THE EFFECT OF SHARING ON THE CACHE AND BUS PERFORMANCE OF PARALLEL PROGRAMS

Susan J. Eggers and Randy H. Katz

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

Abstract

Bus bandwidth ultimately limits the performance, and therefore the scale, of bus-based, shared memory multiprocessors. Previous studies have extrapolated from uniprocessor measurements and simulations to estimate the performance of these machines. In this study, we use traces of parallel programs to evaluate the cache and bus performance of shared memory multiprocessors, in which coherency is maintained by a write-invalidate protocol. In particular, we analyze the effect of sharing overhead on cache miss ratio and bus utilization.

Our studies show that parallel programs incur substantially higher miss ratios and bus utilization than comparable uniprocessor programs. The sharing component of these metrics proportionally increases with both cache and block size, and for some cache configurations determines both their magnitude and trend. The amount of overhead depends on the memory reference pattern to the shared data. Programs that exhibit good per-processor-locality perform better than those with fine-grain-sharing. This suggests that parallel software writers and better compiler technology can improve program performance through better memory organization of shared data.

1. Introduction

The cache behavior of uniprocessor programs has been extensively analyzed (e.g., [Agar88a, Alex86, Good87, Hill87, Przy88, Smit85, Smit87]), and the effect on performance of changing cache parameters is now well understood. For small and medium sized caches, increasing the cache size, despite the additional cache access time, causes a drop in the miss ratio that is substantial enough to reduce the effective memory access time. For very large caches, the miss ratio still falls; but the balance reverses between the decreasing miss ratio and an increasing cache access time, and the effective access time begins to rise. The miss ratio trend for increasing block size is not as consistent. A larger block size also reduces miss ratios, but only up to a certain size. After reaching this memory pollution point, miss ratios begin to climb. But even the declining miss ratio does not always increase performance, because of additional bus traffic latency caused by the larger transfer block.

Cache and bus metrics of parallel programs should be higher than those of uniprocessor programs, because additional bus traffic is required to maintain coherent caches. This is a critical performance issue in single-bus multiprocessor design, since bus bandwidth is the limiting performance factor in such a system. If the cache and bus behavior of parallel programs, varying across cache and block sizes, is radically different from uniprocessor programs, then new rules of thumb are needed to design memory systems for multiprocessors.

The goal of this research is twofold: first, to analyze (via trace-driven simulation) the cache and bus behavior of parallel programs running under write-invalidate coherency protocols; and second, to compare this behavior to that of their uniprocessor counterparts. In particular, we study parallel programs that (1) execute on single-bus, shared memory multiprocessors, in which cache coherency is maintained in hardware by a write-invalidate coherency protocol, and that (2) use a programming paradigm with large grain parallelism, i.e., a process.

Our research shows that parallel programs do have different cache and bus behavior than uniprocessor programs; and that it is the references to shared data that are responsible for the difference. The results are most dramatic for increasing block size. Here the proportion of sharing-related misses to total misses rises, and either elevates miss ratios without changing their declining trend, or sometimes reverses the trend. Sharing also worsens the miss ratios when increasing cache size; again, the effect is more pronounced with larger cache sizes. For most programs sharing-related bus traffic dominates bus utilization cycles with large caches (128K bytes and up) and all block sizes studied (4 to 32 bytes). At these cache configurations it is the sharing traffic that creates the multiprocessor bus bottleneck.

These results indicate that larger caches and block sizes, the traditional techniques for improving cache performance, are less effective with parallel programs than uniprocessor programs. However, additional performance improvements can still occur using software techniques. For the programs analyzed, the amount of sharing overhead depended on the intra-block memory reference pattern for shared data. Programs that exhibit good per-processor-locality performed better than those with fine-grain-sharing. If programmers (or compilers) are aware of memory reference behavior when writing (generating) parallel code, they can attain better program performance by altering the memory organization of the shared data.

In previous work we have characterized the pattern of references to shared data; in this paper we analyze its effect on cache and bus performance. The remainder of the paper begins with three background sections: the first covers aspects of the methodology and the workload; the second introduces write-invalidate protocols and the type of sharing costs they induce; and the third characterizes memory references of parallel programs and discusses how they affect write-invalidate sharing costs. Sections 5 and 6 contain studies of the cache and bus behavior of parallel programs, each investigating the effects of changing both block and cache size. The last section summarizes the results; the summary discusses the implications of cache and bus performance of the parallel programs, both for multiprocessor cache design and software design.

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 [Patt85], assuming one cycle per instruction execution. Not all instructions follow this model, e.g., multiply and divide; therefore the bus utilization results will be slightly overestimated, because the simulation processors return to use the bus more quickly than in a real machine. With the exception of those cache parameters that are varied in the studies, 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 the Berkeley Ownership cache coherency protocol [Katz85], in-cache address translation [Wood86], segment-based addressing, no fetch-bypass on reads, a test-and-test-and-set sequence for securing locks [Wood87], and many of the timing constraints of the actual SPUR implementation. Bus activity is implemented using a modified NuBus 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 (see Table 2-1). The choice of application area was deliberate, so that the workload being analyzed was appropriate for the underlying architecture. 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 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 the same 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 shared data (see Figure 2-1). The shared data are 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 (via software trace generators) on a per-processor basis. The number of processors in the simulations depends on the number of processors that were 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 dynamically in the simulator. Statistics are generated from approximately 300,000 references per processor, after steady state has been reached. (See [Egge88] for a more detailed discussion of the methodology.)

3. Write-invalidate Cache Coherency Protocols

We have chosen to support coherency with a hardware cache coherency protocol. We believe that a hardware technique will have better performance on single-bus, shared memory multiprocessors than known solutions in software. Write-invalidate is a type of distributed, hardware protocol, that is implemented on each cache controller in the multiprocessor. It maintains 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 subsequent updates (provided there are no intervening accesses by other processors) without either violating coherency or further utilizing the bus. The

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.

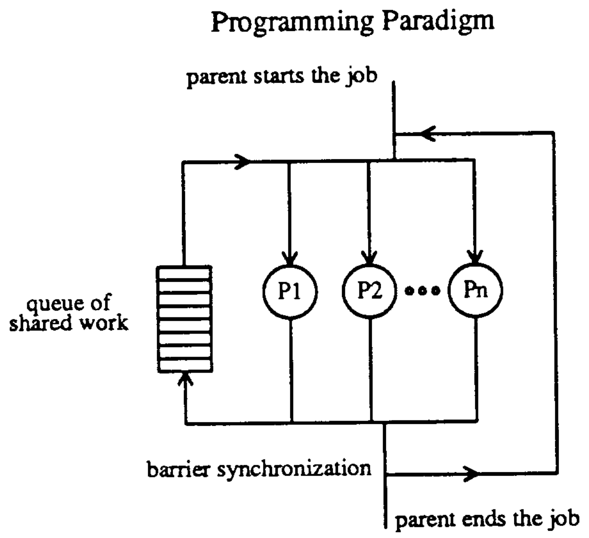


Figure 2-1: Flow Chart of the Parallel Programming Paradigm. This simplified representation illustrates the programming paradigm of the parallel programs. The parent process starts and ends the program, and forks child processes that do the parallel portion of the computation. Each child process executes the same code. At certain points in the parallel computation, the children resynchronize and then repeat the computation. Within each iteration, the children process different portions of the work queue, which resides in shared memory. For example, in a parallelized circuit simulator the circuit would be divided into groups of devices (nodes). In each iteration, each child would process a particular node. Data sharing occurs because the inputs and outputs of the nodes interconnect, and a node may be processed by different child processes in different iterations.

invalidation is accomplished by an invalidating bus operation. Caches of other processors monitor the bus via the snoop portion of their cache controllers. When they detect a match between the address associated with the invalidation signal and one of their cache tags, they invalidate the entire cache block containing the address.

In write-invalidate protocols, there are two sources of bus-related coherency overhead. The first is the invalidation signal needed to maintain coherent caches. 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). (A more detailed description of the write-invalidate protocols appears in [Arch86].)

4. Characterizing the Memory Reference Behavior of Shared Data

The pattern of references to shared data can be characterized by two distinct modes of behavior. In the first, *per-processor-locality* within the cache block, a particular processor makes multiple, consecutive writes to the words within a block, uninterrupted by accesses from other processors. In the other, *fine-grain-sharing*, processors contend for one or more words within the block, and the number of per processor consecutive writes is very low.

Whether a program exhibits per-processor-locality or fine-grain-sharing affects the amount of coherency overhead incurred. Per-processor-locality reduces coherency overhead by decreasing both the number of invalidations and the number of invalidation misses. Conversely, when there is fine-grain-sharing, the number of invalidations and invalidation misses is higher. For both types of memory reference behavior, the larger the cache block, the more pronounced the effect on coherency overhead. (See [Agar88b, Egge88] for additional discussion on the characterization and analysis of sharing.)

Per-processor-locality can benefit both the writer and the readers of a cache block containing a shared address. For example, after an invalidation, a writing processor possesses the only cached copy of the block. It pays the coherency overhead (the invalidation signal) for the first write to the block, but can update the remaining words without additional bus operations. An analogous situation exists for the readers. In this case the invalidation miss penalty is paid only for the first read to the block. All other reads are cache hits, and, from the point of view of coherency overhead, are free.

The performance loss of fine-grain-sharing also occurs for both writers and readers of the shared data. Alternating writes by different processors to the different words within a block produce separate invalidations for each write. The increased number of invalidations is responsible for a subsequent rise in invalidation misses. The greater the number of processors contending for an address, the greater the number of invalidation misses.

5. The Effect of Sharing on Miss Ratios

5.1. Varying Block Size

Cache miss ratio studies of uniprocessor programs have indicated that for a fixed size cache, the miss ratio initially drops as the block size of the cache increases [Agar88a, Alex86, Good87, Hill87, Przy88, Smit87]. The decline is due to improved cache hits because of locality of reference. However, as block size continues to increase, the decrease in the miss ratio tapers off. For small and medium sized caches, those in the range of 4K bytes to 16K bytes, the miss ratio decline may terminate at some particular block size (in [Good87], it is 32 bytes and 128 bytes, respectively), after which the miss ratio begins to rise. The termination is known as the memory pollution point.[1] As cache size grows, the pollution point shifts to an increasingly larger block size. For 128K bytes caches, [Agar88a] reports that the pollution point is not reached with block sizes up to 32 bytes (the configurations in this study). Therefore, for uniprocessor programs, the miss ratios should continue to decline until that point.

Analysis of the parallel traces indicates that their miss ratios do not always follow the trend of uniprocessor programs (see Figure 5-1). CELL and SPICE consistently exhibit the expected decline; but the miss ratios for TOPOPT and VERIFY actually *increase*. Their rise is slight, and, for one of the traces, not continuous across all block sizes. However, their behavior was completely unexpected, given the uniprocessor literature. Miss ratios for the shared references only,[2] depicted in Figure 5-2, indicate almost identical behavior, but at higher miss ratios. Due to the poorer locality of reference for shared data, miss ratios for shared data were 5 to 7 times greater than total miss ratios, depending on the particular trace and block size.

Eliminating invalidation misses from the miss ratio calculation leaves a uniprocessor component that approximates uniprocessor miss ratios. When the invalidation misses were excluded from this analysis, the uniprocessor miss ratio component for all traces corroborated results from previous cache studies of uniprocessor programs. In other words, the uniprocessor miss ratio component declined as block size increased, and the marginal rates of decline also decreased with block size (see Figure 5-3). The actual miss ratio values were less than the multiprogramming miss ratios reported in [Agar88a]; however, this is to be expected, since the traces contain applications references only. The miss ratio trend of SPICE is most typical of the results of the composite applications workload reported in [Good87].

The predictable trend of the uniprocessor component of the miss ratios suggests that it is the invalidation misses that are responsible for the variable miss ratio behavior of the parallel programs. A more detailed examination reveals that two interacting factors determine the miss ratio trends (see Figure 5-4). First, as block size increases, invalidation misses become a larger fraction of total misses. Therefore they become an increasingly significant determinant of miss ratio behavior. Second, at small block sizes, the uniprocessor misses dominate. However, at larger block sizes the number of invalidation misses is either a substantial (CELL, VERIFY and SPICE) or overwhelming (TOPOPT) proportion of the total. The combination of these factors forces the miss ratios to follow the trend of the invalidation misses. For many block sizes the invalidation misses are the single most

[1] For a fixed sized cache, a larger block size results in fewer cache lines. The pollution point occurs because memory references take place to noncontiguous data that do not reside in the cache, while contiguous, but unreferenced, data remain in the larger block. Until the pollution point is reached, the larger block size implicitly prefetches data that will be referenced in the near future.

[2] The shared miss ratio is the number of misses to shared data divided by total references to shared data.

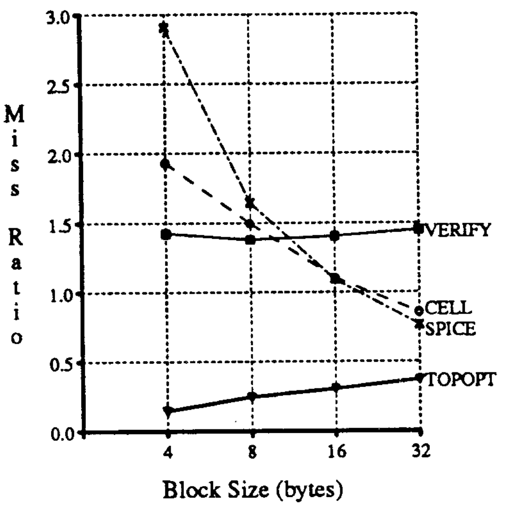


Figure 5-1: Miss Ratios
Parallel traces exhibit two miss ratio trends as block size increases. Miss ratios decline for programs with per-processor-locality (CELL and SPICE); however, for programs with fine-grain-sharing (TOPOPT and VERIFY), they are dominated by the misses caused by intra-block contention for shared data, which produce a rising curve.

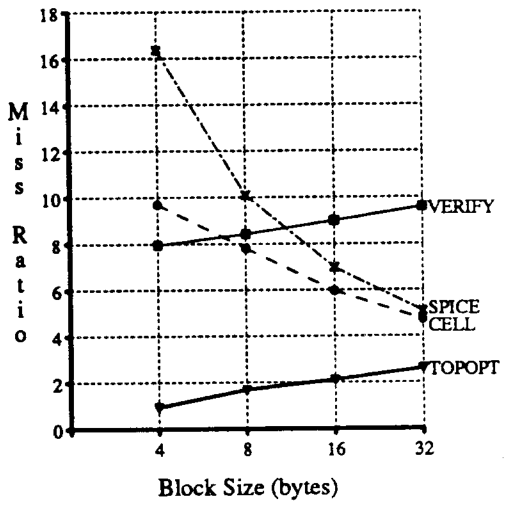


Figure 5-2: Shared Miss Ratios
The shared miss ratios (misses to shared data divided by total references to shared data) follow the trend of the total miss ratios, but have a value that is 5 to 7 times higher. The higher figures reflect the poorer locality of shared data. (All block size graphs are for a 128K byte cache.)

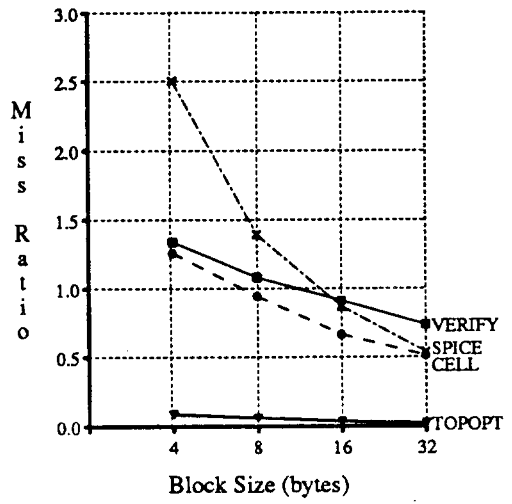


Figure 5-3: Uniprocessor Component of the Miss Ratio
The uniprocessor component of the miss ratio for parallel programs mimics the declining miss ratio behavior of uniprocessor programs. This suggests that the variability of the miss ratio curves for parallel programs is caused by the invalidation misses

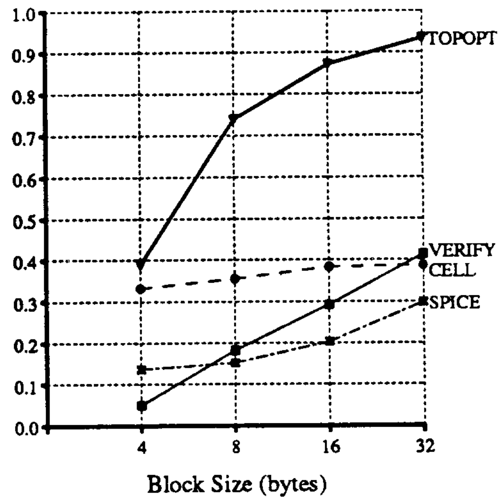


Figure 5-4: Ratio of Invalidation Misses to Total Misses
The ratio of invalidation misses to total misses increases as block sizes increase. At larger block sizes the invalidation misses of three of the traces comprise a substantial portion of the total; and for TOPOPT they dominate the miss ratio behavior. (The numbers are the geometric mean of the ratio of invalidation to total misses, across all processors.)

determining factor in miss ratio behavior.

The traces exhibited two distinct invalidation miss trends.[3] For programs whose memory reference pattern for shared data is dominated by per-processor-locality, such as CELL and SPICE, the number of invalidation misses declines as block size is increased (see Figures 5-5 and 5-6). In these traces the processors tended to read several contiguous words in succession, all of which had been previously invalidated. For example, consider a 32 byte and 4 byte block size; with the larger block size, the invalidation miss penalty is incurred only for the first of eight words; with the smaller block, it is incurred for each. Because the invalidation miss trend reinforced that of the uniprocessor miss ratios, the miss ratios declined. SPICE, in particular, had good locality of reference. Its data structures had been sized to the ELXSI 6400 64 byte cache block, explicitly to avoid fine-grain-sharing for addresses within the block. Therefore for block sizes considered in this study (up to 32 bytes), little contention was observed.

For programs with fine-grain-sharing within a block, such as TOPOPT and VERIFY (see Figures 5-7 and 5-8), the declining uniprocessor miss ratio was offset by the increase in the number of invalidation misses and their large proportion within total misses. Invalidation misses had the largest effect on TOPOPT, and for two reasons; first, the trace had the most fine-grain-sharing, and second, it had a low uniprocessor miss ratio, because its working set fit into the 128K byte cache.

A short note should be made about the miss ratio behavior of the components of shared data, i.e., locks and the shared applications data they protect. The applications data are responsible for the high shared miss ratio depicted in Figure 5-2. Miss ratios for the locks were considerably lower, indicating that sharing in all programs was very sequential, i.e., there was little contention for locks. This behavior is partially determined by the programming paradigm. When the programs first begin execution, there is unusual contention for the locks protecting the queue of work, since all child processes try to take their first unit of work simultaneously. However, only one process will obtain access to the queue at a time. Since each process does a comparable amount of computation, they will thereafter access the queue in the same order, spaced in time by the execution time of the critical section. This self-scheduling is disrupted by synchronization barriers, which are used to separate phases in the computation. The disruption causes more busywaiting and therefore an increase in references to the locks. However, it occurs infrequently, particularly when compared to the longer periods of self-scheduling.

5.2. Varying Cache Size

The benefits of increasing cache size on miss ratio for uniprocessor programs are well known. Numerous trace-driven studies over a variety of workloads have all confirmed that miss ratio drops as cache size is increased, but that the improvements diminish for large caches [Agar88a, Alex86, Good87, Hill87, Przy88, Smit87].

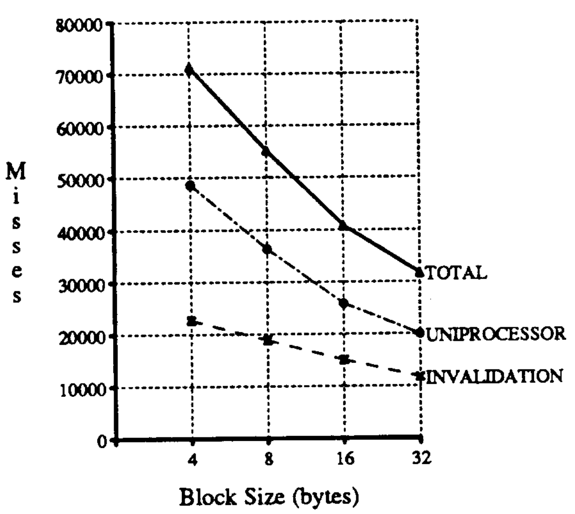
Shared programs do not experience the same miss ratio benefits of increasing cache size. While it is true that their uniprocessor-related misses decline with larger caches, their invalidation misses either rise or, at best, remain constant. The combination produces a miss ratio that declines with cache size, but is higher than for comparable uniprocessor programs.

The parallel traces support this analysis. For all traces, miss ratios decline with increasing cache size (see Figure 5-9), and total miss ratios are higher than their uniprocessor components (see Table 5-1). The discrepancy increases with cache size, because the uniprocessor miss ratio declines more steeply. The exact figures range from 1.02 to 2.2 higher for SPICE, 1.1 to 2.5 higher for VERIFY, 1.1 to 4.7 higher for CELL and 1.7 to 15.4 higher for TOPOPT, as cache size increases from 16K bytes to 512K bytes. (See comparative curves for different types of misses for two of the traces, TOPOPT and VERIFY, in Figures 5-10 and 5-11, respectively.) These results indicate that the benefits of increasing cache size are less pronounced for parallel programs than uniprocessor programs.

The reason is the presence of invalidation misses. The number of invalidation misses is inversely proportional to the number of block replacements. At small cache sizes, the number of block replacements is relatively high. If we assume that shared data are replaced at the same rate as private data or instructions, then a proportion of shared data blocks, equivalent to the percentage of blocks replaced, will be eliminated from the cache. They therefore cannot be invalidated and, consequently, will not incur invalidation misses.[4] As cache size increases, the

[3] The write run results in [Egge88] and tracking cache block behavior with our simulator corroborate the difference in behavior between the two groups of traces.

[4] They will, however, like all data and instructions, incur replacement or capacity misses [Hill87]. However, this is a consequence of the smaller cache size, rather than the type of data (shared), and will occur for *all* data and instructions. As caches get larger, some of the capacity misses become invalidation misses. No matter which category they fall in, i.e., no matter what the cache size, they still contribute to the miss ra-



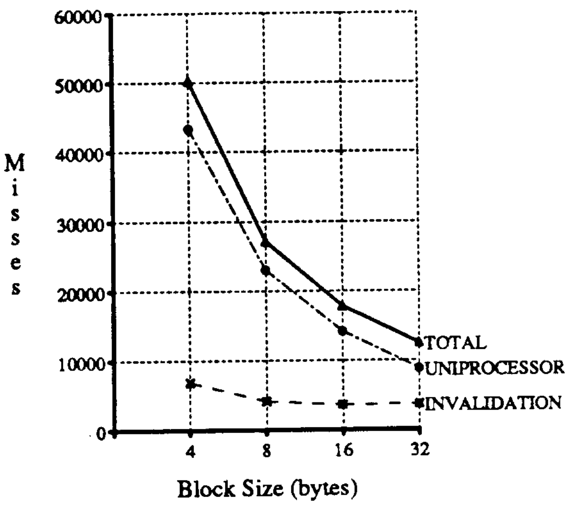
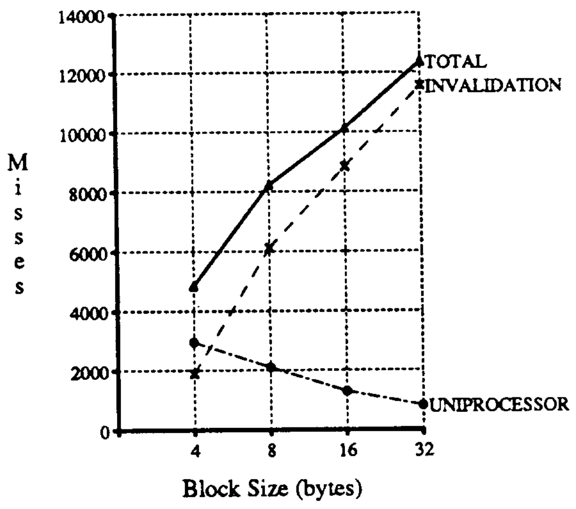


Figure 5-5: Classification of Misses for CELL

Figure 5-6: Classification of Misses for SPICE

The memory reference pattern of shared data in both CELL and SPICE is one of per-processor-locality. Therefore invalidation and uniprocessor misses both decline, producing miss ratio curves that are similar to uniprocessor programs.



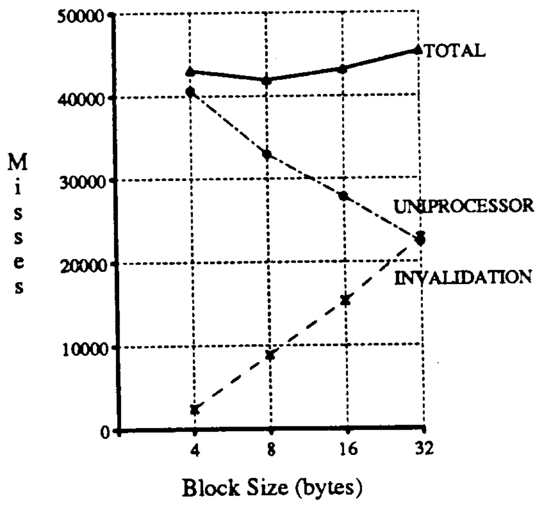


Figure 5-7: Classification of Misses for TOPOPT

TOPOPT is the trace in which the invalidation misses had the most effect and for two reasons: first, it exhibited the most intra-block fine-grain-sharing; and, second, the uniprocessor miss ratio was low, because the working set fit into the 128K byte cache.

Figure 5-8: Classification of Misses for VERIFY

Although VERIFY's memory reference pattern was one of fine-grain-sharing, the uniprocessor misses were proportionately high, particularly at small block sizes. Therefore the miss ratio at first declined, then rose. (Since the individual processor figures for VERIFY were widely skewed, the ratio of invalidation misses to total misses does not match the geometric means in Figure 5-4.)

tio.

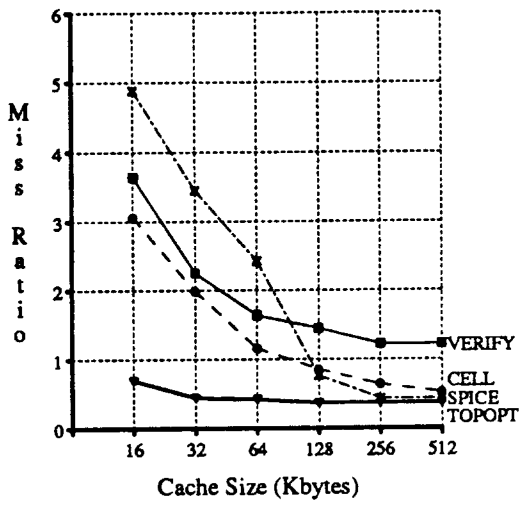


Figure 5-9: Miss Ratio
Increasing the cache size causes the miss ratio for parallel programs to decline. However, the miss ratio is higher than for uniprocessor programs, because of invalidation misses. (All cache size graphs assume a 32 byte block.)

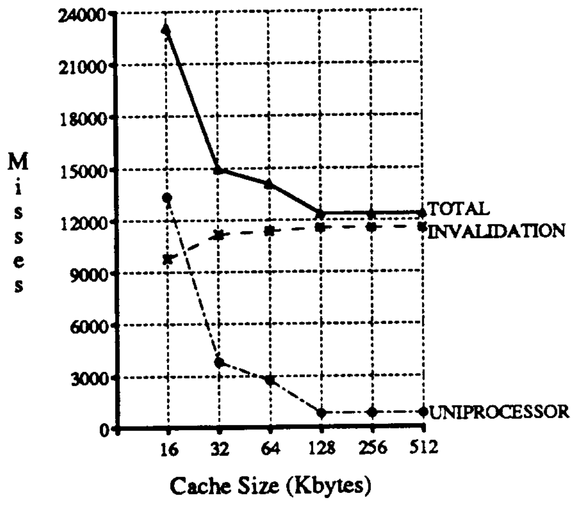


Figure 5-10: Classification of Misses for TOPOPT
Of all the traces TOPOPT has the most fine-grain-sharing within the cache block. The effects of that intra-block contention are manifested by the distortion to the miss ratio caused by the number of invalidation misses. (Note that the scale of the y-axis is roughly one_sixth that of VERIFY in Figure 5-11.)

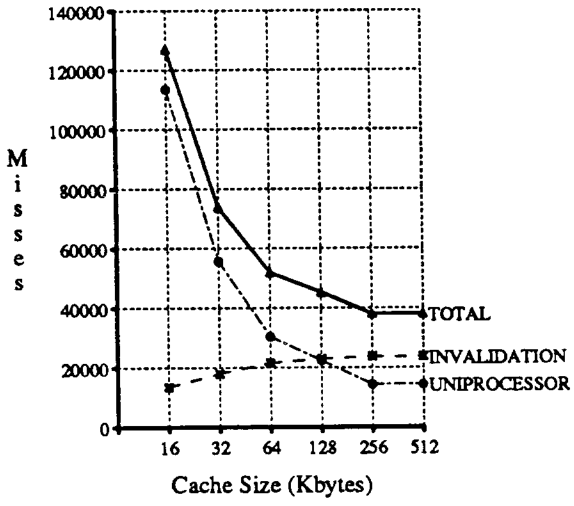


Figure 5-11: Classification of Misses for VERIFY
Miss trends for VERIFY typify the effect of sharing with increasing cache size. The presence of invalidation misses causes the total number of misses (and hence the miss ratio) to be higher than for a uniprocessor program. Their rise, as cache size increases, widens the gap between the total and uniprocessor miss ratio. CELL and SPICE have similar curves, although fewer misses in absolute and fewer invalidation misses proportionately. Their lower figures are due to per-processor-locality.

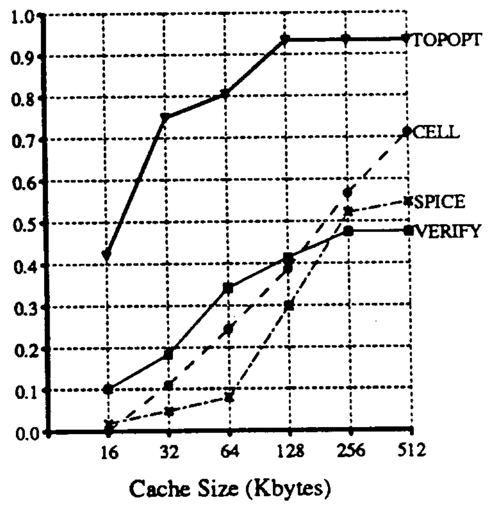


Figure 5-12: Ratio of Invalidation Misses to Total Misses
The ratio of invalidation misses to total misses increases with increasing cache size. The rise is much steeper than with increasing block size for the traces with per-processor-locality, CELL and SPICE. (The numbers are the geometric mean of invalidation to total misses, across all processors. Since the individual processor figures for VERIFY were widely skewed, their geometric mean does not match the absolute misses in Figure 5-11.)

Percentage Change in Miss Ratio						
Trace	Miss Ratio Type	Cache Size Spread (in bytes)				
		16K-32K	32K-64K	64K-128K	128K-256K	256K-512K
CELL	Total	-35.193	-41.487	-26.267	-25.147	-18.225
	Uniprocessor	**-39.832**	**-50.511**	**-41.174**	**-49.245**	**-54.467**
SPICE	Total	-29.444	-29.770	-68.364	-42.592	-1.324
	Uniprocessor	**-31.566**	**-32.189**	**-75.871**	**-61.098**	**-7.120**
TOPOPT	Total	-35.647	-5.812	-12.197	0.000	0.000
	Uniprocessor	**-72.467**	**-26.663**	**-70.330**	**0.000**	**0.000**
VERIFY	Total	-38.203	-27.167	-11.596	-15.354	0.000
	Uniprocessor	**-47.062**	**-43.031**	**-25.338**	**-35.544**	**0.000**

Table 5-1: Percentage Change in Miss Ratio with Increasing Cache Size

This table contains the incremental miss ratio decline as cache size increases. Note that for all programs and all cache sizes, the uniprocessor miss ratios declined more steeply (bold) than total miss ratios. This indicates that uniprocessor programs obtain a greater benefit from increasing cache size.

percentage of block replacements drops. Shared data tend to remain in the cache for a longer period of time, has more opportunity to be invalidated, and, consequently, rereferenced via invalidation misses. The number of invalidation misses should be higher with each successively larger cache, approximately by the percentage decrease in block replacements.[5] For very large cache sizes, in which the program's working set fits into the cache, the incremental number of block replacements is negligible, and the invalidations will tend to level off. Again, the traces confirm the analysis. For all traces, the number of invalidation misses rises with increasing cache size. The increase is most pronounced at smaller cache sizes, at which the change in block replacements is also greater (table not shown).

As was true with the block size figures, the proportion of invalidation misses becomes larger as cache size increases (see Figure 5-12). For the traces with per-processor-locality (CELL and SPICE), the effect of the invalidation misses is more pronounced with larger cache sizes than with larger block sizes. Invalidation misses cause the greatest perturbation for TOPOPT, the trace with the most fine-grain-sharing. Here the proportion of invalidation misses to total misses ranges from 42 to 93 percent, as cache size increases from 16K to 512K bytes. This causes the total miss ratio to be 1.7 to *15.4* times greater than its uniprocessor component (again, see Figure 5-10).

The working sets of TOPOPT and VERIFY fit into the larger sized caches. Once the caches were filled, the number of uniprocessor misses remained constant. The invalidation misses also remained constant, because there was no more block replacement effect.

6. The Effect of Sharing on Bus Utilization

The critical system bottleneck in a single-bus, shared memory multiprocessor is the bandwidth of the system bus. Relatively few processors can be attached to the bus, unless caching is used to reduce their bandwidth requirements. For a single-bus multiprocessor, the most important consideration for cache organization is how well it limits bus utilization. As was implied by the higher miss ratios in the last section, the bandwidth requirements are greater in parallel programs than uniprocessor programs because of the sharing traffic. With large caches and large block sizes, we expect the sharing traffic to dominate the bandwidth and, consequently, dictate the number of processors that can be effectively attached to the bus.

[5] The rising cost of sharing with larger caches is a problem usually associated with write-broadcast coherency protocols (see [Egge]; it is interesting that the problem occurs with write-invalidate as well. (Under write-broadcast protocols, all processors write through for shared data; therefore caches always contain the current value for any shared address.)

6.1. Varying Block Size

Several uniprocessor studies [Przy88, Smit87] have shown that, up to a certain size, increasing the block size can improve bus performance. Two factors are responsible for the improvement: (1) a decreasing miss ratio as block size increases; and (2) a decreasing mean memory delay per memory reference, because memory latency is a proportionately smaller overhead at larger block sizes. The improvement occurs when both factors are substantial enough to counterbalance the increase in bus traffic that also accompanies larger block sizes [Good87, Smit87]. The breakeven point is found where the decrease in the miss ratio is offset by the increase in the average number of cycles per transfer. Results in any bus utilization study are highly dependent on the cycle assumptions for both memory accessing and bus transfer overhead. But, for caches of the size under study, i.e., 128K bytes, and up to 32 byte blocks, at least one study has shown that the average memory access time declines with increasing block size [Agar88a].

Sharing alters bus utilization in two ways. First, invalidation signals and invalidation misses are sources of additional bus traffic, since they do not exist in uniprocessor systems. They cause bus utilization to be higher in parallel programs. Second, the slope of the bus utilization curve is determined by the memory reference pattern to shared data. Programs with fine-grain-sharing have miss ratios that increase rather than decrease with block size. Therefore their miss ratios compound the increase in bus traffic caused by the larger transfer unit, and bus utilization increases. For programs that exhibit per-processor-locality, miss ratios decline, and the marginal miss ratios (as block size increases) are comparable to those for uniprocessor programs. In this case, bus utilization could proceed in either direction, depending on whether the change in the miss ratio is great enough to offset the increase in the average number of cycles per transfer.

The traces under study reflected these effects. For all traces, bus utilization was higher than its uniprocessor component.[6] (The ranges for the individual traces are: 1.9 to 2.2 higher for CELL, 1.7 to 1.8 for SPICE, 2.3 to 17.7 for TOPOPT and 1.3 to 2.6 for VERIFY.) For the most part, the per-processor-locality of CELL and SPICE produced a miss ratio that decreased enough to offset the increase in the average cycles per transfer. The result was bus utilization that decreased over most of the block size spectrum (4 bytes to 32 bytes) (see Figure 6-1). TOPOPT and VERIFY are programs with a fair amount of fine-grain-sharing. The resulting increase in their miss ratios (or a very small decrease for some block sizes for VERIFY), plus the normal rise in the average number of cycles per transfer, produced the increasing bus utilization figures (again, see Figure 6-1). ([Cher88] has also noticed the effect of fine-grain-sharing on bus traffic. In simulations done on a four-processor multiprocessor, in which management of the 256K byte cache was done under software control, two traces exhibited an increase in bus operations per reference, as block size was increased.)

For three of the traces sharing-related bus overhead comprised a substantial portion of total bus cycles across all block sizes but one (4 bytes for VERIFY). For the fourth trace, TOPOPT, they totally dominated bus activity. The ranges are 56 to 94 percent for TOPOPT, 45 to 61 percent for VERIFY (excluding the exception), 47 to 54 percent for CELL, and 40 to 44 for SPICE (see Figure 6-2). (The proportions are higher than the proportions of invalidation misses to total misses, because the cycle figures include cycles for invalidation signals as well as invalidation misses.) The curves clearly show that for 128K byte caches bus bandwidth requirements are determined by the sharing traffic.

Because sharing-related bus overhead is such a large proportion of total bus cycles, its behavior as block size increases can dictate the bus utilization trend. TOPOPT is the most extreme example. It has the largest proportion of sharing cycles, and their rate of increase is steep (see Figure 6-3). Although the uniprocessor cycles decline with increasing block size, their rate of decline is more moderate, and they are a very small proportion of total bus cycles. Therefore TOPOPT's total bus utilization curve rises. The other three traces exhibit similar effects, although for the programs with per-processor-locality, the sharing cycle trends pull total bus utilization downwards. (An example appears in Figure 6-4.)

6.2. Varying Cache Size

Increasing cache size is an important design technique for improving bus utilization. With the exception of enlarging either an extremely small block size or a very large cache,[7] it provides a larger performance boost than

[6] Uniprocessor bus utilization is determined by excluding the cycles used for invalidation signals and invalidation misses.

[7] Doubling a very small block, say 4 bytes in size, produces a good performance improvement; increasing an already large cache provides little additional benefit.

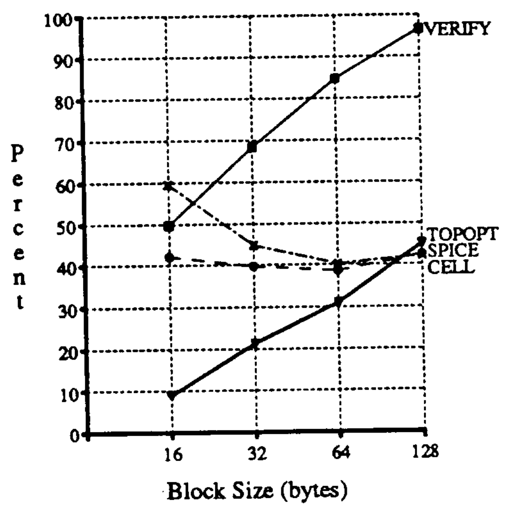


Figure 6-1: Effect of Block Size on Bus Utilization
Bus utilization is calculated as the number of cycles during which a bus operation took place, divided by the total cycles in the simulation. The bus cycles include cycles for the overhead of bus operations, in addition to those counted in bus traffic figures. The per-processor-locality of CELL and SPICE produced the declining or flattened bus utilization curves; the fine-grain-sharing of TOPOPT and VERIFY exacerbated their already rising average cycles per transfer, resulting in increasing bus utilization.

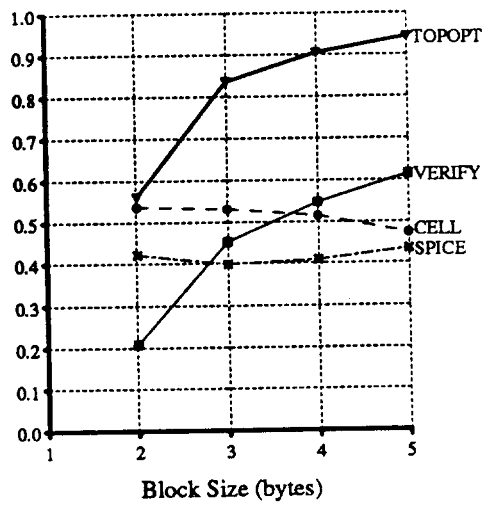


Figure 6-2: Ratio of Shared to Total Bus Cycles
The cycles needed for invalidations and invalidation misses were a substantial portion of or completely dominated total bus cycles, over most block sizes. This indicates that efforts to reduce bus bandwidth demands should concentrate on the sharing-related traffic. (All block size graphs are for a 128K byte cache.)

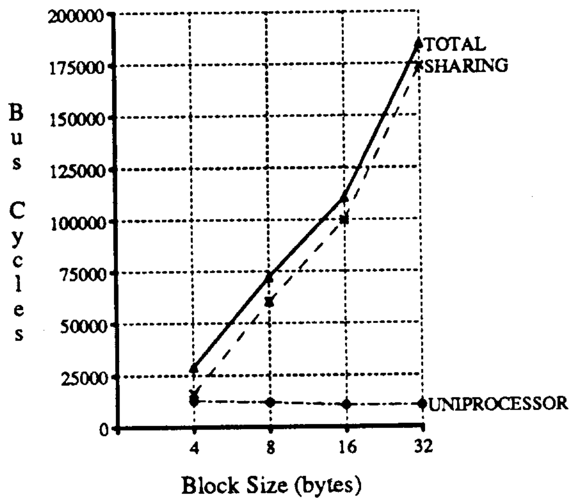


Figure 6-3: Classification of Bus Cycles for TOPOPT
The proportion of sharing-related bus cycles for TOPOPT ranged from 56 to 94 percent. Because they were such a majority of total bus cycles, their behavior forced bus utilization to follow suit.

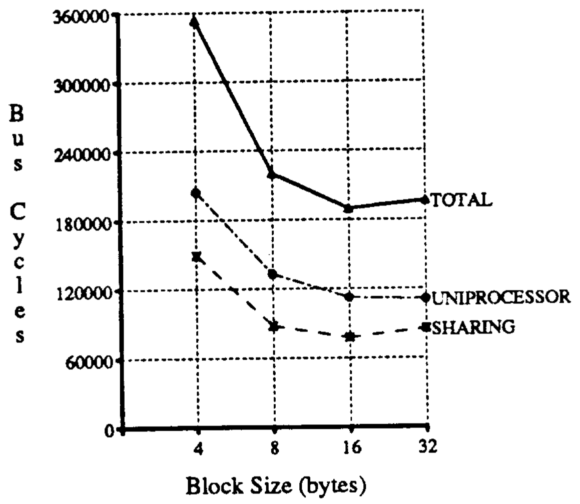


Figure 6-4: Classification of Bus Cycles for SPICE
SPICE was typical of programs with per processor locality. The decline in sharing-related bus cycles reinforced a corresponding drop in uniprocessor bus cycles, producing a falling bus utilization curve.

increasing either block size or set associativity [Przy88]. There are two factors that contribute to the greater improvement. First, the miss ratio is more responsive to cache size than to increases in the other two parameters. Second, the longer cache access time of larger caches is less severe a penalty to effective access time than the cost of increasing either of the other parameters, particularly, the increase in bus traffic with a larger block size.

All of the traces exhibit the expected falling bus utilization (see Figure 6-5), and for the usual reason: a miss ratio that declines with increasing cache size (see Figure 5-9). The decline is particularly sharp for the programs with per-processor-locality, CELL and SPICE, and their bus utilization curves reflect the drop. The decrease in the miss ratios did not translate directly into a comparable change in bus utilization, because of a rise in both the number of cache-to-cache transfers and the number of invalidations. Under the Berkeley Ownership cache coherency protocol, cache-to-cache transfers are the mechanism for satisfying processor reads to write shared data. As cache size increases, the number of cached write shared blocks also increases, and therefore the number of cache-to-cache transfers goes up. In the simulator's memory system (and the implementation of SPUR as well), cache-to-cache transfers require more cycles than memory transfers. The shift to the more expensive type of data transfer, as cache size increases, flattens the bus utilization curve. A more optimized cache controller implementation or a slower memory would have produced a steeper drop. (The effect of invalidation signals on bus utilization was discussed in Section 6.1.)

The proportion of sharing-related bus cycles to total bus cycles is depicted in Figure 6-7. For all traces, cycles due to invalidation signals and invalidation misses rise sharply with cache size. For large caches (128K bytes and up), they dominate bus bandwidth demands. (Results in [Site88] also indicate a rising proportion of sharing traffic with increasing cache size, although the sharing traffic does not dominate, even with one megabyte caches. Traces for their study are concatenated samples of memory references of CAD and expert systems applications running under MACH, in a two processor multiprocessor.)

7. Concluding Discussion

7.1. Implications for Cache and Bus Designers

Cache design is an optimization problem. Its goal is to minimize effective access time by changing various cache parameters. The difficulty is that these parameters alter cache performance in conflicting ways. For example, increasing cache size decreases the miss ratio, but at the expense of a larger cache access time. Increasing the block size also decreases the miss ratio, but only until the pollution point is reached. After that, larger blocks sizes produce a rising miss ratio. An additional drawback of all block size increases is the accompanying increase in the amount of data that are transferred in a single bus operation. The increase in the average cycles per transfer can cause bus utilization to rise even before the pollution point is reached.

Parallel programs, running under write-invalidate coherency protocols, complicate cache design by introducing another factor into the optimization problem: invalidation misses. The studies in this paper have shown that invalidation misses increase miss ratios, sometimes enough to reverse declining miss ratio curves produced by the other factors. For example, as cache size increases, the number of invalidation misses also increases. Invalidation misses occur in smaller caches as well, but in the guise of replacement misses. With larger caches, some replacement misses for instructions and private data are eliminated; those to shared data can only be converted to invalidation misses. The result is a miss ratio that, for most of the traces, ranges from 2.2 to 4.7 times greater than its uniprocessor component, and 15 times greater in the worst case.

Sharing references also derive less benefit than uniprocessor references from a larger block size. Increasing block size either increases the number of invalidation misses or decreases them at a rate that is less than for uniprocessor misses. The type of miss behavior depends on whether the program exhibits per-processor-locality or fine-grain-sharing. In the former, invalidation misses decline with block size, and produce a miss ratio that is higher than comparable uniprocessor programs. When there is fine-grain-sharing, the number of invalidation misses rises dramatically with block size. The increase is enough to reverse the declining miss ratio that occurs with uniprocessor programs in caches of this size (128K bytes).

In all cases the miss ratio is higher than in uniprocessor caches. Therefore designers must use larger or more

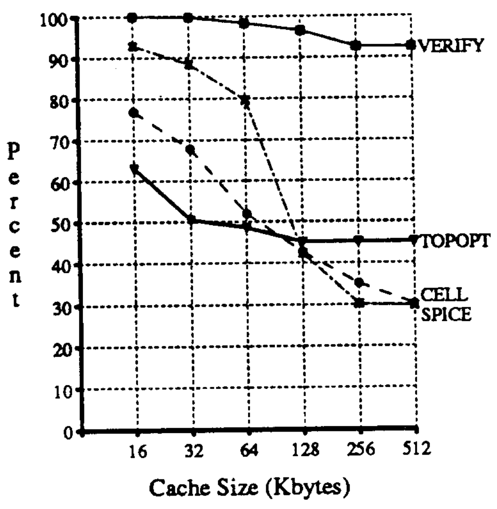


Figure 6-5: Effect of Cache Size on Bus Utilization
As is true for uniprocessor programs, bus utilization for the parallel programs declined with increasing cache size. The benefit of enlarging the cache was greatest for the two programs with per-processor-locality, CELL and SPICE. (All cache size graphs assume a 32 byte block.)

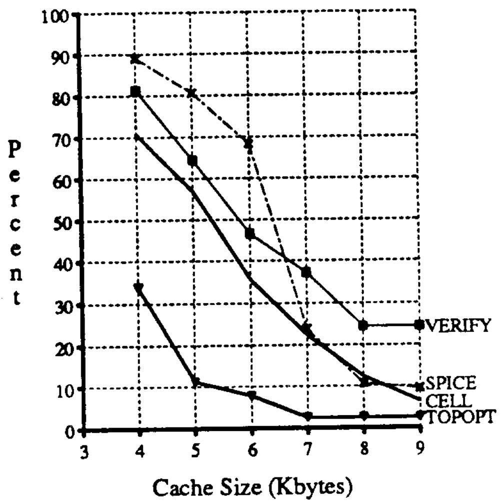


Figure 6-6: Uniprocessor Bus Utilization
Bus utilization was higher than its uniprocessor component. The ranges for the individual traces are: 1.04 to 3.1 for SPICE, 1.2 to 3.7 for VERIFY, 1.1 to 5.1 higher for CELL and 1.9 to 17.7 for TOPOPT.

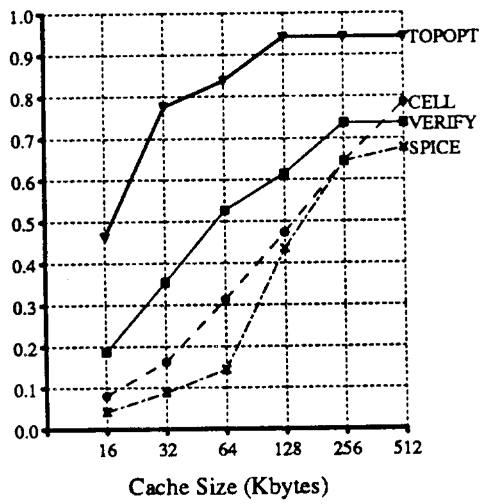


Figure 6-7: Ratio of Sharing Bus Cycles to Total Bus Cycles.

The proportion of sharing-related bus cycles to total cycles rises sharply with increasing cache size. For large caches, they comprise the largest component of bus utilization.

complex caches[8] to obtain the same performance in multiprocessors; even then, they might not be able to obtain this level, because some costs of sharing are inherent in the algorithm, and are unaffected by cache design changes.

[8] For example, greater associativity, multi-level caches, etc.

The choice of block size is dependent on the anticipated workload mix, in particular the balance between programs that exhibit per-processor-locality or fine-grain-sharing.

The additional cache misses, of course, increase bus utilization. Moreover, sharing under write-invalidate protocols introduces another type of bus operation, the invalidation signal, which further increases bus utilization. Bus utilization was 1.04 to 17.7 times higher with increasing cache size, and 1.3 to 17.7 times higher with increasing block size. Even for the small-scale multiprocessors studied, the bus was well utilized, with typical bus utilization figures ranging from 30 to 70 percent. The implication for bus design is a need for additional speed in order to support a larger scale, single-bus multiprocessor. Fast bus architectures (for example, split transaction bus protocols) and faster bus implementations (for example, bipolar and optics) are even more important in multiprocessors than uniprocessor systems.

7.2. Implications for Parallel Software Writers

The performance of parallel programs may be improved by a variety of software techniques for restructuring shared data. The techniques can be used by applications programmers and operating system designers, or compiler writers.

We have seen that shared references were responsible for considerable overhead in the cache and bus performance. Invalidation misses comprised a substantial proportion of total misses for moderate block sizes (16 and 32 bytes, and even smaller for some traces) and large cache sizes (128K bytes and up). For all block sizes and large caches, sharing-related bus traffic accounted for the majority of total bus cycles.

As multiprocessor caches continue to increase in size, uniprocessor misses will become a decreasingly smaller proportion of total traffic; and a correspondingly larger proportion will be due to sharing. Adding processors to such systems will increase sharing traffic in absolute terms. The bottom line is that it is the sharing traffic that will determine bus bandwidth demands, and will eventually limit the scale of the single bus multiprocessor by creating a bus bottleneck.

Given that multiprocessors already have large caches, the bottleneck can only be postponed by improving the cache and bus performance for the shared data portions of the parallel programs. One observation of the programs studied here is that their amount of locality within the cache block largely determined the coherency cost, and therefore their miss ratios and bus utilization. Good per-processor-locality reduces the number of invalidation signals and invalidation misses, which lowers these metrics. On the other hand, fine-grain-sharing, i.e., poor per-processor-locality, has the opposite effect. Thus better memory organization for shared data can improve program execution. If shared data accessed by different processors are allocated to separate cache blocks, then programs with fine-grain-sharing should obtain lower coherency costs, and an improvement in overall performance.

Better data alignment can occur by at least two different means. The first is through explicit programmer specification of the organization of shared data and runtime support for its allocation in shared memory on cache block boundaries. Currently, shared variables may be dynamically allocated by a system runtime routine that makes the data visible to all processes. In our proposed data alignment scheme, the programmer would be responsible for grouping those shared variables that are used by different processors via separate system calls. The routine itself would allocate the shared data in each invocation on cache block boundaries, padding out the block when necessary. The advantage of this approach is the simplicity of its implementation; it is a very straightforward technique for reducing bus traffic under software control. Its disadvantages are that it places the responsibility for optimal runtime memory usage of shared variables entirely on the programmer and requires that the runtime system be aware of the cache block size.

A second method for improving the memory organization of shared data addresses the issue of programmer responsibility, but at an extremely high cost in implementation complexity. The approach involves the automatic compiler detection and consequent memory allocation of per processor shared variables. The techniques involved are similar to those used for the lifetime analysis of objects to reduce garbage collection overhead [Rugg88]. The problem is difficult, because the compiler must analyze references to pointers rather than discrete variables. The set of objects that are linked by pointers may be arbitrarily complex, and it is difficult to detect their dynamic relationship. A precise solution is intractable; in practice, the technique could probably only be used for a subset of easily recognizable structures. Moreover, a compile time analysis produces a conservative, worst-case estimate that may not reflect the actual execution behavior of the program. This can lead to wasted memory and additional bus traffic, because small objects would be allocated to larger cache block units. At this point, automatic compiler detection of shared data that is actually used by a single processor is an open research question; it is not clear that freeing the programmer of the responsibility for optimally allocating shared data is worth the complexity of the automatic solution. The programmer-initiated solution should be tried first to determine whether it can produce the performance benefits of good per-processor-locality.

Acknowledgements.

We wish to acknowledge the efforts of several others who contributed to the work in this paper. Dave Ditzel, Garth Gibson and David Wood were useful sounding boards for the analysis and presentation of many of the results. Fred Douglis, Mark Hill, Jim Larus, Corinna Lee, John Ousterhout and Stuart Sechrest gave valuable comments on an earlier draft. Dominico Ferrari and Yale Patt provided resources for running 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. Dianne DeSousa and her coworkers at ELXSI 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, Digital Equipment Corporation, and California MICRO (in conjunction with Texas Instruments, Xerox, Honeywell, and Philips/Signetics).

References

[Agar88a] 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.

[Agar88b] A. Agarwal and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH", *Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 16, 1 (1988), 215-225.

[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", *IEEE International Conference on Computer-Aided Design*, Santa Clara, CA (November 1986), 30-33.

[Cher88] D. F. Cheriton, A. Gupta, P. D. Boyle and H. A. Goosen, "The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation", *Proceedings 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 410-421.

[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 15th Annual International Symposium on Computer Architecture*, Honolulu HA (May 1988), 373-383.

[Egge] S. J. Eggers and R. H. Katz, Evaluation the Performance of Four Snooping Cache Coherency Protocols, submitted for publication.

[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.

[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).

[Katz85] R. Katz, S. Eggers, D. Wood, C. L. Perkins and R. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings 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.

[Przy88] S. Przybylski, M. Horowitz and J. Hennessy, "Performance Tradeoffs in Cache Design", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (May 1988), 290-298.

[Rugg88] C. Ruggieri and T. P. Murtagh, "Lifetime Analysis of Dynamically Allocated Objects", *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, San Diego (January 1988), 285-293.

[Site88] R. L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM", *Proceedings 15th Annual International Symposium on Computer Architecture*, Honolulu, HA (May 1988), 186-195 .

[Smit85] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice", *Proceedings of 12th Annual International Symposium of Computer Architecture*, 13, 3 (June 1985), 64-73.

[Smit87] A. J. Smith, "Line (Block) Size Choice for CPU Caches", *IEEE Trans. on Computers*, C-36, 9 (September 1987).

[Wood86] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz and D. A. Patterson, "An In-Cache Address Translation Mechanism", *13th Annual International Symposium on Computer Architecture*, Tokyo, Japan (June 1986), 358-365.

[Wood87] D. A. Wood, S. J. Eggers and G. A. Gibson, "SPUR Memory System Architecture", Technical Report No. UCB/Computer Science Dpt./87/394, University of California, Berkeley (December 1987).