Two Adaptive Hybrid Cache Coherency Protocols

Craig Anderson
Apple Computer
1 Infinite Loop, MS 301-4G
Cupertino, CA 95014

Anna R. Karlin
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

Abstract

We present and evaluate adaptive, hybrid cache coherence protocols for bus-based, shared-memory multiprocessors. Such protocols are motivated by the observation that sharing patterns vary substantially between different programs and even cache blocks within the same program. Performance measurements across a range of parallel applications indicate that the adaptive protocols we present perform well compared to both Write-Invalidate and Write-Update protocols.

1 Introduction

In a bus-based multiprocessor, bus contention can lead to increased program execution time because a processor may stall while its cache is waiting for the bus. For many interesting applications, the bus transactions required to maintain cache coherence are the main source of contention. It is known that no single, non-adaptive cache coherence protocol minimizes bus contention for all applications or even for all cache blocks within the same application. In this paper, we present and evaluate new coherence protocols that adapt to application behavior on a block by block basis.

There are two commonly used coherence protocols: *write-invalidate* (WI) and *write-update* (WU). In write-invalidate, a processor invalidates all other cached copies of shared data and can then update its own copy without further bus operations. In write-update, a processor broadcasts updates to shared data to other caches, so the caches stay coherent.

Clearly, there are reference patterns for which these protocols are suboptimal in terms of bus busy cycles. For example, suppose that two processors are alternately writing to the same cache block. Then each write will result in an invalidation, so that a subsequent block read is required. For a sequence of n such alternate writes, the write-invalidate protocol results in n invalidates and n block reads, whereas on the same sequence the write-update protocol results in only n updates. If a block read takes several more bus cycles than an update, this difference will be significant.

On the other hand, suppose that two processors initially read from some cache block and subsequently only one of the processors actively uses the block, say writing to it n times. Then write-update broadcasts each of these updates for a total cost of two initial block reads and n updates. On this same sequence write-invalidate incurs the cost of two initial block reads and only one subsequent invalidate.

A key program characteristic underlying these examples is the *block write run*, which is defined as a sequence of write references to a shared cache block, uninterrupted by either an access to that block by another processor, or replacement of the block [8]. In the first example above, there were many write runs of length one; in the second, a single write run of length n. In general, short write runs favor write-update protocols, while long write runs favor invalidate based protocols. On the other hand, short write runs can lead to poor performance on machines using invalidate-based coherency (likewise long runs and update strategies).

In an attempt to avoid the pathological behaviors of update and invalidate protocols, *hybrid* protocols which combine both strategies have been proposed. Such strategies usually consist of updating for some number of consecutive writes to a block by the same processor, and then invalidating on the next write. To do this, a counter is associated with each cache block. The initial value of the counter is set to the *invalidation threshold.* On an update transaction, the counter for the block being written is decremented. When the block's counter reaches 0, the block is invalidated in all caches except that of the writing processor, updates no longer need to be sent, and the block in the writer's cache becomes exclusive.

The choice of invalidate threshold is critical to the performance of hybrid protocols. If the value is too small, blocks with short write runs will be invalidated too early; too large, and many useless updates will be done for blocks with long write runs. Karlin *et al.* proposed an algorithm whose choice of invalidate threshold did no worse than twice the bus cycle cost of the optimal off-line algorithm [12]. However, it often did not achieve the best performance. One reason why is that using a single, fixed value of invalidate threshold for all blocks and for all applications necessitates a compromise, since applications have widely varying write run distributions (which we show in section 4).

Our solution to this problem is to use an *adaptive* hybrid protocol that varies the invalidate threshold on a per-block basis over time. The adaptive strategies we consider use write-run lengths collected on-the-fly in order to determine the invalidation threshold for that block in the future. For example, if very long write-runs are observed, the adaptive strategy should set the invalidation threshold to 0, i.e. implement WI. On the other hand, if very short write-runs are observed, the adaptive strategy should set the invalidation threshold sufficiently high that only updates are issued. In other words, the adaptive strategies should have the potential to converge to near-optimal performance on a cache block by cache block basis.

In this paper, we present two adaptive coherency protocols. We show that across a range of parallel applications that we have measured, their performance is as good as the better of WU and WI, and on some significantly better. Moreover, both adaptive protocols generally had low read and write miss rates, and low numbers of bus transactions and bytes transferred.

The rest of the paper is organized as follows. Section 2 describes the protocols that we have measured and in particular presents the new adaptive protocols. Section 3 discusses our methodology and benchmarks. Section 4 presents measurements of write-run distributions. Sections 5 and 6 present performance measurements. In section 7, we discuss related work. Finally, sections 8 and 9 outline some limitations of this work and conclusions.

2 The Coherence Protocols

2.1 Background

A major reason why programs fail to experience linear speedup on bus-based multiprocessors is bus contention. Thus it is important to reduce the number of bus busy cycles where possible. In this section, we discuss the bus-cycle costs of each coherence scheme, and describe the adaptive protocols that we will measure.

All the results presented in this paper assume that the system implements *read-snarfing*[1] [17, 10]. In such a system, the cache snoop mechanism of every cache, and not only of the requesting cache, checks the address associated with all completed cache-to-cache read operations on the bus. If the address of the data matches in the cache, but the state of the block is invalid, the cache will read the value of the block from the bus, and set the state of the block to read-shared. 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.

In addition, we assume the existence of a *shared line*, a bus wire which is used by the writing cache to tell if other caches currently possess the block being updated. In a normal update protocol, it is asserted by a cache when it receives an update for a block currently in its cache.

We use the following notation in our discussion.

[1] We present results assuming read-snarfing in order to simplify the presentation: the protocols and analysis are simpler to describe in this case since the number of rereads following a write-run is always at most 1. However, we ran the appropriate variants of the experiments described in this paper without read-snarfing as well. The results were not substantially different. For more information, see [1].

- C_i, the cost in bus cycles of an invalidate transaction.
- C_u, the cost in bus cycles of an update transaction.
- C_r, the cost in bus cycles of reading a cache block.
- R, the *invalidation ratio*, defined as $(C_i + C_r)/C_u$.

Note that each initial read of a block, and each read to a block that was previously replaced, entail bus cycle costs that are incurred regardless of the cache coherency protocol used. Therefore, in the discussion below, we focus only on those bus cycle costs which vary from protocol to protocol.

2.2 The Standard Protocols

The Write-Invalidate Protocol (WI)

In write-invalidate, a processor invalidates all other cached copies of shared data and can then update its own copy without further bus operations. The cost in bus cycles incurred by WI depends only on the number of write runs observed and not on their lengths.

Note that because read-snarfing is used, only one read is required to restore the block to all caches in which is was invalidated and not replaced.

The Write-Update Protocol (WU)

In write-update, a processor broadcasts updates to shared data to other caches, so the caches stay coherent. However, shared blocks might not remain shared for the entire duration of a write-run due to replacement. The sharing line ensures that updates only need be broadcast when the data is actively shared.

The Competitive Protocol (SR)

As described in the introduction, the competitive protocol (SR)[2] associates a counter with each cache block, initialized to R, the invalidation ratio. On an update to an actively shared block, the counter in the writer's cache is decremented. When the counter reaches 0, the block is invalidated in all caches except for that of the writing processor, and the counter is reset to R. In addition, if an update is broadcast, and the sharing line is not raised, the counter is reset to R.

One advantage of this protocol is that its worst-case performance is bounded: the number of bus cycles used by this protocol is provably within a factor of 2 of the number used by the optimal offline protocol [13].

2.3 The Adaptive Protocols

We now describe the two dynamic hybrid protocols. Both protocols are adapted from competitive thread-spinning algorithms found in [11]. In that paper, the algorithms decide when to switch between spinning a thread while waiting for a lock — an inexpensive operation that can be repeated many times — and blocking a thread, which is an expensive operation that is done only once. In our algorithms, updating is analogous to spinning, while invalidating is analogous to blocking (since it entails the subsequent large reread cost).

[2] SR stands for "snoopy reading", the name given to the protocol by Eggers and Katz [8]

These protocols are dynamic versions of the competitive protocol SR. They maintain a counter and an *invalidation threshold* T_b for each cache block b. The counter associated to a block is initialized to the value of T_b.

On an update to an actively shared block, the counter in the writer's cache is decremented. When the counter reaches 0, the block is invalidated in all caches except for that of the writing processor. When the block is read again by some processor, a new value for T_b is computed, and each counter for that block is reset to that value. Whereas the SR protocol maintains a constant threshold (equal to the invalidation ratio R), the adaptive protocols adjust the value of T_b for a block after each write run. The difference between the two adaptive protocols is in how they make this adjustment.

The Random Walk Protocol (RW)

In the Random Walk protocol, the value of T_b for each cache block b is 0 when the program starts. It is then adjusted as follows: if the length of the most recent write run to b exceeds the invalidation ratio R, then the invalidation threshold T_b is decremented by 1, as long as it stays nonnegative. On the other hand, if the write run to b has length less than or equal to R, the threshold T_b is increased by 1 as long as it does not exceed the invalidation ratio.

This approach ensures that blocks that consistently have long write runs will sooner or later have their thresholds reduced to 0, so that the block will be invalidated immediately, with no wasted updates. Similarly, for blocks with consistently short write runs, the threshold will be adjusted upwards, so that the block will not be invalidated when it is being actively shared.

For example, suppose that the invalidation ratio is 5, and the current threshold T_b for a particular block is 3. A processor then begins a write run on block b. If the processor only writes the block 4 times before it is accessed by another processor, then T_b will be increased by 1 (to 4). This means that if the subsequent write run also has length 4, T_b will be at an optimal value, and only updates will be issued.

On the other hand, if the block is written 10 times before being accessed by another processor, then T_b will be decreased to 2. Therefore if the subsequent write run is also very long, the protocol will incur a cost of two updates, an invalidate and a reread, rather than three updates, an invalidate and a reread. After two more long write runs, T_b will be reduced to 0, and invalidation will be immediate.

The Random Walk algorithm requires additional memory of 3 to 5 bits per memory block to store the current threshold. Our simulations assume that additional bus lines exist to transmit threshold values to and from processors and memory.

The Last-Three-Samples Protocol (LTS)

Suppose that we are given a probability distribution on write-run lengths, i.e. for each L, we know the probability that the next write-run will be of length L. Then it is possible to determine the optimum threshold for that distribution, i.e. that which minimizes the expected number of bus busy cycles on the next write-run. To do this, we would find the expected cost of using t as a threshold, $EC(t)$, for each possible value of t, and then choose the value which minimizes $(EC(t))$. The last three samples (LTS) protocol approximates this computation, by estimating the probability distribution for the next write-run using the last three write-run lengths observed. Specifically, if the last three write-runs had lengths i, j and k, then LTS chooses the threshold using the hypothesis that the next write-run will have length i with probability 1/3, j with probability 1/3 and k with probability 1/3. Given this approximate distribution, the optimum threshold is computed.

For each choice of i, j, k in this range, the optimum threshold is stored in a static table, so that LTS can lookup the best threshold to use based on the last three observations, and set its invalidation threshold equal to that value.

Note that we limited the values of i, j, and k to $(2*R)$ to keep the size of the table manageable, though in practice limiting the values further to R would not have affected the actual threshold values determined by the algorithm.

When no write history is available for a block, we used 0 as the threshold. When a block had only 1 or 2 write runs, the thresholds were determined by assuming that the distribution is determined by 1 or 2 samples, rather than 3.

The overhead for the LTS algorithm is substantially higher than that of the Random Walk algorithm. Though the LTS doesn't store the current threshold, it does store the lengths of the last three write runs for every memory block. Additional storage is needed for the threshold lookup table. The additional storage needed makes LTS a less practical algorithm than Random Walk.

3 Methodology

Using the Cerberus simulator [4], we simulated the protocols described in the previous section. The WI protocol we used is the Illinois protocol [16], while the WU protocol we used is the Dragon protocol [15]. The Dragon protocol uses a shared line to detect if a block being updated is being actively shared. If the block is not being shared by any other processor, it is marked as exclusive in the writer's cache, and no further updates are done for the length of the write run. WU and the adaptive protocols use the shared line to stop updates when the block is not actively shared.

We chose seven applications to evaluate our protocol. The reference behavior of these applications fall into three categories:

- Applications that have good spatial locality and exhibit high cache hit rates (Gauss, Badgauss, Cholesky).
- An application that exhibits true sharing (MP3D).

- Applications that exhibit both true and false sharing (Topopt, Pverify).
- An application that has few shared accesses (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.

Because the benchmarks we used favored WU, we constructed an additional benchmark that would do poorly with WU. We modified the Gauss benchmark so that all processors access the entire array before starting the row reduction operations. This is done so that the entire array is shared in many processors, forcing an update protocol to do an update transaction for every write. We then ran the modified version of Gauss with a large cache so that the entire array would stay resident during the parallel section of the program. We call this program **Badgauss**.

Cholesky is a program from the SPLASH benchmark suite for sparse matrix factorization [18]. Our simulation used the BCSSTK14 input file (65 million instructions).

MP3D is another program from the SPLASH suite. MP3D is a 3-dimensional particle simulator which has poor locality and incurs a great deal of coherence misses. We ran MP3D on 20000 molecules for 100 steps (497 million instructions).

Topopt performs topological optimization on VLSI circuits using a parallel simulated annealing algorithm [6]. This application exhibits a fair amount of both true and false sharing [7]. We used the cpla1.lomim input file (1.7 billion instructions).

Pverify determines whether two boolean circuits are functionally identical [14]. It exhibits a great deal of false sharing, even when using moderate-sized blocks [7]. 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. Barnes exhibits a very low coherence miss rate, even when 64 processors are used. We simulated 128 bodies (111 million instructions).

Except for the Badgauss benchmark (where we used a very large cache size), we simulated 128K processor caches. We used a block size of 64 bytes and a bus size of 32 bits. Update transactions transferred 4 or 8 bytes of information, depending on the size of the data item being written.

The relatively small bus width makes it more expensive to read an entire cache block than to update the part of the block which has been written. In general, the size of the bus and the system's block size are not important to the success of hybrid schemes; what is important is that the ratio of the cost doing an invalidate (including block rereads) versus the cost of an update is high enough. The ratio is high when block sizes are large relative to the bus size, or when snarfing is not used and the average number of block rereads is at least 2 or 3.

Given these parameters, the cost of an invalidate transaction (C_i) is 2 cycles, the cost of reading a cache block from another cache (C_r) is 19 cycles, and the cost of an update (C_u) is 4 cycles. (Similar values are obtained using, for example, 128 byte blocks and a 64 bit bus.) We assumed a sequentially consistent system and a split-transaction bus when main-memory accesses were performed. Main memory required 10 cycles to produce the first (32-bit) word, and a cycle for every subsequent word in the block.

We were able to accommodate up to 32 processors on a single bus because the bus cycled at the same speed as the processor. In the case where the processor is much faster than the bus, fewer processors could be effectively connected because of increasing bus saturation, which makes reducing bus traffic all the more important.

For the Random Walk algorithm, the initial value of the invalidate threshold was set at 0 (invalidate immediately). Similarly, the threshold used for the LTS when there were no prior samples was 0. Varying the initial value showed no appreciable effect on the results of the adaptive protocols.

4 Write runs

In this section, we present the distribution of write-run lengths measured for four of the applications. We will discuss distributions for both 4 and 32 processors runs; unfortunately, in most cases space constraints allow us to include only the graphs for the 32 processor, 128K cache run. The additional graphs may be found in [1].

Gauss

The most noticeable feature of the Gauss data is the large spike at a write length of 8 (Figure 1). The spike is expected due to the access patterns in the program. As each processor modifies its rows, it writes every element to the right of the pivot column. Since every 64 byte cache block can contain 8 double floating point values, most write runs are 8 long. The working set for Gauss fits in less than 512K; however, even with 32 128K caches, conflict misses still cause blocks containing some matrix elements to be evicted on each iteration. Thus there are not many write runs of length greater than 8.

If not for replacement and a sharing line, a WU protocol would perform worse than WI on Gauss, since the lengths of the vast majority of write-runs exceed the breakeven point (invalidation ratio) of 5. As we shall see, however, during many of the long write-runs, the block did not remain actively shared, enabling WU to perform as well as WI.

Badgauss

The Badgauss benchmark was run with larger caches (512K) and greater associativity (32) than the Gauss benchmark. Since conflict and capacity misses were all but eliminated, the distribution of write run lengths shifted from Gauss's many runs of length 8 to runs of length greater than 100 (Figure 1). In addition, since (by design) the algorithm makes the entire array actively shared, under WU the sharing line will be activated on every write to the array. Thus under WU every write to a block containing array elements will incur a write update transaction. This should cause a

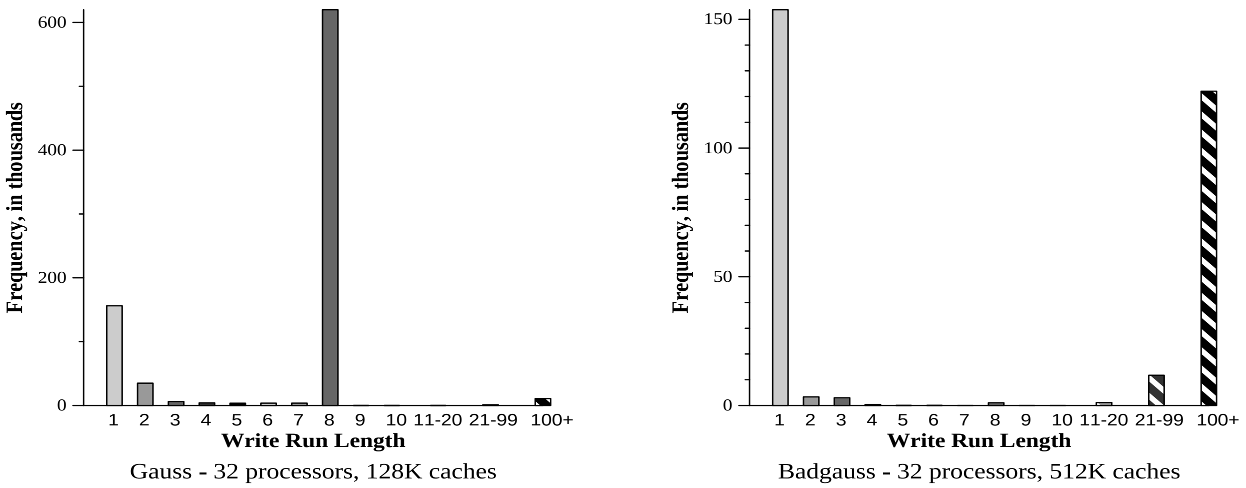

Figure 1: Write run lengths for Gauss and Badgauss

large drop in performance versus WI, especially when large number of processors are used.

Cholesky

Like Gauss, Cholesky also has a spike at write runs of length 8 (Figure 2). However, unlike Gauss, Cholesky also has many write runs of length other than 8. When 4 processors were used with 32K caches, Cholesky had a wide distribution of write runs, with a large number of runs in the 11–20 and 21–99 range. When 128K caches were used, the length of most write runs (except those of length 100 or greater) increased. When 32 processors were used the number of runs of length 1 and 8 dramatically increased, though there were still a non-trivial number of longer write runs. Unlike the 4 processor case, there were few differences between the 32K and 128K cache runs.

Pverify

Because of large amounts of both false and true sharing, the vast majority of write runs in Pverify are very short — 1 or 2 writes before the block is referenced by another processor (Figure 2). Such write patterns heavily favor update-based protocols. Changing the cache size had little effect on write run distributions for either 4 or 32 processors.

Summary

We have presented write-run distributions for a set of benchmarks. As expected, the distributions vary rather significantly from application to application. Interestingly, for the most part the distributions tend to be bimodal, with many very short write runs and some very long write runs A more detailed analysis of the write-run data shows that the variance in the write-run lengths on a per block basis is significantly smaller than that of the overall application. This is to be expected, since the reads and writes to a shared cache block represent accesses to shared data structures, and these tend to be fairly consistent over time, even though the access patterns to different parts of the data structure may be quite varied.

Nonetheless, all together, these measurements suggest that there is potential for adaptive protocols to perform as well as the better of WI and WU. In particular, for those cache lines where the write-run lengths are uniformly short or uniformly long, the adaptive protocols may be able to converge to near optimal performance.

5 Performance

We now turn to the measurements of the performance of the various protocols. We begin with a summary of the simulated execution time of the five on each of the benchmarks using 128K caches and 32 processors. All speedup graphs in this paper are relative to the execution time when running the same program on a single processor using the same architectural parameters.

Figure 3 shows the speedups the various protocols achieved. Both adaptive protocols performed well compared to WI and WU. On MP3D, LTS ran 27% faster than WI, and 16% faster than WU. On Topopt, LTS was 14% faster than WI, and just as fast as WU. On Barnes, Gauss, Badgauss and Cholesky, LTS was between 1 and 5% faster than the better of WU and WI. On Pverify, LTS was 70% faster than WI, but slower (3%) than WU.

The performance of Random Walk was generally even better. On MP3D, Random Walk ran 24% faster than WI, and 13% faster than WU. On Topopt, Random Walk was 27% faster than WI, and 11% faster than WU. On Barnes, Gauss, Badgauss and Cholesky, Random Walk was between 1 and 5% faster than the better of WU and WI. On Pverify, Random Walk was 70% faster than WI, but slightly (2%) slower than WU.

The nonadaptive hybrid protocol SR (with invalidation threshold of 3) did not do nearly as well as the adaptive protocols. For those programs where WI works better than WU (Barnes, Gauss, Badgauss, and Cholesky), it generally did about as well as WI. For those applications that did significantly better using

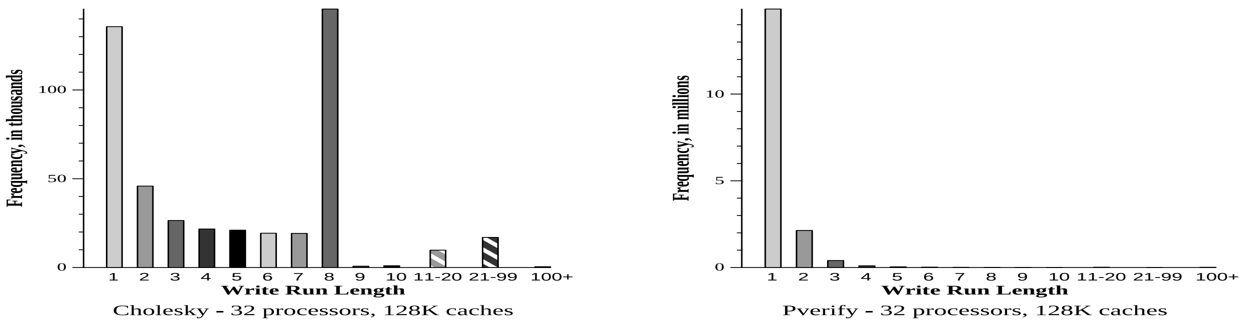

Figure 2: Write run lengths for Cholesky and Pverify

WU (MP3D, Pverify, and Topopt), SR did relatively poorly. On MP3D, it did as well as WI, while the dynamic protocols performed 25% better. It did not come near the performance of the adaptive protocols (or WU) on Pverify, and actually performed worse than all protocols on Topopt.

From the point of view of other metrics, the adaptive protocols were also very successful. As expected, WU typically had a substantially lower read miss rate than WI. The adaptive protocols had read miss rates much closer to those of WU. On the other hand, WU typically required many more bus arbitrations than WI. The adaptive protocols issued a number of bus transactions much closer to that of WI, and sometimes smaller than both. Finally, the number of bus bytes transferred by the adaptive protocols was closer to the smaller of WU and WI. (Frequently, there was a substantial difference between the two, and neither was uniformly superior.)

In the next section, we present a detailed description of the results of some of the benchmarks [3].

6 Detailed Measurements

Badgauss

We begin with the results for our artificial benchmark Badgauss on which WU performs poorly. We used 512K caches so that the entire array would remain cached in all processors, forcing WU to do an update transaction on every write. As expected, WU did poorly on Badgauss, though the effect of the many update transactions did not affect performance much until 32 processors were used (Figure 4). The reason for this is that a relatively large amount of computation is done for every store in the program. Thus it is only when the cost of an update became high (due to contention) that the additional bus load caused by the large number of updates became a factor. WU used 40–60 times more bus transactions than invalidate, and transferred 8–13 times the data (Figure 5).

Because the write runs to the blocks that make up the array were much longer than the invalidation threshold, the adaptive strategies always invalidated on these blocks, thus avoiding the poor behavior of WU. On the other hand, by converging to WU behavior for blocks with short write runs, they succeeded in transferring less than 40 % of the bus bytes transferred by WI.

Cholesky

Because of the number of long write runs, the WI protocol performed better than WU on Cholesky (Figure 6). The two adaptive protocols performed about the same as WI, varying in performance no more than 1% for all configurations. WU performed the worst of the 4 protocols. For both cache sizes, WU performed 5% worse using 16 processors, and about 10% worse using 32 processors. This performance decrease occurred despite the reductions in both read and write miss rates under WU. WU reduced read misses for the 32K configuration by about 12% for 4 and 16 processors, and 50% for 32 processors. The reductions were even greater when 128K caches were used. The adaptive protocols also reduced the read and write miss rates over WI — though in most cases not as much as WU. The exceptions were for 16 processors and 32K caches, where both adaptive protocols did better than WU, and 32 processors and 32K caches, where RW did better than WU.

The reason for the performance decline under WU can be seen in the graphs of the number of bus transactions and the number of bytes transferred (Figure 7). The WU protocol consistently used many more bus transactions than the WI protocol. When 32K caches were used, the increase ranged from 1.7 times as many for 4 processors to 2.9 times for 32 processors. The increase was even worse for 128K caches, where the increase ranged from 3.2 to 3.6 times that of WI. The number of transactions for WU did not increase by very much when going from 32K to 128K caches, but the number of transactions for WI decreased when the cache size was increased, leading to the greater disparity in the number of bus transactions. The increase in the number of bytes transferred was not as severe — it ranged from 11% to 33% for 32K caches and from 34% to 45% for 128K. The additional load placed on the bus by the many update transactions lead to the decreased performance of WU when compared to WI. The performance impact of using WU versus WI would have been even greater if the overall bus utilization of the application had been greater.

Using the adaptive protocols also increased bus

[3] Due to the poor performance of SR, and lack of space, we omit the detailed measurements of its performance from the next section.

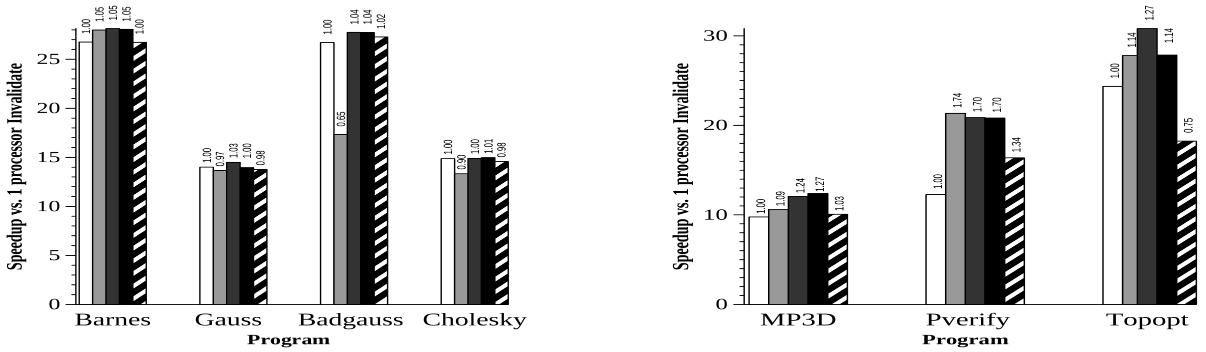

Figure 3: Results summary: 128K Caches, 32 Processors

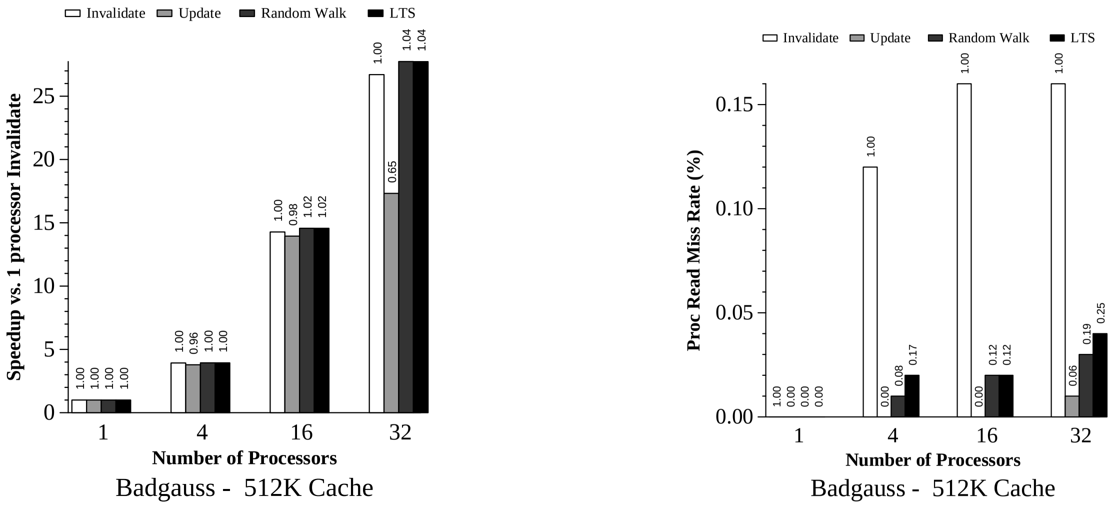

Figure 4: Speedups(L) and read misses(R) for Badgauss

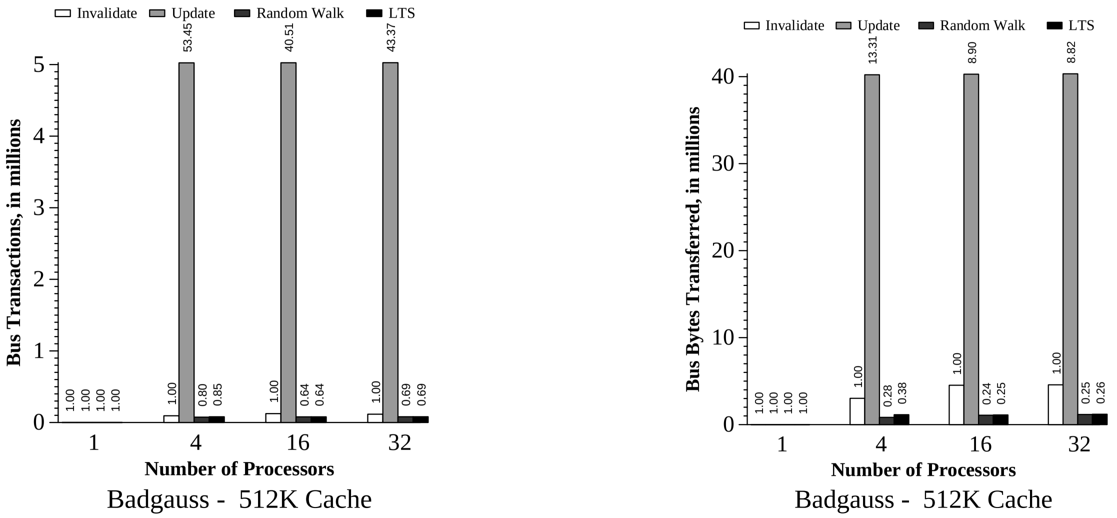

Figure 5: Bus transactions(L) and bytes transferred(R) for Badgauss

transactions for the adaptive protocols, but not to the degree that WU did. The increase in bus transactions ranged from 6–24% for 32K caches, and from 14% to 33% for 128K caches. The increase of RW was in general slightly higher than that of LTS. Except for the 4 processor, 128K cache case, the adaptive protocols reduced the number of bytes transferred below even that of WI. The decrease ranged from 1%–3% for 4 and 16 processors and 32K caches to about 10% for the other configurations.

Overall, the adaptive protocols did fairly well on Cholesky. They performed about as well as the better of the two standard protocols (WI), while decreasing the miss rate without a huge increase in the number of bus transactions, and in most cases transferring less data than the WI protocol.

Pverify

As discussed in section 4, Pverify has a great deal of true and false sharing, which leads to a large number of very short write runs. Therefore, we would expect that an update strategy would work better than invalidate.

When 32K caches were used, much of the traffic on the bus was due to reads and writebacks due to capacity and conflict misses. Because of this, using WU or an adaptive scheme over WI did not dramatically increase the poor speedup of the application. For 16 processors, using RW increased performance by 19%, with WU and RW close behind at 18% and 15% respectively. For 32 processors, WU was best with a 19% improvement, while RW and LTS both performed 15% better. The improvements were entirely due to the reduction in bus traffic, since the bus is completely saturated when 16 or 32 processors are used. The three non-invalidate schemes reduced the read miss rate by about 20% for all configurations. The write miss rate was cut by all three by more than 80%. The reduction in misses helped to reduce the number of bus transactions by 7–10%, and the number of bytes transferred by 12–19%.

The differences between using WI and WU or an adaptive strategy were much larger when 128K caches are used (Figure 8). WU, RW, and LTS increased performance by 20% when 16 processors were used. For 32 processors, using WU increased performance by 74%, with RW and LTS behind at 70%. Read misses were cut by about 60% for 4 and 16 processors using WU and RW, and slightly less for LTS. All three protocols cut misses by about 50% when 32 processors were used. Write misses were also reduced by about 85% for LTS, and by 95% or more for RW and WU. Because of the reduced number of misses, bus transactions were cut by about 20% for all configurations (Figure 9). The reduction in number of bus transactions, plus the fact that updates only transfer 8 bytes, while re-reads due to invalidates transfer 64 bytes (most of which are usually not used in Pverify) resulted in a drop in the number of bytes transferred by 40–50% for all three non-invalidate protocols over WI.

These results are not surprising, given the poor behavior of Pverify when using large block sizes [2]. While false sharing hurts performance under both WI and WU, it it much worse under WI, since the entire cache block is "ping-ponging" between the false sharing caches, rather than just the values actually being written.

7 Related Work

There have been various studies of hybrid cache coherence protocols. Eggers and Katz compared the performance of the hybrid protocol of [12], which they called "snoopy reading", with WU and WI [9]. Using a value of 3 for the invalidation threshold, they found that neither WU nor WI was uniformly superior and that snoopy reading consistently had performance which was intermediate between the two.

Archibald introduced a protocol which improves over hybrid protocols in which the counter is associated with the reader (as opposed to the writer). In his protocol, a block is not automatically invalidated when its count reaches 0 [3]. Instead, a cache simply doesn't raise the shared line if the relevant block's count is 0. If some cache whose block count is not 0 indicates that it is keeping the block by raising the shared line, then no cache invalidates its block. The block is invalidated in the sharing caches only when all counts for the block are 0. This scheme also has the advantage that a succession of single writes from different processors will not cause other copies to be deleted. It does this without any additional hardware required.

Veenstra and Fowler evaluated the cache coherence protocol used by the DEC Alpha AXP [19], which uses the snoopy reading scheme with an invalidate threshold of 2 [20]. They reported that using a threshold of 2 resulted in shared blocks being invalidated too quickly, thus applications did not gain from selective updating. Increasing the invalidate threshold in order to get more updating behavior does have drawbacks, however. A processor executing a moderate or long write run will have to pay for several unnecessary updates before it can invalidate. This is a problem with using a static value for the invalidate threshold.

More recently, Dahlgren compared Archibald's protocol and a slight variant of the snoopy-reading protocol [5]. He found that Archibald's protocol consistently outperformed snoopy-reading. He also found that by using read snarfing, write caching and (necessarily) a release consistency model, along with Archibald's protocol, substantially increased program performance of Archibald's protocol over a base architecture executing an invalidate protocol.

The primary difference between Dahlgren's work, and the work in this paper, is that we assume that a stricter memory consistency model (*sequential consistency*) must be provided to the application. This rules out using a coalescing write buffer. Applications running on machines without write buffers have not only longer write runs on average, but also a wider variance in the distribution of write run lengths. It is this variance which makes it difficult for a hybrid coherence protocol that uses a fixed threshold to perform well.

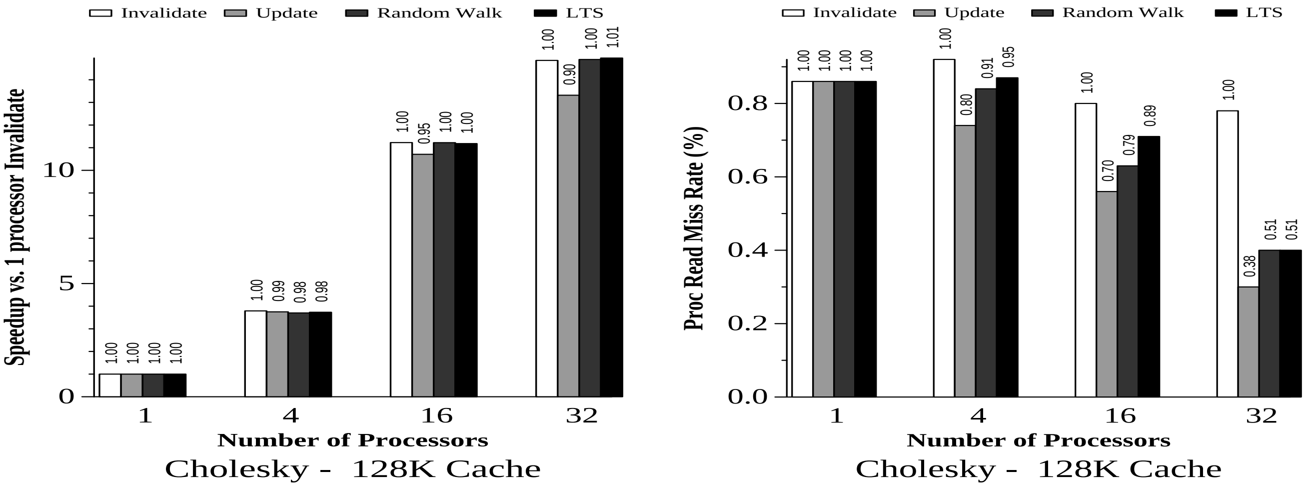

Figure 6: Speedups (L) and read misses (R) for Cholesky

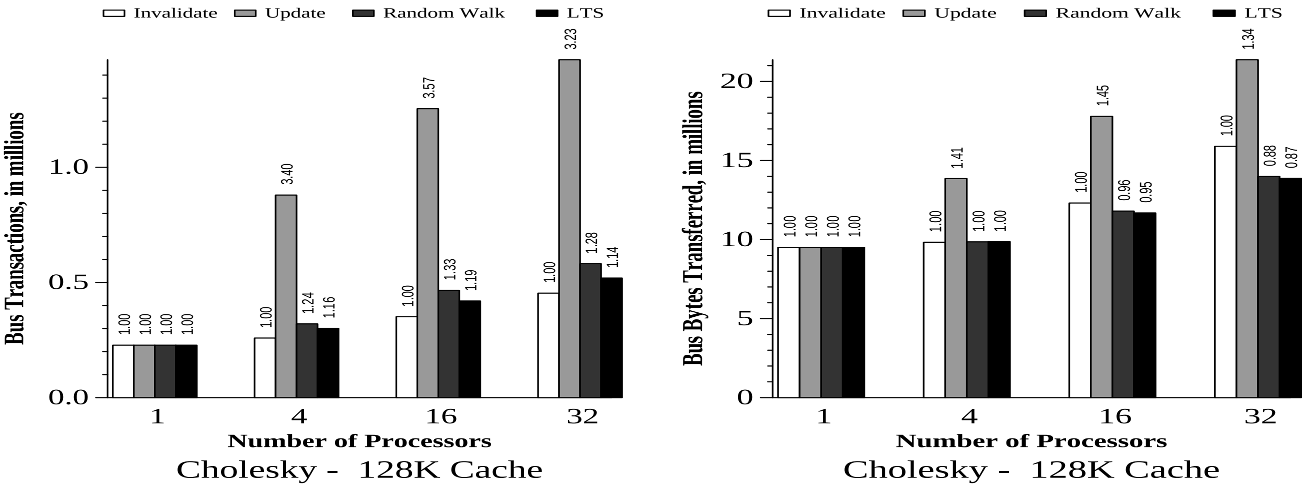

Figure 7: Bus transactions (L) and bytes transferred (R) for Cholesky

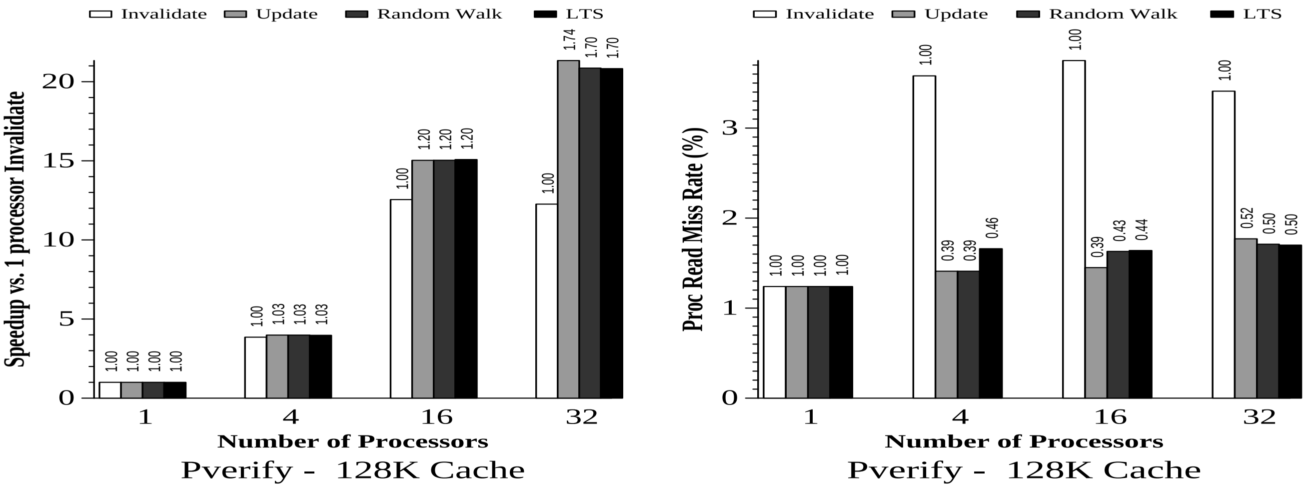

Figure 8: Speedups and read misses for Pverify

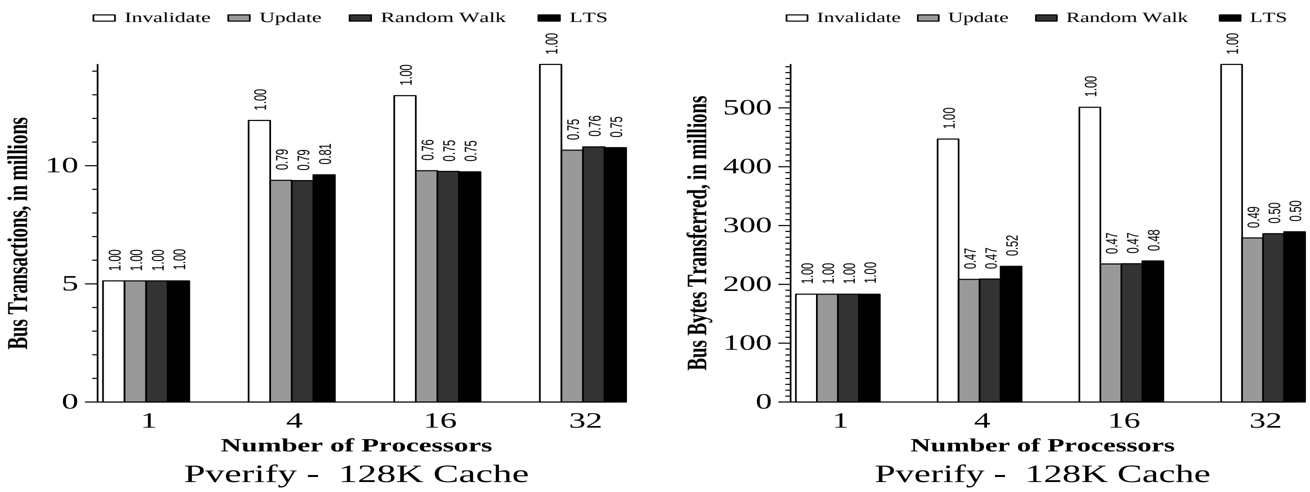

Figure 9: Bus transactions and bytes transferred for Pverify

8 Limitations

This study demonstrates the performance advantages of adaptive coherence protocols via simulation of a collection of parallel benchmarks. It remains unclear whether the cost of the adaptive schemes is justified by the increase in performance gained by using them. Given that LTS has a higher hardware cost than RW, and in most cases did not perform better than RW, it is hard to justify using LTS over RW. In our test programs, RW significantly outperformed the better of WU or WI only on MP3D and Topopt. In most of the other cases, RW did as well or slightly better than the best choice of WI or WU for a particular application. In the final analysis, one needs to weigh the benefit of using an adaptive protocol and obtaining performance equal to the better of WI and WU versus the additional complexity and hardware needed to implement the adaptive protocols. In addition, there is no guarantee that optimizing a coherence protocol to reduce bus busy cycles will lead to better overall system performance.

On the surface, it may be surprising that WU performed as well as it did. One reason is that the performance of WU degrades less than WI in the presense of false sharing, since only those portions of the block which are written are transmitted over the bus, rather than the entire block which is the case with WI. In addition, our simulation assumptions favored WU over WI, since we assumed both a large block size and a fast bus relative to the speed of the processors. Since WU usually required more bus transactions than WI, increasing the cost of initiating a bus transaction would affect WU more than WI. In addition, in many of the benchmarks the application's working set did not fit in cache, which caused blocks that were not being actively used in a given cache to be replaced. This tended to increase the number of blocks that are found in only one cache, i.e. blocks that may be written without an update transaction. In systems with large (second-level) caches, replacement activity may be minimal or non-existent. In this case, a block which is written exclusively by a single processor, but was referenced at one time by another processor (perhaps in the algorithm's initialization phase) would still have updates sent for every write. Note that the Badgauss benchmark was constructed to exhibit this kind of behavior. Applications that exhibit this sort of behavior perform poorly under WU, but the adaptive protocols handle the situation well by recognizing the long write run(s) and invalidating the block immediately.

9 Conclusions

This paper evaluates adaptive, hybrid cache coherency protocols for shared-memory multiprocessors. The write-run distributions that we have collected indicate that there is large variability between write-run lengths for different cache blocks and between different programs. This suggests that no fixed non-adaptive algorithm can perform uniformly well, even on the small sample of applications we studied.

Our results indicate that easily implementable adaptive protocols perform well compared to WI and WU. Random Walk in particular did uniformly as well as the better of WI and WU, and sometimes much better than that. Moreover, it is a simple protocol with low time and space overhead. The adaptive protocols did very well with respect to other performance metrics as well. They achieved low miss rates, low numbers of bus transactions and low numbers of bus bytes transferred; in all cases, closer to the better of WI and WU. The performance advantages of the adaptive protocols increased with the number of processors and the size of the caches simulated.

References

[1] C. Anderson. *Improving the Performance of Bus-Based Multiprocessors.* PhD thesis, University of Washington, 1995.

[2] C. Anderson and J.-L. Baer. Two techniques for improving the performance of bus-based multiprocessors. In *IEEE Int. Symp. on High-*

Performance Computer Architecture, pages 264–275, 1995.

[3] J. Archibald. A cache coherence approach for large multiprocessor systems. In *International Conference on Supercomputing*, pages 337–345, 1988.

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

[5] F. Dahlgren. Boosting the performance of hybrid snooping cache protocols. In *Proc. 22nd Int. Symp. on Computer Architecture*, pages 60–69, June 1995.

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

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

[8] S. Eggers and R. 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.

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

[10] J. Goodman and P. Woest. The Wisconsin Multicube: A new large-scale cache coherent multiprocessor. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 422–431, 1988.

[11] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 41–55, 1991.

[12] A. Karlin, M. Manasse, L. Rudolf, and D. Sleator. Competitive snoopy caching. In *Proc. of 27th Symposium on Foundations of Computer Science*, pages 244–254, 1986.

[13] A. Karlin, M. Manasse, L. Rudolf, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[14] 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.

[15] E. McCreight. The Dragon computer system: An early overview. Technical report, Xerox Corp., 1984.

[16] M. Papamarcos and J. 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.

[17] 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.

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

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

[20] J. Veenstra and R. Fowler. The prospects for on-line hybrid coherency protocols on bus-based multiprocessors. Technical Report 490, University of Rochester, 1994.