

EVALUATING THE PERFORMANCE OF FOUR SNOOPING CACHE COHERENCY PROTOCOLS

Susan J. Eggers

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

Randy H. Katz

Computer Science Division
Dept. of Electrical Engineering & Computer Science
University of California
Berkeley, California 94720

Abstract

Write-invalidate and write-broadcast coherency protocols have been criticized for being unable to achieve good bus performance across all cache configurations. In particular, write-invalidate performance can suffer as block size increases; and large cache sizes will hurt write-broadcast. Read-broadcast and competitive snooping extensions to the protocols have been proposed to solve each problem.

Our results indicate that the benefits of the extensions are limited. Read-broadcast reduces the number of invalidation misses, but at a high cost in processor lockout from the cache. The net effect can be an increase in total execution cycles. Competitive snooping benefits only those programs with high per-processor locality of reference to shared data. For programs characterized by inter-processor contention for shared addresses, competitive snooping can degrade performance by causing a slight increase in bus utilization and total execution time.

1. Introduction

Snooping cache coherency protocols [Arch86] are a good match for bus-based, shared memory multiprocessors, because they take advantage of the broadcast capabilities of the single interconnect. Within the snooping coherency category, two techniques, *write-invalidate* and *write-broadcast*, have been developed. In write-invalidate a processor invalidates all other cached copies of shared data and can then update its own without further bus operations. Under write-broadcast, a processor broadcasts updates to shared data to other caches, so that all copies are the same.

Both techniques have been criticized for being unable to achieve good bus performance across all cache

configurations. In particular, write-invalidate performance can suffer as block size increases because of inter-processor contention for addresses within the cache block; and large cache sizes will hurt write-broadcast, because of continued bus updates to data that remains in the cache but is no longer actively shared.

Enhancements to the original protocols have been proposed to solve each problem. A read-broadcast extension [Good88, Karl88, Sega84] to write-invalidate reduces the number of misses for invalidated data by allowing all caches with invalidated blocks to receive new data when any of them issues a read request. It should therefore improve both the miss ratio and bus utilization of write-invalidate. A competitive snooping protocol, introduced in [Karl86, Karl88], was designed to limit the number of broadcasts in write-broadcast. It therefore puts a cap on the performance loss caused by large caches.

The goal of this paper is twofold: first, to measure the performance problems in the write-invalidate and write-broadcast protocols, as block or cache size increases; and second, to gauge the extent to which the read-broadcast and competitive snooping extensions solve each problem. All studies were done via trace-driven simulation of parallel applications. Our results indicate that read-broadcast reduces the number of invalidation misses, but at a high cost in processor lockout from the cache. The net effect can be an *increase* in total execution cycles. Competitive snooping benefits only those programs with high per-processor locality of reference (sequential sharing) to shared data. For programs characterized by inter-processor contention (fine-grain sharing) for shared addresses, competitive snooping can degrade performance by causing a slight increase in bus utilization and total execution time.

We have used trace-driven simulation of parallel programs in two other studies. In [Egge88] trace-driven analysis verified a model of coherency overhead in write-invalidate and write-broadcast protocols. [Egge89b] studied the effects of increasing block and cache size on the cache and bus behavior of parallel programs running under write-invalidate protocols. A

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

summary of the block size results from that paper is the basis for the evaluation of write-invalidate protocols in this work.

The remainder of this paper begins with a brief description of the methodology. The two companion protocol studies follow. Each begins with a description of the original protocol and empirical evidence of the performance loss caused by increasing block or cache size. Then the protocol extensions are described, and the extent to which they improve performance is measured. Section 3 reviews write-invalidate protocols and the effects of increasing block size on miss ratio and bus utilization studied in [Egge89b]. Section 4 presents the read-broadcast extension and its benefits and costs to both performance and cache controller implementation. Write-broadcast and the effects of increasing cache size on bus traffic is covered in section 5. Section 6 discusses the competitive snooping alternative and its performance relative to write-broadcast. The last section integrates the results of both studies.

2. Methodology and Workload

We used trace-driven simulation in our analysis. Our simulator emulates a simple shared memory architecture, in which a modest number of processors (five to twelve) are connected on a single bus. The CPU architecture is RISC-like [Pat85], assuming one-cycle per instruction execution. Not all instructions follow this model; therefore the bus utilization results will be slightly overestimated and throughput underestimated, because the simulation processors return to use the bus more quickly than in a real machine. However, the studies in this paper rely only on the *relative* values of these metrics between coherency mechanisms; absolute figures are not germane to the analysis and conclusions. All other metrics used in the studies, for example, cache miss rates and numbers of bus operations, should be unaffected.

With the exception of those cache parameters that are varied in the studies (cache size, block size and coherency protocol), the memory system architecture is roughly that of the SPUR multiprocessor [Hill86]. The simulator's board-level cache is direct mapped, with one-cycle reads and two-cycle writes. Its cache controller implements a test-and-test-and-set sequence for securing locks [Wood87], processor lockout from the cache during snoop activity, and many of the timing constraints of the actual SPUR implementation. Bus activity is implemented using a modified NuBus arbitration protocol [Gibs88], and bus contention is accurately modeled.

The inputs to the simulator are traces gathered from four parallel CAD programs, developed for single-bus, shared memory multiprocessors (Table 2-1). The choice of application area was deliberate, so that the workload being analyzed was appropriate for a small-scale machine. One program is production quality (SPICE); the others are research prototypes. Two of the programs are based on simulated annealing algorithms. CELL [Caso86] uses a modified simulated annealing algorithm for IC design cell placement, and placed twenty-three cells in our trace. TOPOPT [Deva87] does topological compaction of MOS circuits, using dynamic windowing and partitioning techniques. Its input was a technology independent multi-level logic circuit. VERIFY [Ma87] is a combinational logic verification program, which compares two different circuit implementations to determine whether they are functionally (Boolean) equivalent. The circuits used for the trace were combinational benchmarks for evaluating test generation algorithms. The final program, SPICE [McGr86], is a circuit simulator; it is a parallel version of the original direct method approach, and its input was a chain of 64 inverters.

All applications use a coarse-grain programming paradigm for carrying out parallel activities. The granularity of parallelism is a process, in this case one for each processor in the simulation. The model of execution is single-program-multiple-data, with each process independently executing identical code on a different portion of

Parallel Applications			
Trace Name	Architecture, Operating System	Program Description	Number of Processors
CELL	Sequent Balance, Unix	simulated annealing algorithm for cell placement	12
TOPOPT	Sequent Balance, Unix	simulated annealing algorithm for topological optimization	11
VERIFY	Sequent Balance, Unix	logic verification	12
SPICE	ELXSI 6400, Embos	direct method circuit simulator	5

Table 2-1: Traces Used in the Simulations. The traces used in the sharing simulations were gathered from parallel programs that were written for shared memory multiprocessors. The programs are all "real", being either production quality (SPICE) or research prototypes.

shared data. The shared data is divided into units that are placed on a logical queue in shared memory. Each process takes a unit of work from the queue, computes on it, writes results, and then returns the unit of work to the end of the queue.

The traces were generated on a per-processor basis. The number of processors in the simulations is identical to the number of processors used in trace generation. For SPICE this number is 5, and for the Sequent traces either 11 or 12. Each per-processor trace is a separate input stream to the simulator. Synchronization among the streams depends on the use of locks and barriers in the programs, and is handled directly by the simulator. Statistics are generated from approximately 300,000 references per processor, after cache steady state has been reached. (See [Egge89a] for a more detailed discussion of the methodology.)

3. The Write-Invalidate Protocols

3.1. Protocol Description

Write-invalidate protocols maintain coherency by requiring a writing processor to invalidate all other cached copies of the data before updating its own. It can then perform the current update, and any subsequent updates (provided there are no intervening accesses by other processors) without either violating coherency or further utilizing the bus. The invalidation is carried out via an invalidating bus operation. Caches of other processors monitor the bus through the snoop portion of their cache controllers. When they detect an address match, they invalidate the entire cache block containing the address.

Berkeley Ownership [Katz85] is a write-invalidate protocol that has been implemented in the SPUR multiprocessor [Hill86]. It is based on the concept of cache block ownership. A cache obtains exclusive ownership of a block via two invalidating bus transactions. One is used on cache write misses and obtains the block for the requesting processor, at the same time it invalidates copies in other caches. The second is an invalidation signal that is used on cache write hits. Once ownership has been obtained, a cache can update a block locally without initiating additional bus transfers. A block owner also updates main memory on block replacement and provides data to other caches upon request. Cache-to-cache transfers are done in one bus transfer, with no memory update.

Write-invalidate protocols have two sources of bus-related coherency overhead. The first is the *invalidation signal* mentioned above. The second is the cache misses that occur when processors need to rereference invalidated data. These misses, called *invalidation misses*, would not have occurred had there been no sharing. They are present because the shared data had previously been written, and therefore invalidated, by another processor. They are additional to the customary,

uniprocessor misses (for example, first-reference misses and those necessitated by block replacements).

3.2. The Write-Invalidate Trouble Spot

Because they create a data writer that can access a shared block without using the bus, write-invalidate protocols minimize the overhead of maintaining cache coherency in two cases: when there are multiple consecutive writes to a block by a single processor (sequential sharing), and when there is little inter-processor contention (fine-grain sharing) for the shared data. Periods of severe contention, however, cause coherency overhead to rise. Inter-processor contention for an address produces more invalidations; the invalidations interrupt all processors' use of the data and increase the number of invalidation misses to get it back. The result is that shared data pingpongs among the caches, with each processor's references causing additional coherency-related bus operations. The greater the number of processors contending for an address, the more frequent the pingponging.

The problem is exacerbated by a large block size, because contention can occur for *any* of the addresses in the block. Therefore the probability that the block will be actively shared increases. An invalidation to one word in a block causes all other words to be invalidated. When other processors subsequently reread these addresses, additional read misses are incurred. The overhead is paid even when a processor reads an address that was not updated. With small block sizes, particularly those of only one word, a write to one address has less effect on reads to another.

3.3. Empirical Evidence for the Trouble Spot Analysis

In [Egge89b] we studied the effect on both miss ratio and bus utilization of increasing block size and cache size under write-invalidate protocols. (The write-invalidate protocol used in the simulations was also Berkeley Ownership.) The results quantify the loss in performance due to invalidations and invalidation misses. In particular, they support the above analysis concerning the adverse effects of fine-grain sharing, as block size increases.

Parallel programs with both types of sharing behavior suffer from coherency overhead. Unlike uniprocessor misses [Agar88, Alex86, Good87, Hill87, Smit87], invalidation misses react less favorably to increasing block size. [Egge89b] found that the proportion of invalidation misses to total misses increased with larger block sizes, and for three of the traces was significant. (The proportions grew from .32 to .37 for CELL, .14 to .30 for SPICE, .06 to .51 for VERIFY and .39 to .94 for TOPOPT, as block size was increased from 4 to 32 bytes.) For programs with sequential sharing (CELL and SPICE), (total) miss ratios were higher than for comparable uniprocessor programs and declined with increasing block size at a slower rate.

The effect on programs with fine-grain sharing (TOPOPT and VERIFY) was more severe. Here invalidation misses increased with increasing block size, not only in proportion to total misses, but in absolute numbers as well. (The proportion of invalidation misses for TOPOPT and VERIFY is stated above; the percentage increase in number of misses was 511 and 840 percent, respectively.) Their dominance was so complete that they caused miss ratios to rise with block size, rather than fall, as is normally the case with uniprocessor programs in caches of this size (128K bytes).

The additional cache misses increased bus utilization. Moreover, sharing under write-invalidate protocols introduces another type of bus operation, the invalidation signal, which further increased bus utilization. Bus utilization rose 407 and 94 percent for TOPOPT and VERIFY, as block size increased from 4 to 32 bytes. Even for the small-scale multiprocessors studied (12 processors at most), the bus was well utilized, with bus utilization figures of 45 and 97 percent, respectively, at the 32 byte block size. Bus utilization for CELL and SPICE was midrange, higher than for uniprocessor programs, and declined over the block size spectrum.

4. The Read-Broadcast Extension

4.1. Protocol Description

Since invalidation misses play such a large role in the cache and bus performance of parallel programs at large block sizes, coherency protocols that can reduce them are desirable. *Read-broadcast* [Good88, Sega84] is an enhancement to write-invalidate protocols designed explicitly for this purpose. Under read-broadcast snoops update an invalidated block with data from the bus, whenever they detect a read bus operation for the block's address. Detection is positive whenever the tag of the snooped address matches that of a cached block, and the block state is invalid. The read-broadcast extension adds little complexity to the cache controller hardware. An examination of the SPUR cache controller implementation indicates that one additional minterm is required in the snoop PLA for the detection. Assuming that the snoop can have access to the cache in a short and bounded amount of time, a buffer large enough to hold the data as it comes from the bus is also needed. If timely snoop access to the cache cannot be guaranteed, an extra bus line is necessary to delay transmission of the data. Finally, control to implement read-interference¹ is

¹ Read-interference occurs when a processor has queued a bus read request for an address that is read-broadcast before the requesting processor obtains the bus. During the read-broadcast the requesting processor updates its cache with data from the bus. Therefore it can satisfy its read reference directly from the cache and no longer requires the bus operation. Control is needed to detect this interference and cancel the requesting processor's pending read bus operation.

required to meet the invalidation miss limit, described below.

The technique improves the performance of write-invalidate by limiting the number of invalidation misses to one per invalidation signal. One invalidation miss occurs if the bus operation is a read issued by a cache with a previously invalidated block. No invalidation misses result when the bus read is a first-reference or replacement miss. Subsequent rereads by processors that have received data on a read-broadcast will be a cache hits rather than invalidation misses.

4.2. Read-Broadcast Results

4.2.1. The Benefits to Miss Ratio and Bus Utilization

Read-broadcast reduced the number of invalidation misses (see Table 4-1). For three of the traces (CELL, TOPOPT and VERIFY) the drop ranged from 13 to 51 percent, over all block sizes. The decrease for SPICE was much lower. SPICE data structures had been explicitly sized to the ELXSI 6400 64-byte cache block to avoid inter-processor contention for addresses within a block. Therefore, for block sizes considered in this study, up to 32 bytes, little contention was observed; and read-broadcast consequently brought less benefit.

Because of the decrease in invalidation misses, the proportion of invalidation misses within total misses was less than for write-invalidate (Figure 4-1). This is important, because increases in block and cache size produce steeper reductions in uniprocessor misses than invalidation misses. Therefore, to the extent that misses in parallel programs are caused by normal cache accesses rather than sharing activity, cache performance will improve as block and cache sizes increase. At larger block sizes invalidation misses for CELL, TOPOPT and VERIFY dropped to between a quarter and a third of the total. (Under Berkeley Ownership they had ranged from thirty to over forty percent.) But for TOPOPT invalidation misses still dominated miss ratio behavior at most block sizes (90 percent at 32 bytes at maximum). As with the original write-invalidate protocol, the ratio of invalidation to total misses for all traces rose with increasing block size.

For the most part the consequence of the drop in invalidation misses was a decline in the total miss ratio (see Table 4-1). CELL and TOPOPT had moderate decreases (13.7 to 15.6 percent and 17.2 to 33.8 percent, respectively); VERIFY had a wider range of decrease (1.0 to 19.3 percent). The miss ratio for SPICE did not decline across all block sizes, and, when it did, the decrease was small. The small increases occurred because the samples in comparative (Berkeley Ownership vs. read-broadcast) simulations covered a slightly different set of references. The difference in samples was caused by the elimination of invalidation misses from the read-broadcast simulations. Changing invalidation misses to cache hits allows processors to process references more

Comparison of Berkeley Ownership & Read-Broadcast							
Trace	Block Size (bytes)	Invalidation Misses			Miss Ratio		
		Berkeley Ownership	Read Broadcast	Decrease (percent)	Berkeley Ownership	Read Broadcast	Decrease (percent)
CELL	4	22649	13566	40.1	1.93	1.67	13.7
CELL	8	18823	11264	40.2	1.49	1.28	14.1
CELL	16	15040	8942	40.5	1.10	0.93	15.6
CELL	32	11748	7325	37.6	0.86	0.73	14.4
SPICE	4	6918	6663	3.7	2.90	2.97	-2.2
SPICE	8	4143	3870	6.6	1.64	1.65	-0.2
SPICE	16	3607	3447	4.4	1.09	1.10	-0.4
SPICE	32	3726	3009	19.2	0.77	0.74	3.4
TOPOPT	4	1890	922	51.2	0.15	0.12	20.1
TOPOPT	8	6117	4706	23.1	0.25	0.20	17.2
TOPOPT	16	8835	6459	26.9	0.30	0.23	23.2
TOPOPT	32	11556	7385	36.1	0.37	0.25	33.8
VERIFY	4	2441	2062	15.5	1.42	1.41	1.0
VERIFY	8	8921	7786	12.7	1.38	1.34	2.6
VERIFY	16	15371	11497	25.2	1.40	1.28	9.1
VERIFY	32	22957	13717	40.2	1.45	1.17	19.4

Table 4-1: Comparison of Invalidation Misses and Miss Ratios for Berkeley Ownership and Read-Broadcast. This table depicts the decline in the number of invalidation misses and the miss ratio that occurred with read-broadcast. The drop in invalidation misses was less pronounced for SPICE, because its shared data had been optimized for a block size larger than the maximum studied here. This small decline, coupled with a slight rise in uniprocessor misses, produced rising miss ratios (negative decreases) for some block sizes. (All simulations were run with a 128K byte cache; miss ratios are the geometric mean across all processors.)

quickly than under Berkeley Ownership. The effect is to slightly alter the set of references executed and the global order in which they are processed under the two protocols. For SPICE the consequence was a slight rise in the uniprocessor component of the miss ratio for read-broadcast (relative to Berkeley Ownership), which offset the small decline in the number of invalidation misses. For the other traces the sample discrepancy was considerably less, the uniprocessor misses were almost identical, and the reduction in the number of invalidation misses was also greater. Therefore the drop in invalidation misses produced a corresponding decline in the miss ratio.

The critical system bottleneck in a single-bus, shared memory multiprocessor is the bandwidth of the system bus. Therefore the most important consequence of read-broadcast is the effect of its lower miss ratios on bus utilization. The improvement (i.e., drop in bus utilization) ranged from 8.7 to 10.9 percent for CELL, .8 to 5.1 percent for SPICE, 14.3 to 22.6 percent for TOPOPT and .8 to 11.5 percent for VERIFY. (Details appear in Table 4-2.) To put the read-broadcast benefit in perspective, the change was large enough to allow an additional two processors for TOPOPT, and one each for CELL and VERIFY, and still maintain the same level of bus utilization. (SPICE had lower bus utilization for the block sizes that had a slight rise in the miss ratio, because the total

cycles in the simulation were higher with read-broadcast. The cycle increase was due to a greater delay in obtaining the bus and several other read-broadcast-related factors that are discussed below.)

The magnitude of the drop in both miss ratio and bus utilization was moderate. The performance gain was less than expected because of the extremely sequential nature of sharing in the programs. Sequential sharing can be measured by several metrics. The most pertinent for a study of invalidation misses is the average number of processors that reread an address between writes by different processors. For all traces this figure averaged around one (1.1 for CELL, .7 for SPICE, .8 for TOPOPT and 1.0 for VERIFY), with the distribution heavily weighted by zeros and ones. (CELL had the most evenly spread distribution, with 2 or more processors rereading between 25 and 21 percent of the time. This accounts for its greater decline in invalidation misses. SPICE had the most skewed distribution, with between 91 and 98 percent of the writes followed by zero or one rereads. Its improvement was the least of the traces.) In actual practice the number of invalidation misses was quite close to the read-broadcast limit of one. This was true even for the traces characterized by fine-grain sharing (TOPOPT and VERIFY). If there had been more processors involved in the contention, read-broadcast would have provided greater benefit.

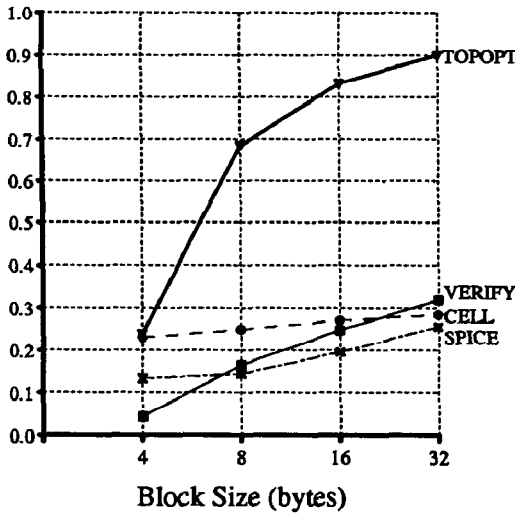


Figure 4-1: Ratio of Invalidation Misses to Total Misses for Read-Broadcast. Under read-broadcast the ratio of invalidation misses to total misses still increases with block size, although the proportions are lower than with Berkeley Ownership. At larger block sizes the invalidation misses for three of the traces have dropped to between a quarter and a third of the total; for TOPOPT they still dominate miss ratio behavior. (The numbers are the geometric mean of the ratio of invalidation to total misses, across all processors.)

Comparison of Berkeley Ownership & Read Broadcast				
Trace	Block Size (bytes)	Bus Utilization		
		Berk. Own.	Read Bdcst.	Change (percent)
CELL	4	42.155	38.470	8.743
CELL	8	39.798	35.849	9.924
CELL	16	38.592	34.383	10.906
CELL	32	42.559	38.042	10.614
SPICE	4	59.546	59.070	0.798
SPICE	8	44.821	44.159	1.477
SPICE	16	40.298	39.948	0.870
SPICE	32	42.221	40.061	5.117
TOPOPT	4	8.925	6.979	21.806
TOPOPT	8	21.289	18.247	14.288
TOPOPT	16	30.972	25.656	17.165
TOPOPT	32	45.108	34.895	22.640
VERIFY	4	49.738	49.346	0.788
VERIFY	8	68.380	66.802	2.307
VERIFY	16	84.760	79.215	6.543
VERIFY	32	96.566	85.491	11.469

Table 4-2: Comparison of Bus Utilization for Berkeley Ownership and Read-Broadcast. This table depicts the decline in bus utilization that occurred with read-broadcast over Berkeley Ownership. (All simulations were run with a 128K byte cache; bus utilization figures are the geometric mean across all processors.)

4.2.2. The Cost in Per Processor and System Throughput

The reduction in invalidation misses did not come for free. Read-broadcast has two side effects that contribute to processor execution time: an increase in processor lockout from the cache and an increase in the average number of cycles per bus transfer. Their consequence for three of the traces was an increase in total execution cycles over the Berkeley Ownership simulations.

The more important of the two factors is the increase in processor lockout from the cache. Cache lockout occurs because of CPU and snoop contention over the shared cache resource. The CPU must use the cache for fetching the current instruction (on a miss in the on-chip instruction cache or for all instructions if there is no on-chip cache), obtaining data referenced by the current instruction, and prefetching subsequent instructions. In machines like the one being simulated, with a RISC-based architecture, no on-chip instruction cache and a cache access time that matches the cycle time of the CPU, the CPU needs to access the cache each cycle.² At the same time, the snoop also needs access to the cache for maintaining coherency. Read-broadcast requires more snoop-related cache activity than Berkeley Ownership, because snoops must deposit data into the cache on some bus reads and more snoops must update the processor's cache state on subsequent invalidations. The first operation does not occur under Berkeley Ownership, and the latter occurs less frequently. Both activities divert the CPU from its normal instruction execution and contribute to program slowdown.

The increase in lockout with read-broadcast was substantial (278 to 305 percent for CELL, 147 to 191 percent for SPICE, 35 to 87 percent for TOPOPT and 143 to 329 percent for VERIFY). On the average 42 percent of total lockout cycles was attributable to taking data on read-broadcasts, and 40 percent to the state updates. (Cache-to-cache transfers account for the remainder.) The increase due to these factors was softened somewhat by the lockout savings from a decline in cache-to-cache transfers that had satisfied invalidation misses under Berkeley Ownership.

However, in terms of total execution cycles, processor lockout was a minor cost. The ratio of lockout to total cycles averaged 5.8 percent for all traces, across most block sizes. The lone exception was VERIFY's 32 byte block simulation, in which processor lockout accounted for an appalling 21 percent of total cycles. The importance of processor lockout is that for three of the traces (CELL, SPICE and VERIFY), its increase *wiped out the benefit to total execution cycles gained by the decrease in invalidation misses*. The consequence was a slight increase in total execution cycles, ranging from .9 to 3.6 percent. The lone exception was TOPOPT,

² In CPUs with instruction caches on-chip, prefetching accesses would replace many of the instruction accesses.

in which the benefit from declining invalidation misses was greater than the cost of processor lockout; here the improvement in total execution cycles varied from .1 to 7.7 percent, as block size increased from 4 to 32 bytes.

The negative effect of processor lockout would not be as severe with a more optimized cache controller implementation. In the SPUR implementation, the priority for using the cache belongs to the processor rather than the snoop, and the two run on asynchronous clocks. Therefore the snoop must negotiate to obtain use of the cache (via separate request and grant cycles), and acknowledge that it has finished. A more optimized implementation would eliminate the handshaking cycles by using a single clock for the entire system.

A lower bound can be placed on processor lockout by eliminating the extra cycles from the above results: read-broadcast is then assumed to cost only the number of cycles needed to fill the cache. The results indicate that, even under these best case assumptions, the increase in processor lockout cycles is greater than the decrease in invalidation miss cycles for more than half the simulations. For these simulations read-broadcast still causes a net gain in total execution cycles. (The major exception was TOPOPT. Since it had fewer execution cycles under read-broadcast even with the less optimized implementation, it is not surprising that the lower bound assumptions would bring further improvement.)

The second factor that contributed to an increase in processor execution time was a rise in the average number of cycles per bus transaction. The increases ranged from .3 to 3.1 percent, for all traces and over all block sizes, and averaged around one. There are two causes. The first is the additional cycle required in the read-broadcast implementation for the snoops to acknowledge that they have completed the operation. Under write-invalidate the same snoops are not actively involved in the bus operation; they merely do a lookup and decide to take no action. The lookup can easily be subsumed in the time required for either the cache-to-cache or memory transfer that satisfies the invalidation miss. The second is the need to update the processor's state on both read-broadcasts and simple state invalidations. For both operations more caches are involved than with invalidation misses and state invalidations under Berkeley Ownership. Therefore there is a greater probability that the update will be delayed, because the processor is using the cache to service a memory request.

4.3. Write-Invalidate/Read-Broadcast Summary

The criticism of write-invalidate, that multiple-processor contention within the block would cause excessive invalidation misses as block size is increased, was not born out by the analysis of these traces. It is true that the number of invalidation misses rose with increasing block size, and for the traces with fine-grain sharing this caused an adverse effect on miss ratios and bus utilization. However, most of these misses were caused by a reread by a *single* processor. Therefore the read-

broadcast solution had less impact than was originally postulated.

Still, at first glance it appears that read-broadcast is a good extension to the write-invalidate protocols, primarily because it is an extremely low cost solution (in terms of additional cache controller complexity) for the moderate benefit it provides. However, when the increase in both processor lockout and average cycles per bus transaction are considered, for most of the simulations the result is a net *gain* in total execution cycles.

Read-broadcast would be more beneficial if two conditions were different. The most important is if the workload were one in which more processors were contending for the data (for example a one producer/several consumers situation). In this case the reduction in invalidation misses would be greater. The second condition, which is a second order effect, is a more optimized cache controller implementation, designed to minimize the cycles consumed during processor lockout.

5. The Write-Broadcast Protocols

5.1. Protocol Description

Write-broadcast protocols broadcast updates to shared addresses, so that all caches and memory have access to the most current value. Blocks are known to be shared through the use of a special bus line. Snoops assert this signal whenever they address match on an operation for a block that resides in their caches. When a writing processor detects an active shared line, it issues a broadcast. In the absence of an active shared signal, the processor completes the write locally. Thus, the signal provides write-through for shared data, but allows a copy-back memory update policy to be used for private data.

Write-broadcast protocols have potential performance benefits for both private and actively shared blocks. First, an inactive shared line prevents needless bus operations to data that reside only in the cache of the writing processor. In addition, because it broadcasts all shared updates, write-broadcast avoids the pingponging of shared data that occurs in programs with inter-processor data contention under write-invalidate. However, for data that is shared in a sequential fashion, with each processor accessing the data many times before another processor begins, the write-through policy for shared data may degrade bus performance.

In the write-broadcast protocols coherency overhead stems entirely from the bus broadcasts to shared data. They occur for all updates to data that is contained in more than one cache, and for the first update to an address after the writing processor has the only copy. (In this case the block has been replaced in the other caches.)

The particular write-broadcast protocol that has been used in this study is the Firefly protocol,

implemented on the DEC SRC Firefly [Thac88]. It differs from other write-broadcast protocols in that it updates memory simultaneously with each write to shared data.

5.2. The Write-Broadcast Trouble Spot

[Egge89b] demonstrated that sharing-related bus traffic will require multiprocessors to have larger or more complex caches than uniprocessors to obtain comparable performance. The requirement is particularly troublesome for the write-broadcast protocols, because larger cache sizes can cause an increase in broadcast operations. As cache size grows, the lifetime of cache blocks increases, because of a decline in block replacements. Shared data tends to remain in a cache for longer periods of time, long past the point when its processor has finished accessing it. However, its presence in the cache drives the shared bus line, giving the illusion of sharing. Therefore write-broadcasts continue for data that is no longer being actively shared.

5.3. Empirical Support for the Trouble Spot

The traces confirm this analysis. For all traces, the number of write-broadcasts rises with increasing cache size (see Figure 5-1). CELL and SPICE have a much larger increase than TOPOPT and VERIFY (84.2 and 100.3 percent over the entire cache size range, versus 3.7 and 15.2 percent). The steepness of their rise correlates with several factors, the most important of which is the pattern of inter-processor references to shared data. For CELL and SPICE this pattern is characterized by good

per processor locality (sequential sharing) for shared data within a coherency block. Sequential sharing is indicated by long average write run lengths³ for the blocks. (The exact figures are 4.9 for CELL and 6.2 for SPICE.) In small caches not all the writes in a long write run result in write-broadcasts. First, shared data is replaced more frequently than in larger caches, and, secondly, in these traces only two processors are involved in the sharing the vast majority of the time. The combined effect is that data may reside in only one cache for the final writes in a write run, allowing these writes to take place locally. In an infinite cache, *all* writes become write-broadcasts, because blocks remain in the cache indefinitely. Therefore, as cache size increases, more writes in a long write run will result in bus broadcasts; and the greater the average write run length, the greater the increase in write-broadcasts. TOPOPT and VERIFY, on the other hand, had short average write run lengths, 1.21 and 2.2, respectively. The smaller length was one of the factors responsible for the more level write broadcast curves, as cache size increased.

A second factor contributing to the shape of the curves is the rate of block replacement. Within a particular trace, the increase in write-broadcasts (with cache size) is most pronounced for smaller caches, where the drop in block replacements is also greatest. Thirdly, at large cache sizes the working sets of TOPOPT and VERIFY fit into the cache. The number of block replacements drops to zero and the level of write-broadcasts remains constant.

Despite the rise in write-broadcasts, bus utilization fell for all traces (see Figure 5-2).⁴ The decrease is due to the positive effects of increasing cache size on the uniprocessor component of bus utilization, which dropped an average of 84 percent over the cache size range. It is offset somewhat by the increase in write-broadcast cycles (see a representative trace in Figure 5-3).

For all traces, the proportion of write-broadcast cycles within total cycles increased dramatically with increasing cache size (see Figure 5-4). The increase only leveled off at the point at which the working set of the program fit into the cache. At the largest cache sizes the write-broadcast cycles dominated bus activity for all traces. The high ratio of sharing cycles to total cycles means that with large cache sizes, sharing bus traffic will

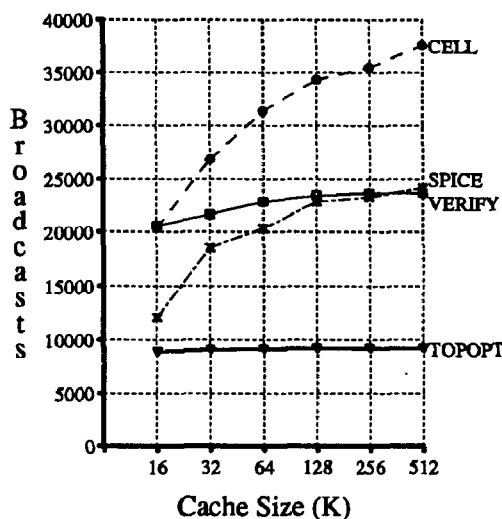


Figure 5-1: Write Broadcasts to Shared Data under Firefly. In the Firefly protocol the number of write-broadcasts increases with increasing cache size for all traces, giving credence to the "illusion of sharing" hypothesis.

³ A write run is a sequence of write references to the shared addresses in a coherency block by a single processor, uninterrupted by any accesses by other processors. The length of a write run is the number of writes it contains. The average write run length is that figure, averaged over all coherency blocks [Egge88]. In other words the average write run length is the average number of writes that are issued for the addresses within a particular block, each time a new processor writes to them.

⁴ The only exception is the transition to a 512K byte cache for SPICE.

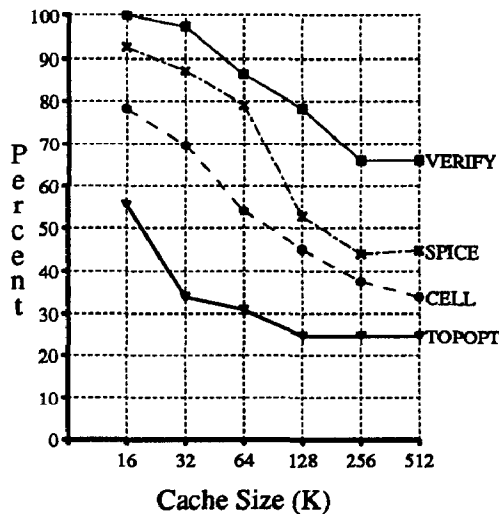


Figure 5-2: Bus Utilization under Firefly. Despite the rise in write-broadcasts, bus utilization fell because of the benefits of large caches on uniprocessor misses.

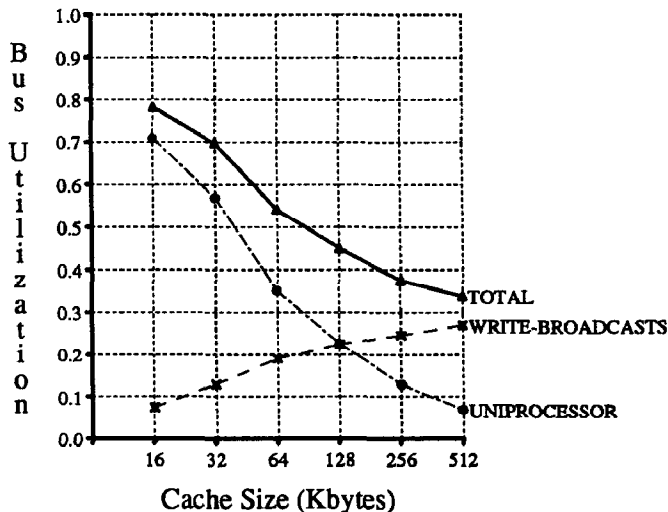


Figure 5-3: Bus Cycles for CELL under Firefly. This classification of bus cycles for CELL illustrates the effect of write-broadcast cycles on total bus cycles, using the Firefly protocol. Write-broadcast cycles rise with increasing cache size; uniprocessor bus cycles tend to fall. The two effects produce bus utilization that still declines, but less steeply than for uniprocessor programs.

be the cause of the bus bottleneck. Therefore a protocol that limits the number of write-broadcasts is desirable.

6. Competitive Snooping

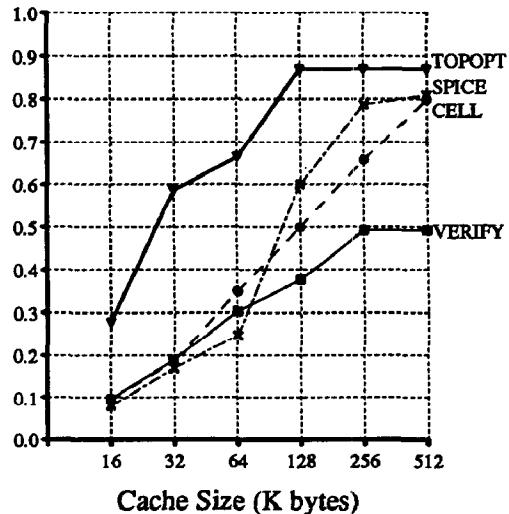


Figure 5-4: Ratio of Broadcast Cycles to Total Bus Cycles. The ratio of write-broadcast cycles to total bus cycles increases with increasing cache size under Firefly. The rise is much steeper for the traces with longer average write run lengths, CELL and SPICE.

6.1. Protocol Description

Competitive snooping [Karl86, Karl88] is a write-broadcast protocol that switches to write-invalidate when the breakeven point in bus-related coherency overhead between the two approaches is reached. The breakeven point for a particular address occurs when the sum of the write broadcast cycles issued for the address equals the number of cycles that would be needed for rereading the data had it been invalidated. Competitive snooping thus limits coherency overhead to twice that of optimal.⁵

The first algorithm proposed in [Karl86] (called "Standard-Snoopy-Caching") assumes that an adversary can choose any processor to either write or reread a shared address. A counter, whose initial value is the cost in cycles of a data transfer, is assigned to each cache block in every cache. On a write broadcast, a cache that contains the address of the broadcast is (arbitrarily)⁶

⁵ Larry Rudolph makes a very apt analogy between the rationale behind competitive snooping and the dilemma faced by any novice skier. The beginning skier is hesitant to buy skis immediately for fear that his/her interest in skiing might be a passing fancy. On the other hand renting week after week can be costly. The pivotal question is therefore *when* to stop renting and make the purchase. Not knowing ahead of time which will be his or her preference, the budding skier should rent until he or she has spent an amount equivalent to the purchase price of new skis; and then buy the skis. Like competitive snooping, this course of action limits the total cost to twice that of optimal.

⁶ The particular choice of cache does not affect the worst-case bound.

chosen, and its counter is decremented. When a counter value reaches zero, the cache block is invalidated. When all counters for an address, other than that of the writer, are zero, write-broadcasts for it cease. Any reaccess by a processor to an address resets its cache's counter to the initial value. The algorithm's lower bound proof demonstrates that the total costs of invalidating are in balance with the total costs of rereading.

In an alternate algorithm (called "Snoopy-Reading") the adversary is allowed to read-broadcast on rereads. In order to obtain the lower bound of the previous algorithm, the coherency algorithm is given the same capability. All other caches with invalidated copies take the data, and reset their counters. As in the original scheme, when a cache's counter reaches zero, it invalidates the block containing the address; and write broadcasts are discontinued, when all caches but that of the writer have been invalidated.

Read-broadcasting by the adversary also prompts other changes in the coherency algorithm. For example, on a write-broadcast *all* caches that contain the updated address decrement their counters rather than only one; and the decrementing is done on consecutive write broadcasts *by a particular processor*, rather than any processor. The simultaneous decrements complement the simultaneous cache updates on read-broadcasts, i.e., they reduce the costs of broadcasting to match the cheaper rereads. The single writer requirement corresponds to all counters being reset on an access by another processor. More than one processor referencing the data indicates (obviously) that there is sharing. As long as data is shared, a good competitive coherency algorithm will broadcast rather than invalidate. Broadcasting occurs as long as counter values are greater than zero. Therefore when a processor other than the current writer accesses the data, all counters are reset to force broadcasting.

The advantages of the alternate scheme over the original are that (1) it is well suited for a workload in which there are few rereads (as is the case with these traces) and (2) its implementation doesn't require hardware to "arbitrarily" choose a cache for counter decrementing. When there are few rereads, a competitive coherency algorithm should make the data private sooner rather than later, in order to avoid unnecessary broadcasts. By requiring all processors to decrement their counters simultaneously, Snoopy-Reading can invalidate more quickly than Standard-Snoopy-Caching.

In the simulator's implementation of Snoopy-Reading, a writing processor keeps track of the number of its consecutive writes to each address (through cache state values). When the breakeven point for broadcasts has been reached, it signals to the other caches to invalidate. The breakeven point was defined to be the maximum of the ratio of data transfer to write-broadcast cycles that is used in the algorithm and the value three. The constant insures that write-broadcasts will continue long enough to prevent busywaiting over the bus. A processor uses the first of the three broadcasts for setting the

lock, and the second for clearing it. At this point the lock is still present in other caches, and processors can detect locally that it has been freed. On the third broadcast (which, if it occurs, demonstrates that the address is not a lock), the data is invalidated. This implementation requires a six-value coherency state, and a correspondingly larger PLA for both the snoop and the portion of the cache controller that services memory requests for the CPU.

6.2. Competitive Snooping Results

Competitive snooping decreased the number of write-broadcasts issued for all traces (see Table 6-1). The benefit was greater for those traces with sequential sharing (CELL and SPICE). (Recall that their average write run lengths were 4.9 and 6.2.) Given the breakeven

Write-Broadcasts				
Trace	Cache Size (Kbytes)	Firefly	Comp. Snoop.	% Change
CELL	16	20402	13199	35.31
CELL	32	26841	15507	42.23
CELL	64	31300	15514	50.43
CELL	128	34287	15212	55.63
CELL	256	35444	15192	57.14
CELL	512	37579	15338	59.18
SPICE	16	12076	4510	62.65
SPICE	32	18555	5900	68.20
SPICE	64	20362	6373	68.70
SPICE	128	22925	7045	69.27
SPICE	256	23344	7251	68.94
SPICE	512	24184	7412	69.35
TOPOPT	16	8918	8218	7.85
TOPOPT	32	9111	8352	8.33
TOPOPT	64	9190	8410	8.49
TOPOPT	128	9244	8458	8.50
TOPOPT	256	9244	8458	8.50
TOPOPT	512	9244	8458	8.50
VERIFY	16	20589	18091	12.13
VERIFY	32	21726	18835	13.31
VERIFY	64	22914	19097	16.66
VERIFY	128	23476	19107	18.61
VERIFY	256	23719	19330	18.50
VERIFY	512	23719	19330	18.50

Table 6-1: Comparison of Write-Broadcasts for Firefly and Competitive Snooping. This table depicts the decline in the number of write-broadcasts that occurred with competitive snooping. The drop was most pronounced for CELL and SPICE, which had the longest average write run lengths. Identical values across cache sizes for TOPOPT and VERIFY indicate that their working sets fit into the caches. (All simulations were run with a 32 byte block.)

Sharing Cycles							
Trace	Cache Size (Kbytes)	Firefly	Competitive Snooping				Percentage Change
		Write Broadcasts	Write Broadcasts	Invalidations	Invalidation Misses	Total	
CELL	16	167122	108850	24489	17820	151159	9.55
CELL	32	221925	129716	33051	28706	191473	13.72
CELL	64	259327	130740	37395	39140	207275	20.07
CELL	128	285430	129361	40597	51286	221244	22.49
CELL	256	295069	129527	41450	55567	226544	23.22
CELL	512	312668	130360	42849	57944	231153	26.07
SPICE	16	102645	39190	7912	2236	49338	51.93
SPICE	32	158119	51491	13660	12786	77937	50.71
SPICE	64	172139	55384	15115	15168	85667	50.23
SPICE	128	191106	60515	18126	18068	96709	49.40
SPICE	256	193971	61880	18491	18262	98633	49.15
SPICE	512	200782	63020	19076	18907	101003	49.70
TOPOPT	16	75828	74927	1603	2655	79185	-4.43
TOPOPT	32	77214	76249	1916	3366	81531	-5.59
TOPOPT	64	77936	76821	1920	3238	81979	-5.19
TOPOPT	128	78256	77120	1942	3380	82442	-5.35
TOPOPT	256	78256	77120	1942	3380	82442	-5.35
TOPOPT	512	78256	77120	1942	3380	82442	-5.35
VERIFY	16	170952	155223	9228	8679	173130	-1.27
VERIFY	32	183516	165910	10798	12157	188865	-2.91
VERIFY	64	194813	170477	12007	15809	198293	-1.79
VERIFY	128	199733	171116	12744	18125	201985	-1.13
VERIFY	256	200341	171961	13323	19132	204416	-2.03
VERIFY	512	200341	171961	13323	19132	204416	-2.03

Table 6-2: Comparison of Sharing Cycles for Firefly and Competitive Snooping. This table depicts the difference in the number of cycles for the sharing-related bus operations for Firefly and competitive snooping. The decline in write-broadcast cycles is offset by cycles for invalidation signals and invalidation misses. For TOPOPT and VERIFY the combination of a smaller cycle savings in write-broadcasts and the additional cycles related to invalidations produced a net increase in sharing-related cycles. (All simulations were run with a 32 byte block.)

point in the simulations, each trace saved on the average, 2 or 3 broadcasts each time a different processor wrote to a shared address.⁷ The average write run lengths for TOPOPT and VERIFY were below the simulator's breakeven point (1.2 and 2.2, respectively). Therefore no broadcast savings was accrued in most cases.

The corresponding decrease in the number of write-broadcast cycles was offset to varying extents by the additional cycles for invalidation signals and invalidation misses (see Table 6-2). For CELL and SPICE the effect was to reduce the percentage improvement in cycles consumed in sharing-related bus operations to 10 to 26 percent for CELL and 49 to 52 percent for SPICE. However, the savings was still substantial enough to

⁷ Technically this is true only for the larger caches. At smaller cache sizes the savings would be less. See the discussion on the effect of average write run length on write-broadcast protocols in section 5.3.

cause a drop in bus utilization relative to write-broadcast. The decline in bus utilization for CELL ranged as high as 19 percent; for SPICE as high as 30 percent. For all simulations but two (CELL with 16K and 32K byte caches) the lower bus utilization produced fewer total execution cycles.

For TOPOPT and VERIFY the smaller decline in write-broadcasts, coupled with the additional cycles for invalidation signals and invalidation misses, produced an *increase* in sharing-related bus cycles. This increase was responsible for a slight rise in bus utilization over write-broadcast (1.6 to 4.5 percent for TOPOPT and .8 percent at most for VERIFY). Higher bus utilization brought an increase in total execution cycles. (Details on bus utilization and total execution cycles appear in Table 6-5.)

6.3. Write-Broadcast/Competitive Snooping Summary

Bus Utilization & Total Execution Cycles							
Trace	Cache Size (Kbytes)	Bus Utilization			Total Execution Cycles		
		Firefly	Competitive Snooping	Percentage Change	Firefly	Competitive Snooping	Percentage Change
CELL	16	78.21	78.24	-0.04	2251417	2275472	-1.07
CELL	32	69.65	69.41	0.34	1722507	1726670	-0.24
CELL	64	54.07	51.94	3.95	1367997	1358706	0.68
CELL	128	45.13	41.29	8.49	1267754	1246316	1.69
CELL	256	37.52	32.68	12.90	1196530	1170737	2.16
CELL	512	33.88	27.56	18.67	1156534	1128079	2.46
SPICE	16	92.66	92.24	0.46	1385228	1344603	2.93
SPICE	32	87.17	86.09	1.24	1078916	1007891	6.58
SPICE	64	79.10	77.34	2.22	886776	795919	10.25
SPICE	128	52.80	43.43	17.74	603795	517028	14.37
SPICE	256	44.06	31.98	27.42	559356	474377	15.19
SPICE	512	44.88	31.38	30.08	553071	474123	14.27
TOPOPT	16	55.55	56.42	-1.56	491294	495603	-0.88
TOPOPT	32	33.89	34.96	-3.13	389304	391695	-0.61
TOPOPT	64	30.76	31.82	-3.43	381349	382676	-0.35
TOPOPT	128	24.68	25.79	-4.51	364345	364798	-0.12
TOPOPT	256	24.68	25.79	-4.51	364345	364798	-0.12
TOPOPT	512	24.68	25.79	-4.51	364345	364798	-0.12
VERIFY	16	99.97	99.97	0.00	1760674	1786211	-1.45
VERIFY	32	97.41	97.58	-0.17	1002740	1017567	-1.48
VERIFY	64	86.24	86.58	-0.39	744443	749358	-0.66
VERIFY	128	78.18	78.25	-0.08	677634	682098	-0.66
VERIFY	256	65.99	66.08	-0.14	617141	622265	-0.83
VERIFY	512	65.99	66.08	-0.14	617141	622265	-0.83

Table 6-5: Comparison of Bus Utilization & Total Execution Cycles for Firefly and Competitive Snooping. This table depicts the change in the bus utilization and total execution cycles that occurred with competitive snooping. The decrease in sharing-related cycles for CELL and SPICE resulted in a decline in both. And, conversely, the increase in sharing cycles for TOPOPT and VERIFY produced a rise. (All simulations were run with a 32 byte block.)

The extent to which competitive snooping improves the performance of write-broadcast depends on the pattern of references to shared data. When sharing is sequential, as exhibited by relatively longer average write run lengths, the benefit is greatest. Here the savings in write-broadcast cycles decreases bus utilization and total execution time. As inter-processor contention for the shared addresses rises, competitive snooping becomes less attractive. The decrease in write-broadcasts diminishes, and in some cases can be offset by the rise in invalidations and the more expensive (in numbers of cycles) invalidation misses. The result is an increase in bus utilization and total execution time. (An alternative argument is that programs with fine-grain-sharing for shared addresses are a good match for write-broadcast protocols. Therefore, they have less need for competitive snooping, and it consequently provides less benefit.)

7. Summary of the Paper

This paper contains two companion studies of bus-based, shared memory cache coherency protocols. The purpose of each is twofold: first, to measure the performance loss of changing particular cache parameter values on well-known snooping coherency techniques; second, to determine to what extent extensions, designed specifically to eliminate deficiencies in the original protocols, achieve performance improvements. In the first study, read-broadcast was proposed to eliminate the rise in invalidation misses in write-invalidate protocols that occur with increasing block size. In the second, competitive snooping was intended to limit the increase in bus broadcasts caused by increasing cache size in write-broadcast coherency protocols.

Our results have found that neither extension produces a savings in coherency overhead across all workloads studied. In those cases in which there was a performance loss, the original protocol, write-invalidate or write-broadcast, was a good match for the program. Therefore there was not much room for improvement;

and the extension often introduced secondary costs which outweighed the small savings in coherency overhead. Furthermore, both extensions required some additional hardware complexity.

Our particular workload is characterized by sequential sharing, i.e., data is shared by very few processors at a time. Therefore read-broadcast reduced the number of invalidation misses only moderately, and at a high cost in processor lockout from the cache. In some cases, the net effect was an increase in total execution cycles. These results clearly indicate that read-broadcast is inappropriate for programs with sequential sharing. However, if more processors had been involved in the sharing, for example, a single-producer, multiple-consumer situation, read-broadcast would have provided more benefit for a similar cost in processor lockout.

Competitive snooping benefitted only those programs in which the reference pattern to shared data was very sequential. In this case the decline in the number of write-broadcast cycles was greater than the additional cycles introduced by invalidations and invalidation misses; the net effect was a drop in bus utilization. However, for programs characterized by inter-processor contention (fine-grain sharing) for shared addresses, competitive snooping degraded performance by causing a slight increase in bus utilization and total execution time. Competitive snooping works well in programs that would have incurred less coherency overhead with write-invalidate protocols (rather than write-broadcast). The reason is that it uses invalidations to terminate broadcasts to shared data.

Acknowledgements.

We wish to acknowledge the efforts of several others who contributed to the work in this paper. Garth Gibson and Dave Ditzel gave valuable comments on an earlier draft. Mark Manasse consulted on competitive snooping issues. Dominico Ferrari and Yale Patt provided resources for running many of the simulations. Andrea Casotto (CELL), Steve McGrogan (SPICE), Srinivas Devadas (TOPOPT) and Hi-Keung Tony Ma (VERIFY) donated the parallel programs and a considerable portion of their time discussing them. John Sanguinetti wrote the ELXSI trace generator; Dianne DeSousa helped with the generation of the ELXSI trace; Sequent provided the software on which the Sequent trace generator was based; and Frank Lacy generated and postprocessed the Sequent traces. This work was supported by an IBM Predoctoral Fellowship, SPUR/DARPA contract No. N00039-85-C-0269, NSF Grant No. MIP-8352227, Digital Equipment Corporation, and California MICRO (in conjunction with Texas Instruments, Xerox, Honeywell, and Philips/Signetics).

References

- [Agar88] A. Agarwal, J. Hennessy and M. Horowitz, "Cache Performance of Operation System and Multiprogramming Workloads", *ACM Transactions on Computer Systems*, 6, 4 (November 1988), 393-431.
- [Alex86] C. Alexander, W. Keshlear, F. Cooper and F. Briggs, "Cache Memory Performance in a UNIX Environment", *Computer Architecture News*, 14, 3 (June 1986), 14-70.
- [Arch86] J. Archibald and J. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors", *ACM Transactions on Computer Systems*, 4, 4 (November 1986), 273-298.
- [Caso86] A. Casotto, F. Romeo and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells", *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, CA (November 1986), 30-33.
- [Deva87] S. Devadas and A. R. Newton, "Topological Optimization of Multiple Level Array Logic", *IEEE Transactions on Computer-Aided Design* (November 1987).
- [Egge88] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu HA (May 1988), 373-383.
- [Egge89a] S. J. Eggers, "Simulation Analysis of Data Sharing in Shared Memory Multiprocessors", Ph.D. thesis, University of California, Berkeley (March 1989).
- [Egge89b] S. J. Eggers and R. H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs", *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston MA (April 1989).
- [Gibs88] G. A. Gibson, "SpurBus Specification", to appear as Computer Science Division Technical Report, University of California, Berkeley (December 1988).
- [Good87] J. R. Goodman, "Cache Memory Optimization to Reduce Processor/Memory Traffic", *Journal of VLSI and Computer Systems*, 2, 1 & 2 (1987), 61-86.
- [Good88] J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor", *Proceedings 15th Annual International Symposium on Computer Architecture*, Honolulu HA (May 1988), 422-431.
- [Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson,

- P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", *IEEE Computer*, 19, 11 (November 1986), 8-22.
- [Hill87] M. D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance", Technical Report No. UCB/Computer Science Dpt. 87/381, University of California, Berkeley (November 1987).
- [Karl86] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator, "Competitive Snoopy Caching", *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, Toronto, Canada (October 1986), 244-254.
- [Karl88] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator, "Competitive Snoopy Caching", *Algorithmica*, 3 (1988), 79-119.
- [Katz85] R. Katz, S. Eggers, D. Wood, C. L. Perkins and R. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 13, 3 (June 1985), 276-283.
- [Ma87] H. T. Ma, S. Devadas, R. Wei and A. Sangiovanni-Vincentelli, "Logic Verification Algorithms and their Parallel Implementation", *Proceedings of the 24th Design Automation Conference* (July 1987), 283-290.
- [McGr86] S. McGrogan, R. Olson and N. Toda, "Parallelizing Large Existing Programs - Methodology and Experiences", *Proceedings of Spring COMPCON* (March 1986), 458-466.
- [Patt85] D. A. Patterson, "Reduced Instruction Computers", *Communications of the ACM*, 28, 1 (January 1985), 8-21.
- [Sega84] Z. Segall and L. Rudolph, "Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor", *Proceedings of the 11th International Symposium on Computer Architecture*, 12, 3 (June 1984), 340-347.
- [Smit87] A. J. Smith, "Line (Block) Size Choice for CPU Caches", *IEEE Trans. on Computers*, C-36, 9 (September 1987), 1063-1075.
- [Thac88] C. P. Thacker, L. C. Stewart and E. H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation", *IEEE Transactions on Computers*, 37, 8 (August 1988), 909-920.
- [Wood87] D. A. Wood, S. J. Eggers and G. A. Gibson, "SPUR Memory System Architecture", Technical Report No. UCB/Computer