Parallel Processing*, pages I–377–381, 1991.

[EK88] Susan Eggers and Randy Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 373–382, 1988.

[EK89] Susan Eggers and Randy Katz. Evaluating the performance of four snooping cache coherence protocols. In *Proc. of 16th Int. Symp. on Computer Architecture*, pages 2–15, 1989.

[Goo87] James Goodman. Coherency for multiprocessor virtual caches. In *Proc. of 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, 1987.

[Jou93] Norm Jouppi. Cache write policies and performance. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 191–201, 1993.

[KMRS88] Anna Karlin, Mark Manasse, Larry Rudolf, and Daniel Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[MIP91] MIPS Computer Systems. MIPS R4000 microprocessor user's manual, 1991.

[PP84] Mark Papamarcos and Janak Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proc. of 11th Int. Symp. on Computer Architecture*, pages 348–354, 1984.

[SBS93] Per Stenstrom, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 109–118, 1993.

[Sin92] Sing, J., W.-D. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, pages 5–44, March 1992.

[SR84] Z. Segall and L. Rudolph. Dynamic decentralized cache schemes for an mimd parallel processor. In *Proc. of 11th Int. Symp. on Computer Architecture*, pages 340–347, 1984.

[TCS92] Charles Thacker, David Conroy, and Lawrence Stewart. The Alpha demonstration unit: A high-performance multiprocessor for software and chip development. *Digital Technical Journal*, 4(4):51–65, 1992.

[VF92] Jack Veenstra and Robert Fowler. A performance evaluation of optimal hybrid cache coherency protocols. In *Proc. of 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 149–160, 1992.

Appendix A - Subblock Protocol Specification

Processor Cache Status Line Status / Subblock Status	Processor Request			
	READ	WRITE	SWAP	REPLACE
INVALID/Invalid	Allocate line Q Read◇	Allocate Line Q Read Ex	Allocate Line Q Read Ex	None
CLEAN SHARED/Invalid	Q Read	Q Read Ex	Q Read Ex	None
VALID EXCLUSIVE/Invalid	Q Read	Q Read Ex	Q Read Ex	None†
DIRTY SHARED/Invalid	Q Read	Q Read Ex	Q Read Ex	None†
CLEAN SHARED/Clean Shared	Hit	Q Invalidate	Q Invalidate	None
VALID EXCLUSIVE/Clean Shared	Hit	Hit → Dirty	Hit → Dirty	None†
DIRTY SHARED/Clean Shared	Hit	Q Invalidate	Q Invalidate	None†
CLEAN SHARED/Dirty Shared	Hit	Q Invalidate	Q Invalidate	Write Back
DIRTY SHARED/Dirty Shared	Hit	Q Invalidate	Q Invalidate	Write Back
VALID EXCLUSIVE/Dirty	Hit	Hit	Hit	Write Back
DIRTY SHARED/Dirty	Hit	Hit	Hit	Write Back

†If other blocks are dirty, they are written back
◇'Q' means to enqueue a request on the bus

Table 2: Direct Processor Actions

Processor Cache Status Line Status / Subblock Status	Other Request		
	Read	Read Ex	Invalidate
INVALID/Invalid	Snarf → CLEAN SHARED → Clean Shared‡	-	-
CLEAN SHARED/Invalid	Snarf → Clean Shared‡	-	-
VALID EXCLUSIVE/Invalid	-	-	-
DIRTY SHARED/Invalid	Snarf → Clean Shared‡	-	-
CLEAN SHARED/Clean Shared	Supply*	Supply* →Invalid	→ Invalid
VALID EXCLUSIVE/Clean Shared	Supply →DIRTY SHARED	Supply →Invalid	Error
DIRTY SHARED/Clean Shared	Supply*	Supply* →Invalid	→ Invalid
CLEAN SHARED/Dirty Shared	Supply*	Supply* →Invalid◇	→ Invalid◇
DIRTY SHARED/Dirty Shared	Supply*	Supply* →Invalid◇	→ Invalid◇
VALID EXCLUSIVE/Dirty	Supply →DIRTY SHARED →Dirty Shared	Supply →Invalid	Error
DIRTY SHARED/Dirty	Supply →Dirty Shared	Supply →Invalid	Error

‡The snarf occurs iff the tag matches, and another cache (not memory) supplies the subblock
*At most one cache will supply the block
◇The requester will be responsible for writing the subblock back

Table 3: Induced Processor Cache Actions

Processor Cache Status Line / Subblock	Successful Request		
	Read	Read Ex	Invalidate
INVALID/Invalid	(Cache to Cache) → CLEAN SHARED → Clean Shared	(Cache to Cache) → DIRTY SHARED → Dirty	N/A
	(From Memory) → VALID EXCLUSIVE → Clean Shared	(From Memory) → VALID EXCLUSIVE → Dirty	
CLEAN SHARED/ Invalid	(Cache to Cache) → Clean Shared	(Cache to Cache) → DIRTY SHARED → Dirty	N/A
	(From Memory) → Clean Shared	(From Memory) → DIRTY SHARED → Dirty	
VALID EXCLUSIVE/ Invalid	(Cache to Cache) → CLEAN SHARED → Clean Shared	(Cache to Cache) → DIRTY SHARED → Dirty	N/A
	(From Memory) → Clean Shared	(From Memory) → Dirty	
DIRTY SHARED/ Invalid	→ Clean Shared	→ Dirty	N/A
CLEAN SHARED/ Clean Shared	N/A	N/A	→ DIRTY SHARED → Dirty
VALID EXCLUSIVE/ Clean Shared	N/A	N/A	N/A
DIRTY SHARED/ Clean Shared	N/A	N/A	→ Dirty
DIRTY SHARED/ Dirty Shared	N/A	N/A	→ Dirty

Table 4: Processor cache response to successful request