

Two Techniques for Improving Performance on Bus-based Multiprocessors

Craig Anderson and Jean-Loup Baer
Department of Computer Science and Engineering
University of Washington
Seattle WA, 98195

Abstract

In this paper, we explore two techniques for reducing memory latency in bus-based multiprocessors. The first one, designed for sector caches, is a snoopy cache coherence protocol that uses a large transfer block to take advantage of spatial locality, while using a small coherence block (called a subblock) to avoid false sharing. The second technique is read snarfing (or read broadcasting), in which all caches can acquire data transmitted in response to a read request to update invalid blocks in their own cache.

We evaluated the two techniques by simulating 6 applications that exhibit a variety of reference patterns. We compared the performance of the new protocol against that of the Illinois protocol with both small and large block sizes and found that it was effective in reducing memory latency and providing more consistent, good results than the Illinois protocol with a given line size. Read snarfing also improved performance mostly for protocols that use large line sizes.

1 Introduction

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 two such techniques in the context of a cache coherent shared-memory multiprocessor architecture. For the first technique, we focus on the relationship between the coherence protocol and the cache block size. While some applications run better when small cache block sizes are used because of

the presence of migratory data or because false sharing is avoided, others execute more quickly with larger block sizes because they exhibit good spatial locality. Most often, an architecture is implemented with the same block size used for both transfer (memory to cache or cache to cache) and coherence. Our technique is a snoopy cache coherence protocol that uses different block sizes for transfer and coherence. The goal of the subblock protocol is to obtain the advantages of using large block sizes for those programs that have good spatial locality while avoiding many unnecessary invalidations that result from migratory data and false sharing. The subblock protocol performance is compared to that of an invalidate protocol, namely the Illinois protocol.

The other technique we evaluate is read snarfing. Read snarfing is an enhancement to snoopy-cache coherence protocols that takes advantage of the inherent broadcast nature of busses. In a system with read snarfing, caches snoop the bus not only for coherence transactions, but also for potentially useful data that is being read by another processor. In this paper, we evaluate the impact that snarfing has on the performance of the Illinois protocol and on our newly developed subblock protocol.

In Section 2 we present our model architecture, in particular a sector cache organization that allows subblocks. Sections 3 and 4 describe the two techniques, subblock protocol and read snarfing, in more detail. Section 5 presents our evaluation methodology with Section 6 giving simulation results. 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. 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 block size L , and an associativity k . Corresponding sector caches have the same (C, L, k) characteristics but in addition the blocks 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 blocks and subblocks have states. A sector cache (C, L, k, b) will therefore require $2L/b$ extra state bits/block, 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 [Sez94].

Systems with sector caches also require more bus lines to transmit bitmasks corresponding to the status of the subblocks in a particular block. 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 block.

We assume caches with dual ported tags, a common feature in current snoop-based systems and an almost mandatory one when snarfing is implemented so that read snarfing snoop checks don’t interfere with processor access.

3 The subblock protocol

3.1 Subblock protocol

The motivation behind sector caches is to design a coherence protocol that takes advantage of applications’ spatial locality by fetching large sized blocks 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] 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 block as possible. If the needed subblock is in another processor’s cache, that cache sends along not only the cached subblock, but all other read-shared subblocks (i.e., subblocks in states *Clean Shared* and *Dirty Shared*, see below) in that block. In contrast with the Illinois protocol, dirty subblocks that are transferred on read misses are not copied to memory (hence the *Dirty Shared* state for ownership). On the other hand, if the subblock is located only in main memory, then memory will respond with the requested subblock, plus any subblocks which are not currently cached. The information about which subblocks are currently cached is determined when the request goes out over the bus. On writes to clean subblocks and write misses, we invalidate only the subblock to be written. We do this in order to avoid the performance penalties associated with false sharing.

To implement our protocol, we used 4 block states, plus 4 subblock states. The block states are useful when there is only one cache that contains a copy of the block (state **VALID EXCLUSIVE**) thus avoiding invalidations, or when all subblocks in the block are either clean – and not owned – or invalid (state **CLEAN SHARED**), thus avoiding replacement. The block states are as follows:

INVALID All subblocks are *Invalid*.

VALID EXCLUSIVE All valid subblocks in this block 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 block may be *Invalid*. There also may be *Dirty* blocks which must be written back upon replacement.

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

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

The subblock states are as follows:

Invalid The subblock is not valid.

Clean Shared A read access to the subblocks will succeed. Unless the block 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.

Additional detail on the subblock protocol, including complete tables of state transitions and an example of the protocol in action may be found in [AB94].

3.2 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 a 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 where the size of an address block dynamically grows or shrinks, in a buddy-system like fashion, in response to various patterns of write sharing. Unlike Chen's protocol, Dubnicki's protocol is not easily adaptable to non-directory based coherence protocols. Tomašević and Milutinović [TM91] advocate a Word Invalidate Protocol (WIP) that uses per-word valid bits and additional block states to reduce the "pollution" of valid blocks with invalid words. Using a synthetic benchmark to evaluate their protocol, they found that the WIP protocol overall performed slightly better than the the write-update Dragon protocol for their parameters.

4 Snarfing

Read snarfing is an enhancement to snoopy-cache coherence protocols that takes advantage of the inherent broadcast nature of the bus. Under read snarfing,

the cache snoop mechanism of every cache, and not only of the requesting cache, checks the address associated with all completed read operations on the bus. If the address of the data matches in the cache, but the state of the block (subblock) is invalid, the cache will read the value of the block from the bus, and set the state of the block (subblock) to read shared (clean-shared in the subblock protocol). The main benefit of read snarfing is that it reduces the number of read misses for blocks that were invalidated because of coherence actions. The first processor to reload a block because of an invalidation miss will effectively supply the block to other processors whose blocks were invalidated in the past, assuming that the invalidated blocks were not overwritten by other data.

We expect that read snarfing will enhance the overall performance of the system (reduction in elapsed execution time and in bus utilization). Note however that snarfing can produce an "anomaly", namely an increase in the miss rate (this actually happened in two of the benchmarks discussed later). To see how this can happen, consider a system that uses test and test and set (TTS) locks. The code for TTS locks is as follows:

```
L1: (1) Load L (L is the lock location)
      (2) If L != 0, goto L1
      (3) Load 1 into a register
      (4) Atomically swap L with register
      (5) If register != 0, goto L1
      (Lock succeeds)
```

To unlock the lock, the holder simply swaps a 0 into the lock location.

Each time a lock is released and acquired, the value of the lock transitions from 1 to 0, then back to 1. A processor unsuccessfully contending for the lock would see 2 misses - one for each transition. One miss would be a read miss at statement (1) which is caused by the transition from 1 to 0, and the other would be a swap (write) miss at statement (4) since the processor successful in getting the lock would have invalidated all other copies. This is the case when bus and lock contention are low.

However, when bus contention is high, it might take a long time for the load that results from the first miss to be completed. The load could be delayed so long that the value loaded is 1, not 0. The processor would only tally one miss (a read miss), not two, as a result of the lock being released and acquired.

When snarfing is used, the processor's read request is satisfied quickly, since one of the contending reads succeeds even under high bus loading. Because of this,

the processor will tally more misses under snarfing. It tallies more write misses, since all processors contending for the lock fall through to statement (5), unlike the case when there is no snarfing and high bus contention. It also tallies more read misses, because under snarfing, all processors contending for the lock reach statement (4) at about the same time. The first contending processor to get the bus will succeed in taking the lock; all others will fail. The first processor to fail (call it A) will branch back to L1, where it spins. The second processor to fail will then execute statement (4), invalidating the copy in A’s cache, which causes a read miss. Depending on the interleaving of requests on the bus, processor A may experience one or more read misses due to failed attempts to take the lock. This undesirable behavior could be avoided by using a queue lock which works well under both heavy and light contention.

Despite the additional misses caused by snarfing, using it can reduce overall execution time and bus loading, because most of the extra misses incurred do not require a bus access to satisfy them; the value of the lock is reloaded through snarfing.

Read snarfing was discussed in previous work by Rudolph [SR84], Goodman [GW88], and Eggers [EK89]. Of these, only Eggers evaluated the performance benefits of read snarfing. Under a single-ported tag directory assumption she found that although read snarfing resulted in a decrease in invalidation misses and bus utilization for most applications, total execution time increased, due to processor lock-out from the cache and the need to for caches to acknowledge that they have completed a snoop. With our assumption of dual-ported tag directories (c.f. Section 2), the processor lock-out problem is no longer present.

5 Methodology

5.1 Benchmarks

We modified the Cerberus multiprocessor instruction-level simulator [BAD89, AB93] to evaluate both the sector cache protocol and snarfing. 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 two improvements but also to check that they were working correctly 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 latency of main memory was assumed to be 10 cycles.

We chose six applications to evaluate our protocol. The applications’ reference behaviors fall into three categories:

- Applications that have good spatial locality and exhibit high cache hit rates, and therefore do well with large block sizes (Gauss, Cholesky)
- Applications that exhibit both true and false sharing, that perform better with small block sizes (MP3D, Topopt, Pverify)
- Application that have few shared accesses, so the choice of block size is less important (Barnes)

Gauss is a standard gaussian elimination program which solves a system of linear equations. We ran our simulation on a 250x250 matrix, comprising approximately 161 million instructions on a single processor.

Cholesky is a program from the SPLASH benchmark suite for sparse matrix factorization [SWG92]. Our simulation used the BCSSTK14 input file (65 million instructions).

Another program from the SPLASH suite is MP3D. MP3D is a 3-dimensional particle simulator which has poor locality and incurs a great deal of coherence misses. We ran MP3D on 50000 molecules for 50 steps (21 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 (1.7 billion instructions).

Pverify determines whether two boolean circuits are functionally identical [MDWS87]. It exhibits a great deal of false sharing, even when using moderate-sized blocks [EJ91]. We ran Pverify on the circuits C880.21.berk1 and C880.21.berk2 (1 billion instructions).

Our final applications is Barnes-Hut, from the SPLASH suite. It simulates the effect of gravity on a system of bodies. Barnes exhibits a very low coherence miss rate, even when 64 processors are used. We simulated 128 bodies (111 million instructions)

5.2 Simulated architectures

Our experiments simulated both usual and sector caches. In all experiments the subblock size b was 8 bytes and caches were two-way set-associative ($k = 2$).

In order to minimize simulation time, we only simulated relatively large caches ($C = 128K$). Previous

work [AB94] indicates that using a smaller cache size does not significantly change the results. The block size was either 8 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 block size of 8 bytes).

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

5.3 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. The two most important ones are the number of (shared) data bytes transferred during execution, and the number of bus transactions needed. These metrics are important not only because they allow us to understand why one protocol performs better than others, but also because they can be used to project how technology advances will affect our results. For example, we can project the impact of faster processors or of wider busses by expressing the total elapsed time as the sum of the “computational cycles” and of the “bus cycles”, with differing values for these “cycles” and a factor for varying bus contention. We are currently building a model that will allow us to predict performance in this fashion. The model will be validated by running the instruction-level simulator with adequate parameters.

6 Results

We report our results in three groups, corresponding to the classification used in section 5.1. First, we present results for the two applications (Gauss and Cholesky) that do well using large block sizes. Next, we present data for the three applications (MP3D, Topopt, and Pverify) that perform better using smaller block sizes. Finally, we report on our results running Barnes, for which the choice of block size is not very important. Note that in the following discussion, “8I protocol” refers to the Illinois protocol used with an 8 byte block size; similarly, “64I proto-

col” refer to the Illinois protocol and a 64 byte block size.

6.1 Gauss

The Gaussian elimination application has little write-shared data, and good spatial locality. Therefore, using a large block size provides important performance benefits (see Figure 1A). Running Gauss on the 64I protocol results in linear speedup up to 16 processors, and a speedup of about 21 on 32 processors. In contrast, the 8I protocol peaks at a speedup of about 10 using 16 processors. The reason for this is that the 8I protocol generates over 7 times as many bus requests as the 64I protocol, both for small and large numbers of processors (Figure 1C). This more than makes up for the small reduction (from 7% to 13%) in the number of bytes transferred when using the 8I protocol (Figure 1B).

The 64 byte subblock protocol outperformed even the 64I protocol on the Gaussian application. Because the subblock protocol transfers as much of a cache block as possible when satisfying read requests, it took advantage of the spatial locality in Gauss, which reduced the number of bus transactions needed. At the same time, the subblock protocol reduced the number of bytes transferred when compared to the 64I protocol, to a level in between that of the 64I and 8I protocol.

Snarfing had a small but beneficial impact on program performance, mostly for the 32 processor configuration. Snarfing reduced the number of bytes transferred for the 64I configuration by 5%; for other configurations, the figure is smaller. This is not surprising, given the limited number of opportunities available for snarfing.

6.2 Cholesky

While linear speedup was not reached in Cholesky, this application actually exhibits good spatial and processor locality; the lack of speedup is due to the small example that was simulated (Figure 2A). Because of its good locality, there was little difference in overall speedup between the various protocols simulated. The 64I protocol did run slightly faster than the 8I protocol, primarily due to the fewer bus transactions needed by the 64I protocol (Figure 1C). The 8I protocol required about 5 times more bus transactions than the 64I protocol to complete the program, whether for 1 processor or 32 processors. These additional bus transactions caused bus utilization to roughly double for the 8I protocol, despite the fact that using a

- △ ·····△ 64 byte subblocks/No Snarfing
- ◇ - - -◇ 64 subblocks/Snarfing
- + - - + 64 byte Illinois/No Snarfing
- × ·····× 64 byte Illinois/Snarfing
- * - - * 8 byte Illinois/No Snarfing
- ·····○ 8 byte Illinois/Snarfing

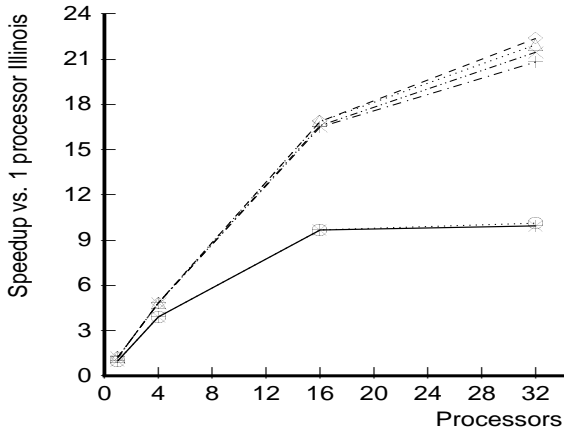


Figure 1A: Gauss (128K Caches)

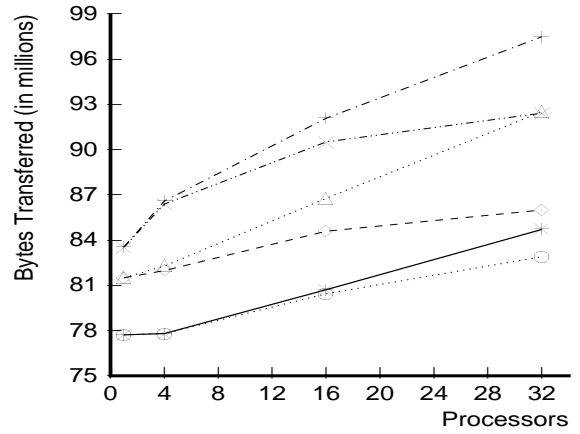


Figure 1B: Gauss (128K Caches)

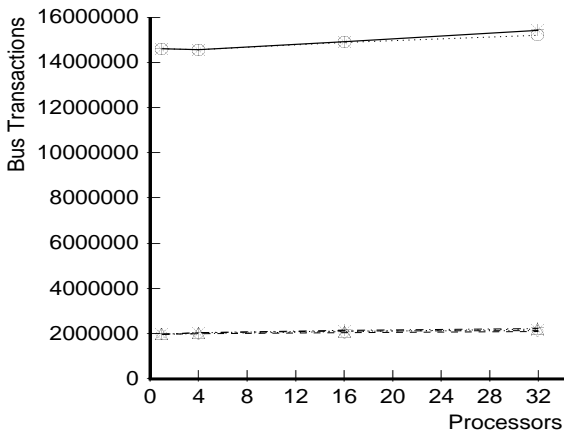


Figure 1C: Gauss (128K Caches)

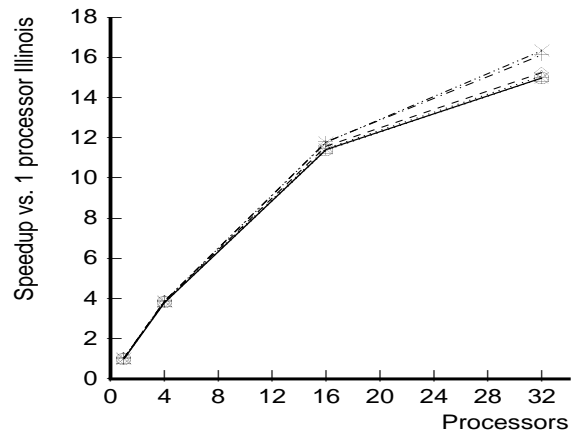


Figure 2A: Cholesky (128K Caches)

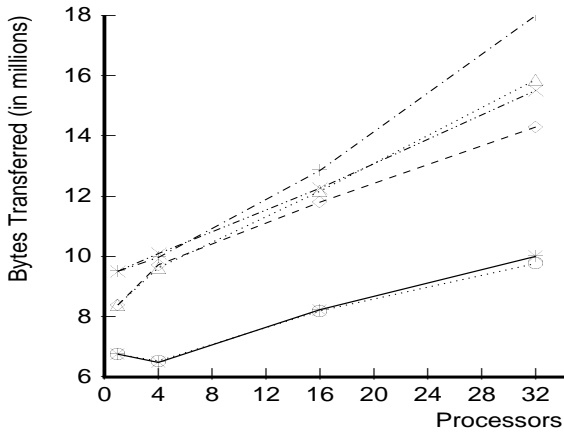


Figure 2B: Cholesky (128K Caches)

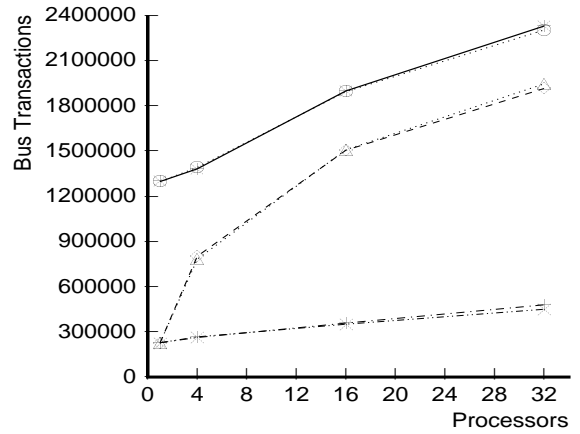


Figure 2C: Cholesky (128K Caches)

smaller block size reduced the number of bytes transferred by at least 33% when using 4 or more processors (Figure 2B).

The subblock protocol performance was closer to that of the 8I protocol than to that of the 64I protocol. The reason for this is the large number of bus transactions required by the subblock protocol: 20% less than the 8I protocol but up to 4 times more than the 64I protocol. The subblock protocol required many more invalidations than the 64I protocol, since invalidations are done on the (smaller) subblock level. It also required more read requests than the 64I protocol. These additional requests were needed on cache to cache transfers, despite the fact that when a cache supplies a subblock in response to a read request, it sends along all other read-shared subblocks. From this we can infer that a great many of the subblocks in each cache are owned by that cache, necessitating a separate read request for each subblock needed.

Overall, snarfing provided relatively small benefit to this application. It did reduce the number of bytes transferred by all three protocols, but only by 13% in the best case. It slightly reduced the number of bus transactions, and had a very small impact on bus utilization.

6.3 MP3D

MP3D has a great deal of both true and false sharing [CGM91, EJ91]. Such sharing tends to limit speedups as more processors are devoted to the problem. Because of MP3D's sharing behavior, using a large block size severely impacts performance when using 16 or 32 processors, as can be seen in Figure 3A for the 64I protocol. Speedup in that case reaches a maximum of 5.77 for 16 processors, a figure already quite low, and actually decreases to 5.33 for 32 processors. By comparison, the small block 8I protocol has speedups of 10.15 for 16 processors and 12.15 for 32 processors (saturation has been reached). The difference in performance is due to the increased amount of data transferred under the 64I protocol (Figure 3B). For 16 processors, the 64I protocol transfers 8 times more data than the 8I protocol with only 17% fewer transactions; for 32 processors, the factors are 8.8 and 12%. The additional traffic hurts performance, since even under the 8I protocol, bus utilization is already high (60% or greater).

Running MP3D under the 64 byte subblock protocol did improve performance over the 64I protocol, but the subblock protocol didn't match the performance gains obtained by using the 8I protocol. The primary reason for this is that the subblock protocol

attempts to transfer as much of a block as practicable when satisfying reads. While this helps performance on applications with a great deal of spatial locality (like Gauss), it results in additional unnecessary data traffic in MP3D. The amount of data transferred is about 3 times that of the 8I protocol and 40% that of 64I. The number of bus transactions is also in between those of 64I and 8I. And, not surprisingly, the speedups are also in between those of the two other protocols.

Snarfing is moderately effective in this application, especially in the 64I protocol. Under the 64I protocol, speedup does not degrade when going from 16 to 32 processors when snarfing is used; moreover, snarfing improves the performance of the 64I protocol by 25% for the 16 processor case, and 30% for the 32 processor case. However, even with snarfing, the 64I protocol is not competitive with the 8I protocol. With large block sizes, it is more likely that two (or more) processors are accessing data in the same block. Thus it is more likely with larger block sizes that a snarfed block will actually be used. This can be seen in Figure 3B, where snarfing decreased the number of bytes transferred under the 64I protocol by 35% for 16 processors, and by 40% for 32 processors. In addition, snarfing reduced 64I's number of transactions by about 17%. These two effects contribute directly to an improvement in performance, since high bus latency due to contention is the bottleneck for this application. In contrast, snarfing reductions for the 8I protocol are, percentage-wise, only half that of the 64I protocol: the amount of data transferred is reduced by only 22% and the number of transactions by 9%. This results in speedup increases of only 5% (for 16 processors) to 10% for 32 processors. Snarfing also improved the performance of the subblock protocol. Just as the subblock protocol's performance was in between that of the 8I and 64I protocols, the benefits the subblock protocol obtained using snarfing were also in between those of the 8I and 64I protocols.

6.4 Toptopt

The Toptopt application exhibits both true and false sharing. Because of this, using an 8 byte block size resulted in much better speedup performance than using a 64 byte block size (see Figure 4A). Using a small block size resulted in near-linear speedup, while using a large block size achieved a speedup of only 10 on 16 processors, and 17 on 32 processors. The effect of false sharing can be seen in graphs that show the number of bytes transferred (Figure 4B), and the number of bus transactions (Figure 4C). The 64I protocol transferred

- △····△ 64 byte subblocks/No Snarfing
- ◇---◇ 64 subblocks/Snarfing
- +---+ 64 byte Illinois/No Snarfing
- ×····× 64 byte Illinois/Snarfing
- *---* 8 byte Illinois/No Snarfing
- 8 byte Illinois/Snarfing

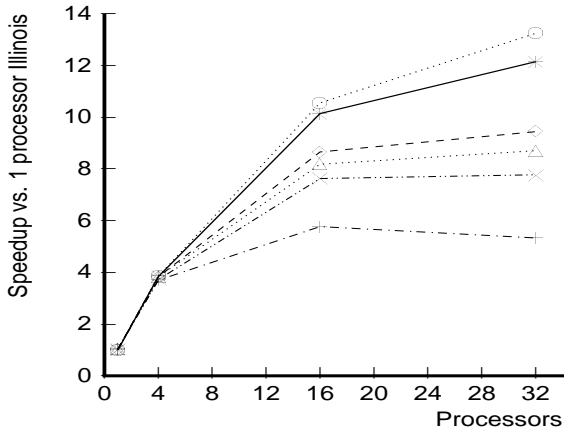


Figure 3A: MP3D (128K Caches)

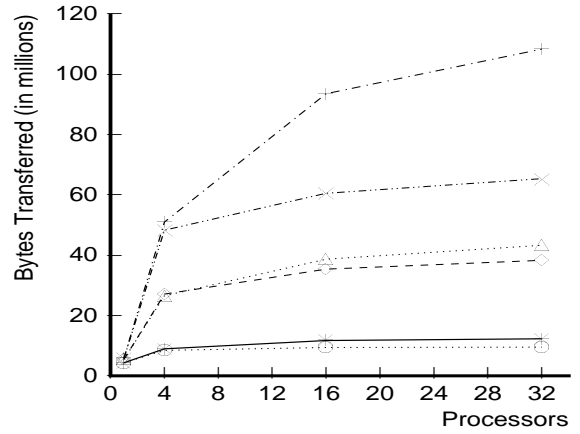


Figure 3B: MP3D (128K Caches)

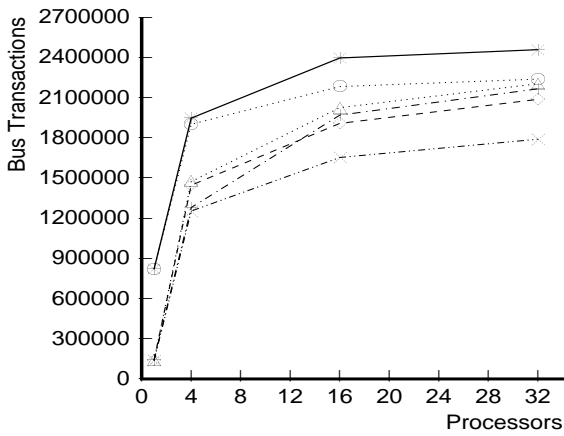


Figure 3C: MP3D (128K Caches)

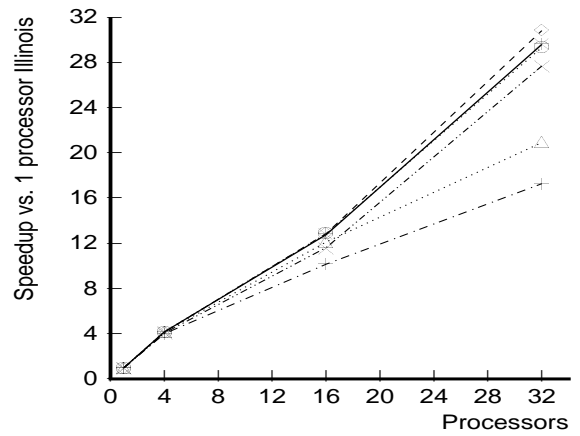


Figure 4A: Topopt (128K Caches)

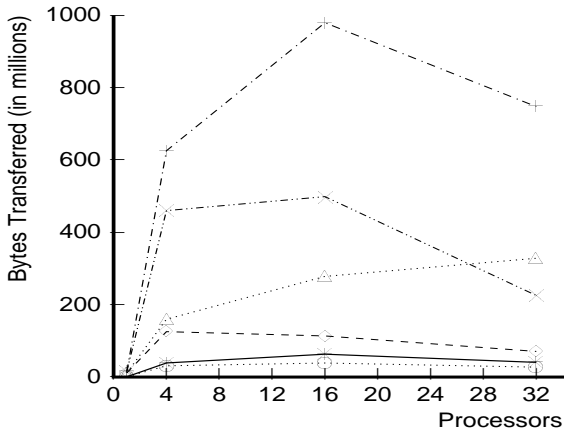


Figure 4B: Topopt (128K Caches)

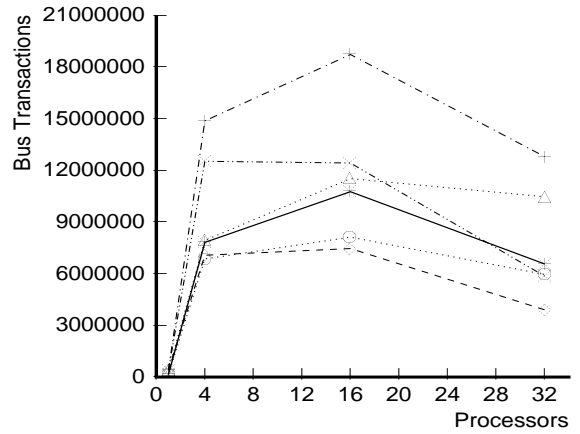


Figure 4C: Topopt (128K Caches)

from 15 to 18 times more data than the 8I protocol when more than 1 processor was used. The 64I protocol also used nearly twice as many bus transactions than the 8I protocol. The decrease in the number of bytes transferred (and bus transactions) for the 64I protocol when 32 processors were used is due to the decrease in capacity/conflict misses.

The performance of the subblock protocol on Topopt was mediocre. Without snarfing, the subblock protocol outperformed only the 64I protocol (without snarfing) on speedup. The subblock protocol also transferred far more data and incurred more bus transactions than the 8I protocol, though it did perform better than 64I protocol in these measures. Part of the reason why the subblock protocol performed poorly in the absence of snarfing is its policy of transferring additional subblocks on cache-to-cache reads in addition to the one requested.

The addition of snarfing to the 64I protocol was effective in narrowing the performance gap between the 64I and 8I protocols. Snarfing increased the speedup for the 32 processor 64I case from 17 to 28. Snarfing cut the number of bytes transferred by a factor of 2 for 16 processors, and by a factor of 3 for 32 processors. It also significantly reduced the number of bus transactions needed. Snarfing reduced the detrimental effect of false sharing on the performance of the 64I protocol. In contrast, snarfing had little effect on the 8I protocol. Snarfing significantly improved the performance of the subblock protocol. With snarfing, the subblock protocol finished faster than either the 8I or 64I protocols. The subblock protocol also did nearly as well as 8I protocol in minimizing the number of bytes transferred, and had fewer bus transactions than the other protocols.

6.5 Pverify

Pverify has great deal of false sharing, even for moderate blocks sizes [EJ91]. It is not surprising, therefore, that using a large block size substantially degrades performance, as can be seen in Figure 5A. In contrast, using a small block size leads to near-linear speedup. The poor performance of the 64I protocol is due to the increased number of invalidations, and the bus traffic needed to satisfy the read misses which result from the invalidations. As can be seen in Figure 5B, using the 64I protocol (without snarfing) transfers over 16 times as much data as the corresponding 8I protocol when 32 processors are used. Using a large block size also increases the number of bus transactions (by 76% for 32 processors) required to complete the application (Figure 5C).

Using the subblock protocol substantially improved the performance of Pverify as compared to the 64I protocol, nearly matching the 8I protocol in speedup. The reason for this can be seen in Figures 5B and 5C, where the subblock protocol reduced both the number of bytes transferred by 65% and the number of bus transactions by 29% (for 32 processors) when compared to the 64I protocol. One interesting aspect to Figure 5B is the drop in the number of bytes transferred under the subblock protocol as the number of processors increases. This is due to the fact that as the total amount of cache in the system increases, capacity and conflict misses decrease, necessitating fewer reads from memory. While this phenomenon also occurs under the 8I and 64I protocols, the effects are masked by the increase in reads due to invalidation misses.

Snarfing was a large benefit to the 64I protocol, while it was only a small benefit for the 8I protocol. For 16 processors or less, the speedup benefit was small for all protocols (at most 10%). However, at 32 processors, snarfing improved the speedup of the 64I protocol by 67%, which is still not enough to make the 64I protocol competitive with the other two protocols. The improvement is due to large reductions in both the number of bytes transferred, and the number of bus transactions. Snarfing reduced the number bytes transferred by 35% for 16 processors, and 43% for 32 processors. Improvements for the number of transactions was smaller: 24% for 16 processors, and 33% for 32 processors. Gains for the 8I protocol were much smaller. There was little difference in speedup, except for 32 processors (6%). Snarfing did improve the number of bytes transferred for the 32 processor by 24%, and the number of transactions by 20%. We observed an anomaly in our results when we looked at snarfing under the subblock protocol. Contrary to what we expected, snarfing slightly decreased performance in all three categories of speedup, bytes transferred, and bus transactions. We are investigating the cause of this unusual result.

6.6 Barnes

Overall, the Barnes-Hut application exhibits good locality, primarily because it does relatively few shared read (10-15%) and shared write references (.1%). Because of this, varying the block size, or using the subblock protocol or snarfing had little effect on overall speedup (Figure 6A). However, if we look at the number of bytes transferred (Figure 6B), and the number of bus transactions (Figure 6C), then we do see differences between the tested configurations. Since the working set for the application fits in a single 128K

- △ ····· △ 64 byte subblocks/No Snarfing
- ◇ - - - ◇ 64 subblocks/Snarfing
- + - - + 64 byte Illinois/No Snarfing
- × ··· × 64 byte Illinois/Snarfing
- * - - * 8 byte Illinois/No Snarfing
- ··· ○ 8 byte Illinois/Snarfing

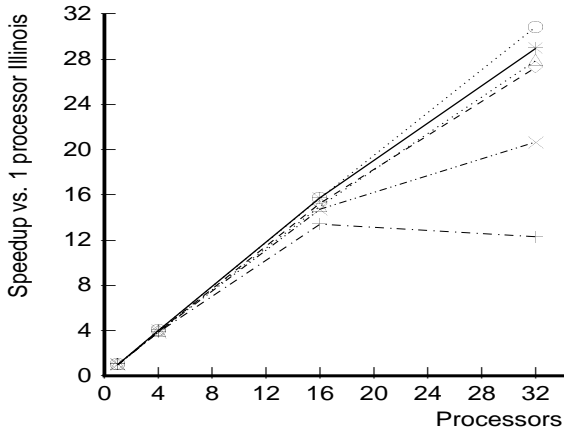


Figure 5A: Pverify (128K Caches)

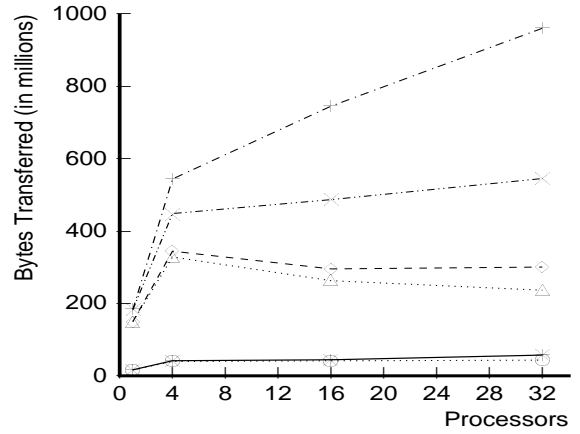


Figure 5B: Pverify (128K Caches)

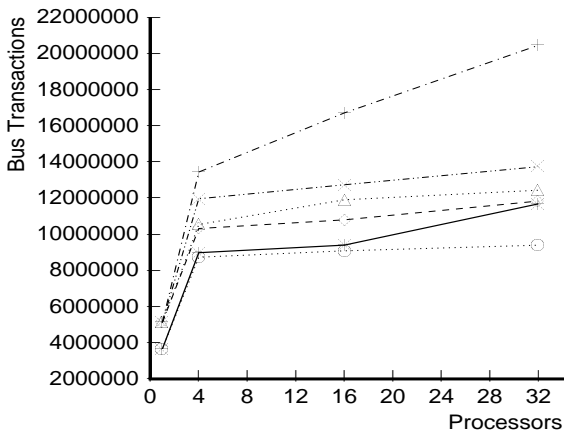


Figure 5C: Pverify (128K Caches)

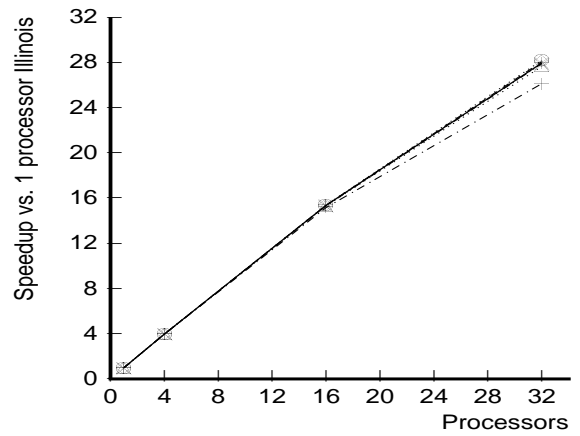


Figure 6A: Barnes (128K Caches)

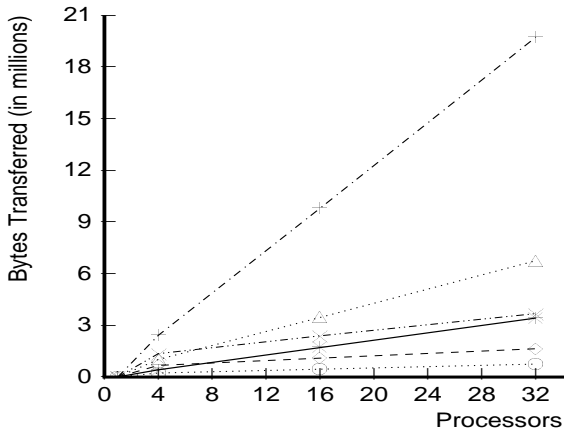


Figure 6B: Barnes (128K Caches)

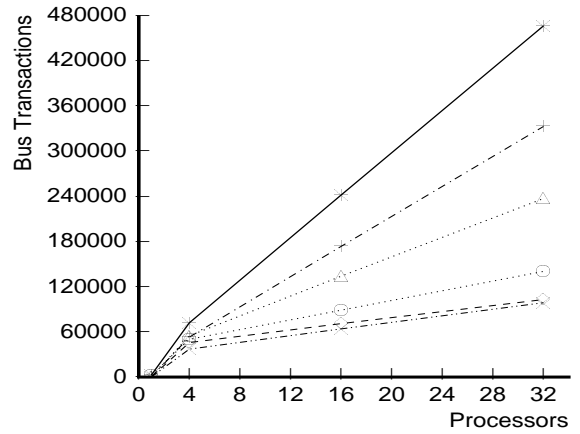


Figure 6C: Barnes (128K Caches)

cache, most of the bus traffic is due to invalidations and the subsequent misses due to invalidations. As we can see in Figure 6B, using the 64I protocol resulted in over 5 times more bytes transferred than the 8I protocol. While using the 64I protocol did reduce the number of bus transactions by 25%, the large number of bytes transferred under the 64I protocol resulted in much higher bus utilization than the 8I protocol.

The subblock protocol performed well on the Barnes application. It substantially reduced the number of bytes transferred over the 64I protocol, approaching the performance of the 8I protocol. It used fewer bus transactions than either the 64I or 8I protocols. And its bus utilization was lower than either of the two Illinois protocols.

Snarfing produced large benefits for all three protocols studied. Using snarfing with the 64I protocol reduced the number of bytes transferred to nearly the level of the 8I protocol which did not use snarfing. Adding snarfing to the 8I protocol reduced the number of bytes transferred by about 75% for the 16 and 32 processor systems. Snarfing also reduced the number of bus transactions. For 32 processors, snarfing reduced the number of transactions by about 70% for both the 8I and 64I protocols, and by 57% for the subblock protocol. Snarfing dramatically reduced bus utilization for the 8I (from 17% to 4% for 32 processors) as well as that of the 64I protocol (from 31% to 7%) and the subblock protocol (14% to 4%).

6.7 Results summary

Overall, the subblock protocol performed well. On four of the six applications (Gauss, Cholesky, Pverify, and Barnes) it executed about as fast (or faster than) the better of two block sizes used with the Illinois protocol. The other two applications (MP3D and Toptopt) performed better using a small block size. However, on Topopt the subblock protocol used with snarfing was faster than all other configurations tested. The subblock protocol did not do well on MP3D because of its policy of transferring multiple subblocks when responding to read requests, a feature that, by contrast, was found to be very beneficial for applications with good spatial locality. On every application, the subblock protocol transferred fewer bytes than the 64I protocol, but more than the 8I protocol. Similarly, for bus transactions, the subblock protocol usually performed in between that of the best choice of block size to reduce bus transactions (64I for Gauss, Cholesky, and MP3D; 8I for the Pverify) and the worst choice; for Barnes, it actually outperformed both.

Generally speaking, snarfing is an effective technique to reduce the amount of data transferred and the number of bus transactions, and to improve execution time, especially when the application does not exhibit good spatial locality. Snarfing greatly improved the performance of the 64I protocol on those application with a great deal of sharing (MP3D, Topopt, Pverify). Snarfing also sharply reduced the amount of data transferred, the number of bus transactions, and bus utilization for Barnes.

7 Conclusion

In this paper, we have presented two techniques for improving application performance on bus-based multiprocessors. The first technique was the use of a subblock cache coherency protocol that takes advantage of the spatial locality found in many programs by using a large sized block for transfers, while avoiding the performance problems caused by false sharing by using a smaller sized subblock for invalidations. The second technique was read snarfing whose intent is to reduce the number of read misses caused by previous coherence actions.

The subblock protocol performed as well or better than the Illinois protocol using the best block size (64 bytes) on those applications with good spatial locality. The subblock protocol also performed well on the applications that exhibited a great deal of true and false sharing. For these applications, it was always better than the Illinois protocol with large block size and was close to the Illinois protocol with small block size with the exception of the MP3D benchmark. When the subblock protocol did not do as well as the Illinois protocol (when using a small block size), the problem was due to the subblock protocol's use of a large transfer block. Our experience shows the difficulty of creating a non-adaptive cache coherency protocol that works well with all applications. The policy of using a large transfer size in our subblock protocol that helped us get good performance in the Gauss application hurts performance in MP3D. Overall, however, the subblock protocol was either best or close to the best of all protocols.

We also investigated the effects of read snarfing, and found that it is an effective technique in improving application performance on bus-based machines, especially when large block sizes are used and the program exhibits a great deal of false sharing. Snarfing is effective in reducing the amount of data transferred and the number of bus transactions, which in turn leads to lower bus utilization and higher application speedups.

The combination of the subblock protocol and snarfing produced the best results in 2 out of 6 cases; of the other 4 cases, it came in a close second in 3 cases. In only one (MP3D) did the subblock protocol ever perform more than 11% worse than the best choice of Illinois protocol (with snarfing).

In future work, we will apply the two techniques to systems that consist of a hierarchy of busses. We will evaluate how effectively the subblock protocol and read snarfing can reduce the amount of bus traffic generated by the application so that more processors can be effectively utilized by the program.

Acknowledgement

This work was supported by NSF grants CCR-91-01541, CCR-91-23308, and CCR-94-01689.

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 Symposium*, pages 142–148, 1993.
- [AB94] Craig Anderson and Jean-Loup Baer. Design and evaluation of a subblock cache coherence protocol for bus-based multiprocessors. Technical Report 94-05-02, University of Washington, 1994.
- [BAD89] E.D. Brooks III, 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.
- [CGM91] David Cheriton, Hendrik Goosen, and Philip Machanick. Restructuring a parallel simulation to improve cache behavior in a shared memory multiprocessor: A first experience. In *Proc. of the Intl. Symp. on Shared Memory Multiprocessing*, pages 109–118, 1991.
- [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.
- [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.
- [GW88] James R. Goodman and Phillip J. Woest. The Wisconsin Multicube: A new large-scale cache coherent multiprocessor. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 422–431, 1988.
- [Jou93] Norm Jouppi. Cache write policies and performance. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 191–201, 1993.
- [MDWS87] H-K. T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli. Logic verification algorithms and their parallel implementation. In *Proceedings of the 24th Design Automation Conference*, pages 283–290, 1987.
- [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.
- [Sez94] André Seznec. Decoupled sector caches: Conciliating low tag implementation cost and low miss ratio. In *Proc. of 21th Int. Symp. on Computer Architecture*, pages 384–393, 1994.
- [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.
- [SWG92] J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, pages 5–44, March 1992.
- [TM91] Milo Tomašević and Veljko Milutinović. A simulation study of snoopy cache coherence protocols. In *Proc. of the 25th Hawaii Int. Conference on System Sciences*, pages 427–436, 1991.