Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations.

Tor E. Jeremiassen and Susan J. Eggers
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195
(206) 543-9349 (206) 543-2118
{torerik,eggers}@cs.washington.edu

Abstract

We have developed compiler algorithms that analyze coarse-grained, explicitly parallel programs and restructure their shared data to minimize the number of false sharing misses. The algorithms analyze the per-process data accesses to shared data, use this information to pinpoint the data structures that are prone to false sharing and choose an appropriate transformation to reduce it.

The algorithms eliminated an average (across the entire workload) of 64% of false sharing misses, and in two programs more than 90%. However, how well the reduction in false sharing misses translated into improved execution time depended heavily on the memory subsystem architecture and previous programmer efforts to optimize for locality. On a multiprocessor with a large cache configuration and high cache miss penalty, the transformations improved the execution time of programmer-unoptimized applications by as much as 60%. However, on programs where previous programmer efforts to improve data locality had reduced the original amount of false sharing, and on a multiprocessor with a small cache configuration and cache miss penalty, the gains were more modest.

1 Introduction

On bus-based, shared memory multiprocessors, much of the "unnecessary" bus traffic, i.e., that which could be eliminated with better processor locality [AG88], is coherency overhead caused by false sharing [TLH90, EJ91]. False sharing occurs when multiple processors access (both read and write) different words in the same cache block. Although they do not actually share data, they incur its costs, because coherency operations are often cache block-based. In a write-invalidate coherency protocol the overhead of false sharing takes the form of additional invalidations when a processor updates data and additional invalidation misses when other processors reread (different) data that resides in the invalidated cache block. In some coarse-grain, explicitly parallel applications, misses due to false sharing comprise between 40% and 90% of all cache misses (over block sizes ranging from 8 to 256 bytes) [EJ91].

False sharing is caused by a mismatch between the memory layout of write-shared data and the cross-processor memory reference pattern to it. Manually changing the placement of this data to better conform to the memory reference pattern (based on profiles derived from trace-driven simulations) can reduce false sharing misses by up to 75% [TLH90, EJ91]. However, manual restructuring requires that the programmer pinpoint the data structures that suffer from false sharing in a particular memory (cache) architecture. (Simulation profiles are generally not available during the development cycle.) To identify these data structures it is necessary to know their layout in memory, the cross-processor memory reference pattern to them, as well as details of the memory architecture and coherency protocol. All this is hard to determine; knowledge of

how each data object is shared is often non-intuitive, and each application must be tailored to the particular memory architecture of the system on which it is running.

For these reasons we have automated the elimination of false sharing. We have developed and incorporated into the Parafrase-2 [PGH+89] source-to-source restructurer a series of compiler-directed algorithms [Anoa, Anob] and a suite of transformations that restructure shared data at compile time. Our algorithms analyze explicitly parallel programs, producing information about their cross-processor memory reference patterns that identifies data structures susceptible to false sharing and then chooses appropriate transformations to eliminate it.

This paper reports the results of applying the static analysis to a set of explicitly parallel programs. While no single algorithm nor transformation eliminated false sharing in all applications, when used together, they were more successful than the programmer-directed efforts. They eliminated on average (across all block sizes and for the entire workload) 64% of the false sharing misses, and up to 90% for the programs that had previously been manually transformed. Although some of the transformations increase the data size of the programs, the negative impact on spatial locality was in all but one program less than the benefits achieved by eliminating false sharing misses. In fact, for two applications spatial locality improved.

The effect of reducing false sharing on execution time was extremely dependent on the memory subsystem architecture. When the local cache, the cache block size (coherency unit) and the miss penalty were large, as they are on the KSR1, the transformations improved execution time by as much as 60%. However, with older technology, such as that used on the Sequent Symmetry, virtually no execution time impact of the transformations was observed. Three factors were responsible. First, the false sharing problem was diminished, because less sharing occurred within the smaller cache blocks and shared data in smaller caches was often replaced before it could be invalidated. Second, memory latency, and thus the cost of each coherency operation induced by false sharing was lower. Third, the small cache size increased the negative impact on capacity misses of transformations that increase the data size, thus offsetting the positive effects from reducing (a small amount of) false sharing.

In the next section we describe our model of parallel programming. Section 3 presents a brief overview of the compile time analysis performed by our system and gives a breakdown of its different levels of accuracy in computing the inter-process reference pattern; section 4 describes the transformations; section 5 describes our workload and methodology; section 6 presents the experimental results; related work is discussed in section 7; and section 8 concludes.

2 Model of Parallel Programming

Our analysis and transformations are aimed at coarse-grained, explicitly parallel programs for shared memory multiprocessors, written in traditional high level languages such as C, that have been enriched with process creation and synchronization primitives. These programs are typical of those that currently execute on small to medium scale, bus-based multiprocessors, both commercially (e.g., Sequent Symmetry [LT88] and the KSR1 [KSR92]) and in research environments (e.g., DASH [LLG+92]).

```
private int pid;
shared barrier_t MyBarr1, MyBarr2, MyBarr3;
shared int NumProcs;
⋮
for ( pid = 1; pid < NumProcs; pid++) {
    if (fork() == 0) {
        Work();
        exit(0);
    }
}
Work();
⋮
```

(a)

```
Work() {
    while (converged != 0) {
        SubPart1(pid);
        Wait_Barrier(&MyBarr1);
        SubPart2(pid);
        Wait_Barrier(&MyBarr2);
        if (pid == 1)
            converged = TestConverged();
        Wait_Barrier(&MyBarr3);
    }
}
```

(b)

```
SubPart1(proc)
int proc;
⋮
cell1 = cell_num1[proc];
cell2 = cell_num2[proc];
⋮
```

(c)

Figure 1: Example program segment that illustrates the use of a process differentiating variable in process creation (a), per-process control flow (b), and shared data access (c).

The granularity of parallelism in these programs is very coarse, on the level of an entire process. In our model the number of processes equals the number of processors and processes do not migrate. The programs conform to an SPMD model of parallel programming: the processes all have identical code, but they need not take the same paths through the program. They may or may not access different data.

Processes are created explicitly, e.g., using a *fork()* system call (illustrated in Figure 1). They are typically spawned in a loop that iterates over the number of processes; each value of the induction variable (e.g., *pid*) is stored in a private (to each process) variable as a de facto process identifier. We call this a process differentiating variable.

Process synchronization is performed using global barriers. When the control flow of a process reaches a barrier, it must wait until all participating processes also reach it. Barriers are used in shared memory multiprocessors as a (relatively) inexpensive mechanism to enforce large sets of cross-process data dependences that otherwise would have to be enforced by a large number of locks. Locks are only used to enforce mutual exclusion, i.e., they serialize access to critical sections. In our false sharing analysis we treat accesses to locks just like accesses to the shared data they protect.

3 Compile-time Analysis

In order to determine which data structures are susceptible to false sharing, where locality may be improved, and which transformations to apply at compile time, we analyze a program and compute an approximation of the memory access pattern of each of its processes. The compiler analysis involves three separate stages. The first determines which sections of code each process executes by computing its control flow graph [Anoa]. The second performs non-concurrency analysis [MR93] by examining the barrier synchronization pattern of the program, delineating the phases that cannot execute in parallel and computing the flow of control between them [Anob]. The third stage performs an enhanced interprocedural, flow-insensitive, summary

side-effect analysis [Bar78, Ban79, Mye81, CK88b] and static profiling [Wal91] on a per-process basis (based on the control flow determined in stage one) for each phase (determined in stage two). In this paper we only discuss those aspects of the static analysis that clarify how it effects the detection of false sharing.

Per-process references to shared data occur either as a result of the processes executing different code (thus accessing different elements of shared data) or by the implicit partitioning of arrays across the processes when they execute the same code. Per-process control flow analysis (stage 1) detects the first case, and summary side-effect analysis and process differentiating variables[1] (stage 3) help detect the second. The side-effect analysis represents the sections of each array that each process accesses using bounded regular section descriptors[2] to describe the index expressions [HK91]. When a regular section descriptor indicates that a process differentiating variable is used in the index expressions for array accesses, we test whether the descriptor identifies disjoint sections of the array for different values of the variable. The array is implicitly partitioned across processes if the sections are disjoint. The per-process control flow analysis, on the other hand, identifies control statements where the control flow of different processes diverges, and uses this information to compute a separate control flow graph for each process. Analyzing shared arrays and structures that are indexed by process differentiating variables, and applying the side effect analysis to the separate control flow graphs yields the sections of shared data that each process reads and writes.

There are two problems with using traditional summary side-effect analysis. First, using a single regular section descriptor to represent a process's array accesses results in a loss of accuracy (in describing the index expressions) when the array is accessed differently in different parts of the program. To improve the accuracy we allow multiple regular section descriptors to exist for each array. Instead of eagerly merging descriptors [CK88a, HK91] (and losing information in the process), we only merge descriptors when very little or no information will be lost, or when the number of descriptors for a single array exceeds some small preset limit. (None of the arrays used in our benchmarks required more than 10 descriptors).

The second problem with summary side-effect analysis is that it does not differentiate between accesses that occur inside and outside loops, thereby failing to discern the dominant (most frequently executed) memory reference pattern. To counter this, we use static profiling information [Wal91] to produce a weighting of the side-effects with respect to estimated execution frequency. This allows us to disregard data structures that are accessed infrequently, or are read frequently but seldom written. Using both static profiling information and multiple regular section descriptors enables us to get a much better picture of the predominant sharing patterns in the programs, and consequently to make more and better data restructuring decisions.

The non-concurrency analysis (stage 2) uses barrier synchronization points to determine which portions of a program can execute in parallel and which cannot. It therefore can detect the memory access pattern of distinct phases of a program (between barriers), and, more importantly, when the pattern shifts. When coupled with static profiling, it can determine the dominant sharing pattern in the program and restructure

[1] As mentioned in section 2, process differentiating variables are private variables that have values that vary across the processes and are invariant throughout the lifetime of the processes. *pid* in Figure 1 is an example.

[2] A regular section descriptor is a vector of subscript positions in which each element describes the accessed portion of the array in that dimension. Each element is either a simple, invariant expression of program variables or constants (when the index expression for that dimension does not contain an induction variable), a range (giving simple, invariant expressions for the lower bound, upper bound and stride), or unknown (when the index expressions are too complex or variable).

for that pattern. For example, in one application the non-concurrency analysis revealed that shared structures were accessed on a distinct per-process basis in all parts of the program except during the final convergence testing. It therefore transformed the data by process, according to the dominant usage. Without non-concurrency analysis, the side effect analysis would have discouraged the transformation.

Including all techniques in the source-to-source restructurer had little impact on the overall compile costs. When techniques commonly used in optimizing compilers (such as call and flow graph construction, alias, dependence and loop analysis) were included in our source-to-source restructurer, the execution time of our algorithms made up only 5% (on average) of the total running time.

4 Transformations

In order to eliminate or significantly reduce the number of false sharing misses, data must be restructured so that (1) data that is only, or overwhelmingly, accessed by one processor is grouped together, and (2) write shared data objects with no processor locality [AG88] do not share cache lines. Two transformations, *group and transpose* and *indirection* [Anoa], address item (1); the third, *padding*, is aimed at item (2).

Group and transpose physically groups data together by changing the layout of the data structures in memory. It groups together vectors in which adjacent elements are accessed by different processors and then transposes the group. If each processor's data is less than the cache block size, it may be padded, so that no two processors' data share a cache block. In addition to eliminating false sharing misses, this transformation improves spatial locality.

When it is not possible to physically change the data layout (because, for example, the affected per-process data structure is embedded into the elements of a dynamically allocated list or graph), we can achieve a similar result by using indirection. Indirection allocates data areas of memory for each processor, places shared data into them, and locates the shared data with pointers that replace the values in the original data structures. Unlike group and transpose, indirection has two possible sources of run time overhead: additional space for the pointers, and an additional memory access for each reference to the data.

The third transformation pads data that is falsely shared in the short term but eventually write shared by all processes over time. Padding the data structures increases the data set size, and may therefore increase conflict and capacity misses, and reduce spatial locality when a processor must access the entire shared area. However, judicious use of padding need not have these effects. In order for spatial locality to benefit write-shared data, it must be synonymous with processor locality, i.e., a processor must access the data over a short period of time. If it does not, other processors will invalidate the data before it can be referenced. Therefore we only pad data structures that lack processor locality, i.e., where the possible loss of spatial locality is insignificant relative to the savings in coherency overhead.

Locks are also padded, to the size of the cache block, rather than allocated with the write-shared data they protect. The latter generates coherence traffic when there is contention for the locks: the processor that holds the busy lock loses exclusive ownership of its cache block, because of reads by waiting processors. Its writes to the data cause invalidations, and then invalidation misses when the waiting processors reread

the status of the lock.

Once all stages of the static analysis have been performed, we use a number of heuristics to detect which data structures are susceptible to false sharing and which transformation should be applied to eliminate it [Anob].

5 Methodology

The static analysis and false sharing detection algorithms were implemented as separate passes in Parafrase-2 [PGH+89].

We perform two kinds of experiments to measure the effects of the data transformations on the programs in our workload. False sharing reductions were measured using trace-driven simulation. Each program was traced (both before and after shared data was transformed) on a 20-processor Sequent Symmetry [LT88], using a software tracing tool for parallel programs [EKKL90]. Cache miss rates were analyzed with a multiprocessor simulator that emulates a simple, shared memory architecture. The processors are assumed to be RISC-like, with a 32 KB first level cache and an infinite second level cache[3].

Execution times were measured on two different shared-memory multiprocessors, a 56-processor Kendall Square Research KSR1 [KSR92], and the Sequent. They represent two very different points in the shared-memory multiprocessor design space, in particular with respect to their memory subsystem designs.

The KSR1 memory subsystem has big caches, big cache blocks and long memory latencies. Each processor has a 512 KB first level cache, divided equally between data and instructions. The block size is 2 KB, and each is divided into 64 byte subblocks. The second level cache contains 32 MB and is divided into 16 KB pages, each consisting of 128 subpages of 128 bytes. A subpage is the unit of coherency. There is a 20-24 cycle latency to access the second level caches, which are connected to a 2-level hierarchical ring-based interconnect. There is a 175 cycle latency to access the cache of another processor on the same ring. If the processor is on a different ring, the latency is 600 cycles.

On the KSR1 the lock data structure is large (80 bytes) and aligned on cache block boundaries. To make more accurate, i.e., implementation independent, comparisons with the simulations and the Sequent experiments, we used KSR1 synchronization primitives to implement a smaller version of locks.

The memory subsystem on the Sequent Symmetry is smaller in scale and faster relative to processor speed. Each processor has a 64 KB, 2-way set associative, unified cache, with 16 bytes blocks. The time to service a cache miss is approximately 4 cycles.

On the KSR1 additional run time statistics such as the number of cache misses (both first and second level), page faults, and memory stall time were available from the KSR Performance Monitoring Library.

The workload consisted of six coarse-grained, explicitly parallel programs, all written in C (Table 1). Water, Mp3d and LocusRoute are from the Stanford SPLASH benchmarks [SWG91]. They are all programs in which considerable programming effort has been expended to improve data locality, including eliminating false sharing. They were included in our workload to see if our static analysis could improve upon programmer

[3] Infinite caches can be used to approximate very large (on the order of several megabytes) second level caches [Anoc].

Program	Description	Lines of C
LocusRoute	VLSI standard cell router	6709
Maxflow	maximum flow in a directed graph	810
Mp3d	rarefied fluid flow	1653
Topopt	topological optimization	2206
Pverify	logical verification	2759
Water	n-body molecular dynamics	1451

Table 1: Benchmarks used in our study.

efforts to reduce false sharing. In Pverify [MDWSV87], Topopt [DN87] and Maxflow [Car88] the programmers made no attempt to improve locality, because of the added complexity of (multiprocessor) cache-conscious programming.

6 Results

We present two sets of results to describe the impact of our analysis and transformations on the benchmarks. The first demonstrates their overall effectiveness in eliminating false sharing, as well as the relative contribution of the different phases of the static analysis and the different transformations. These results were obtained by trace-driven simulation. The second details the effect on overall performance by using data from direct execution on the two multiprocessors.

6.1 Eliminating False Sharing

Figure 2 shows the shared data and false sharing cache miss rates for each program before and after applying transformations, and using the full scope of the static analysis to make the transformation decisions. The total miss rate is not shown, because it closely tracks the shared data miss rate in almost all cases. The figures confirm previous results that false sharing misses have the greatest impact at large block sizes and less at smaller.

The transformations have noticeably reduced the false sharing miss rate for all programs. For Mp3d, Pverify and Water, false sharing misses have been all but eliminated; on average less than 0.35% of shared data misses caused by false sharing remain for any block size. The transformations have also been very successful in eliminating false sharing misses in Topopt. Averaged over multi-word cache blocks the reduction is 79% (56% to 87%). The remainder of Topopt's false sharing is primarily caused by writes into a shared array that is partitioned across the processes in such a way that it is not detectable by our static analysis[4]. For LocusRoute the false sharing miss rate is cut in half for block sizes of 128 and 256 bytes, but is only reduced by 5% to 10% for smaller block sizes. The transformations had the smallest impact on Maxflow. Much of the false sharing in Maxflow is caused by busy, write-shared scalars that have been allocated to the

[4] Using manual inspection of the program and applying an additional transformation we call "privatization", we were able to further reduce Topopt's false sharing miss rate for all block sizes to less than 0.12%. We have not (yet) automated privatization.

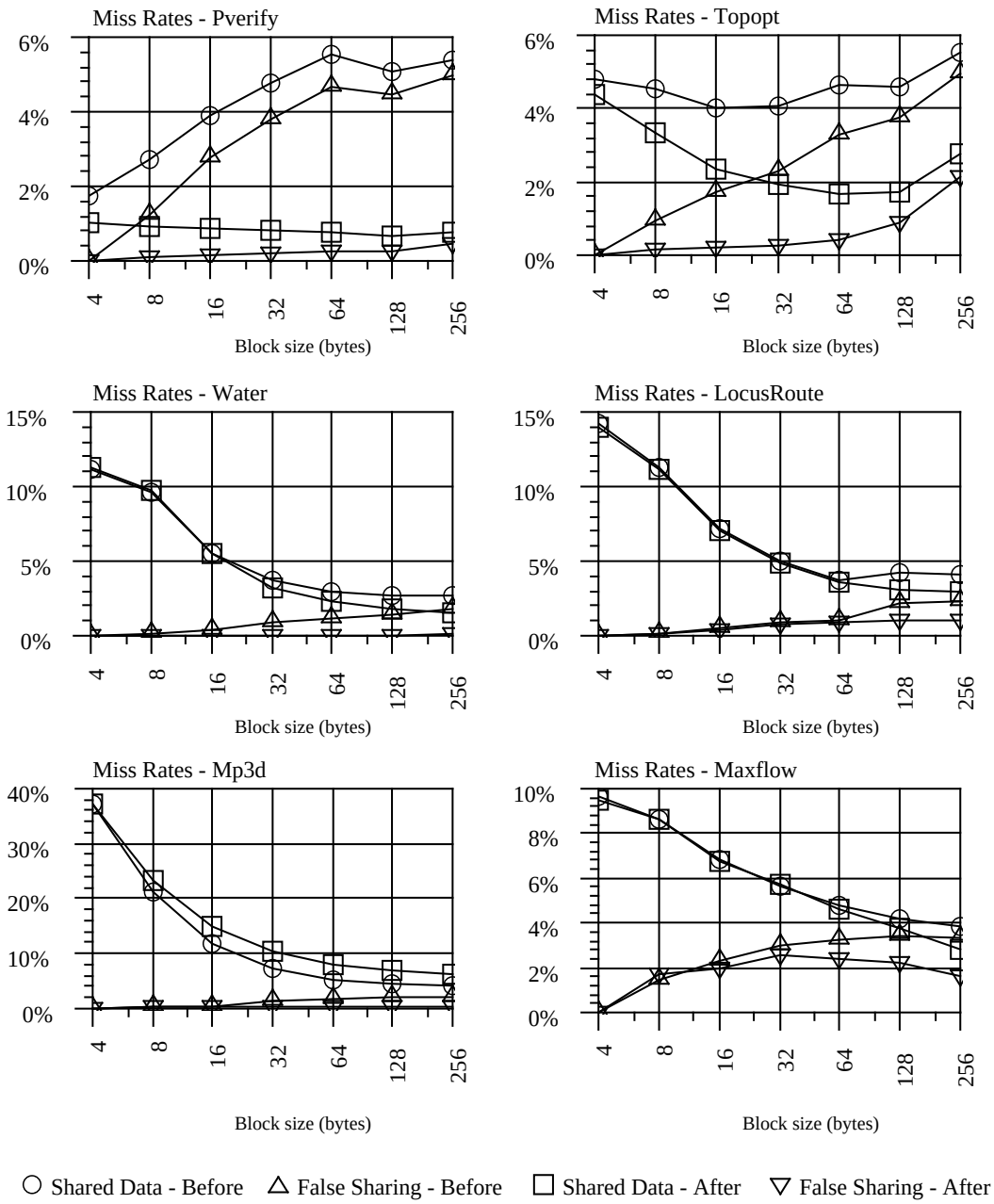

Figure 2: Shared data and false sharing miss rates, before and after transformations, for the programs in our workload. Notice that the miss rate axis varies across the programs.

same cache block. Our static analysis accurately detects per-process accesses to arrays and structures, but cannot identify when write-shared scalars cause false sharing.

Topopt and Pverify had previously been analyzed and transformed manually [EJ91]. The static analysis achieved reductions in false sharing that well exceeded the previous results. For Topopt the manual approach succeeded in eliminating an average of 59% of false sharing misses, while our compile time approach

eliminated 79%. Similar figures for Pverify were 76% versus 91%. Considerable programmer effort had been expended to improve data locality in the SPLASH programs. Nevertheless, our static analysis was able to eliminate a significant portion of the false sharing misses that still remained.

Program	Total reduction	Fraction of reduction by stage of analysis				
		+locks	+side-effect	+multiple RSDs	+profiling	+non-concurrency
LocusRoute	23.6%	23.6%				
Maxflow	21.6%	21.6%				
Mp3d	65.9%	65.9%				
Pverify	91.2%	3.1%	6.4% G		81.6% I	
Topopt	79.9%		12.5% I	6.1% I		61.3% G
Water	99.2%	99.2%				

Table 2: Average (over multi-word block sizes) reduction in false sharing by different stages of analysis and different transformations. A "G" indicates that group and transpose was used, while "I" indicates indirection.

Although false sharing was completely or significantly eliminated (in all programs but one), different analysis and transformations were responsible. To gauge the effectiveness of the different techniques, we examine them in five stages of increasing sophistication. In the first stage we only consider locks and pad them to the size of the cache blocks. Stage two builds upon stage one by including per-process control flow analysis and side-effect analysis, but without the enhancements. Stage three adds multiple regular section descriptors to the side-effect analysis, while stage four includes static profiling. Finally, stage five adds non-concurrency analysis. Table 2 summarizes the results. It depicts the average reduction in false sharing across multi-word cache blocks for each program (column 2), together with a breakdown of the reduction caused by transformations at each stage of analysis (columns 3–7).

Most of the false sharing was caused by contention for locks, and the analysis accurately detected this. For four of the benchmarks, LocusRoute, Maxflow, Mp3d and Water, the most successful transformation was also the simplest (padding locks). The other stages of analysis discouraged additional transformations. (Recall that these are the applications in which programmers had designed shared data structures to maximize spatial and processor locality.)

When the static analysis detected additional opportunities to apply transformations, the highest benefits where obtained with the most aggressive analysis. Pverify has two vectors that were accessed overwhelmingly on a per-process basis. Traditional side-effect analysis (without the enhancements) could not detect this, because other, and less frequent, accesses to these vectors forced a loss of precision when the side-effect information was summarized. By using multiple regular section descriptors and static profiling, our algorithms identified and factored out the dominant access pattern, and consequently transformed the data. For Topopt non-concurrency analysis detected that a set of vectors was accessed on a per-process basis in all but one phase of the program. It therefore transformed for the dominant access pattern.

For all but one program the shared data miss rate mirrors the decline in the false sharing miss rate, i.e., the

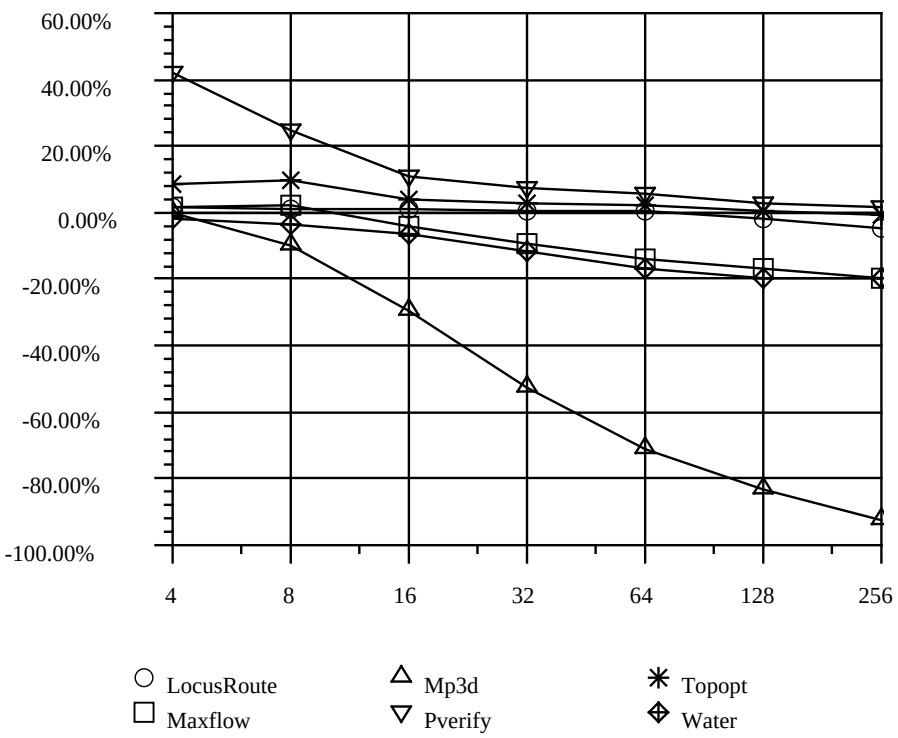

Figure 3: Deviation from expected shared data miss rate.

transformations yield an improvement in processor locality that counterbalances any loss in spatial locality[5]. The effect can be measured by comparing the shared data miss rate of each transformed program with an *ideal* miss rate that reflects no change in spatial locality. The ideal shared data miss rate is computed by reducing the original shared data miss rate by the percentage decline in the false sharing miss rate multiplied by the ratio of false sharing misses to shared data misses. Figure 3 shows the percentage deviation of the shared data miss rate, after transformations, from the ideal; a negative deviation indicates that the ideal shared data miss rate is less than the observed miss rate, i.e., there is a loss in spatial locality. The figure shows that with two exceptions, the transformations have little effect on spatial locality. For LocusRoute, Topopt, Maxflow and Water, the shared data miss rates neither improve on the ideal miss rate by more than 10%, nor do they exceed it by more than 20%. LocusRoute shows the least effect on spatial locality, while the spatial locality in Topopt improves slightly. Figure 2 shows that even though Maxflow and Water both experience a small loss in spatial locality, it is still outweighed by the drop in false sharing.

The exceptions are Pverify and Mp3d. The transformations enhance the spatial locality in Pverify, particularly for the smallest block sizes. Both Pverify and Topopt are examples where indirection actually betters spatial locality, despite the addition of pointers. The transformations do not work well for Mp3d, where the shared data miss rate climbs to almost twice that of the ideal miss rate for 256 byte blocks.

[5] Recall that spatial locality may suffer because of either padding or the addition of a pointer in indirection.

Padding locks to the size of cache blocks in Mp3d turns out not to be profitable, because there is little contention for the locks. Mp3d is the single case when locks should be allocated with the data they protect.

6.2 Execution Time

The previous results demonstrate that the static analysis techniques and transformations are successful in eliminating false sharing misses. However, whether the false sharing miss reductions produce improvements in execution time depends on two factors. The first is the size of the original false sharing problem. Programs that have less false sharing, because programmers had designed shared data structures to minimize false sharing and maximize spatial locality, reap lower benefits than programs in which no such attempts had been made. Second, the design of the memory subsystem architecture has an effect. Machines with a small cache, a small block size and a low cache miss penalty benefit less than architectures with a larger configuration.

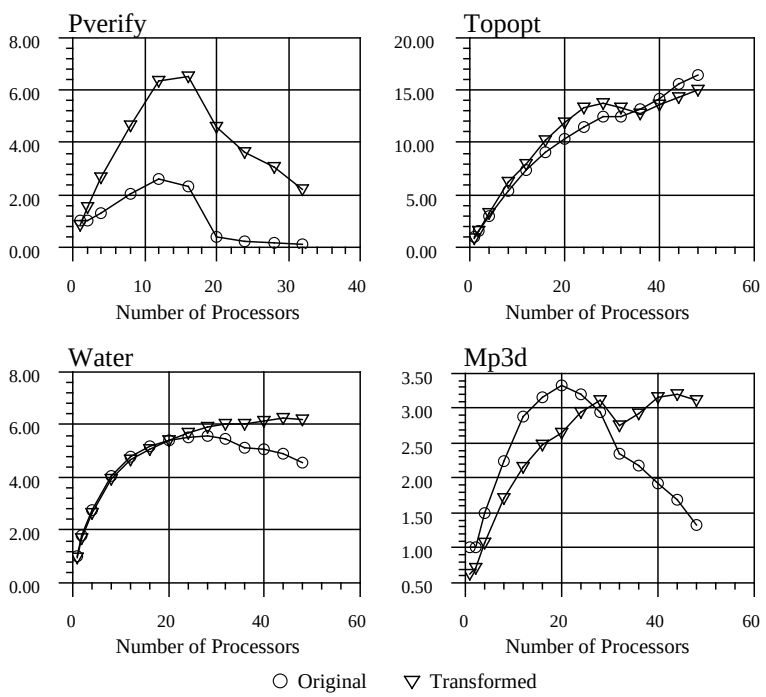

Figure 4: Speedup of original and transformed versions of Topopt, Pverify, Mp3d and Water on the KSR1. Note that the speedup axis varies across programs.

The best results occurred with applications that had not been programmer-enhanced and a multiprocessor with large cache and block sizes and a long miss penalty, i.e., Pverify and Topopt executing on the KSR1. For Pverify (Figure 4), the difference in speedup between the two versions is very large. For up to 16 processors, the average speedup after applying transformations is almost 2.4 times greater than before. Beyond 16, when the problem no longer scales with the number of processors, the processors' contention for data dominates

and the overall performance of both versions is reduced. However, the original version is impacted more severely, because of false sharing of the locks. (In fact, the original version is 2.7 times faster when it runs on a single processor than when it runs on 20!)

The transformed version of Topopt exhibits better speedups than its original version when running on 32 processors or less. For example, execution time is reduced by as much as 14% when running on 24 processors. Data from the KSR performance monitor attributes the speedups to a reduction in memory stall time, which declines an average of 39% for runs of the program using 2 to 32 processors. For runs with more than 32 processors the original version does better.

The results for Pverify and Topopt also show that applying indirection can have an overall positive effect on execution time, even though it adds one more memory access to every reference of the data structure. These accesses are much more likely to be cache hits; therefore the total amount of time spent to access the transformed data structures (more cache hits, but fewer invalidation misses) is less than in the original program.

The difference in speedup between the original and transformed versions of Water (a programmer-optimized program) is slight until more than 24 processors are used. Not until then does contention for locks reach the level where the decrease in false sharing from padding outweighs the loss of spatial locality caused by the additional space. After 24 processors, the difference between the original and transformed versions grows, indicating that the transformations have postponed the point at which the problem no longer scales with the number of processors. The performance of the other programs where locks were padded (LocusRoute, Maxflow) are similarly affected.

As expected, the loss of spatial locality caused by the transformations adversely effects Mp3d's execution time. As with the other programmer-optimized programs, that loss is eventually offset by the reduction in false sharing. At a similar number of processors, the original version begins to suffer from a lack of scalability. Again, the transformations serve to enhance program scalability.

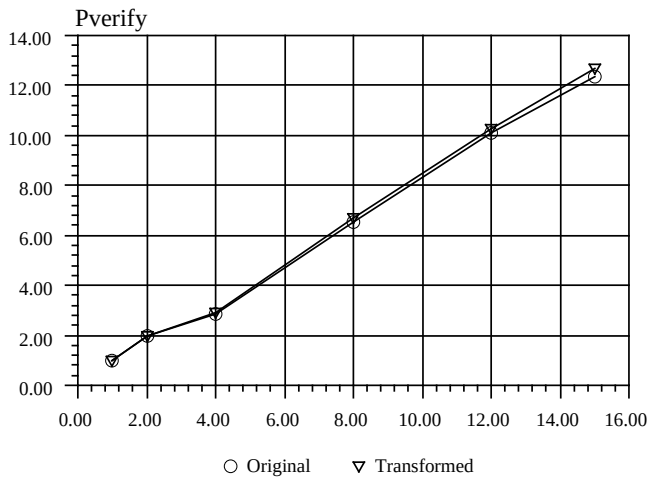

Figure 5: Speedup of original and transformed versions of Pverify on the Sequent Symmetry.

The impact of transformations on the execution time on the Sequent is all but non-existent. (Speedups for Pverify are the largest we saw for any program.) We attribute the lack of improvement to three aspects of the Sequent's cache organization. First, the average lifetime of data objects in the small 64 KB board-level cache is shorter than on the KSR, and they are often replaced before they can be invalidated. This reduces the original false sharing problem. Furthermore, the smaller cache size heightens the negative impact on capacity misses relative to any positive effect on invalidation misses of transformations that increase the data size. Second, the cache block size is relatively small, only 16 bytes; fewer data objects can share cache blocks and subsequently the amount of false sharing is less. Third, the Sequent Symmetry has a low memory latency relative to the processor speed, so reductions in bus traffic have less impact than on the KSR.

7 Related Work

Related work describes either hardware solutions or compile time techniques that reorganize control structures rather shared data.

Dubois et al. [DSR+93] reduced false sharing by either delaying invalidations (at the sender, receiver or both) until special *acquire* or *release* instructions are executed, or performing invalidations on a word basis. Delaying invalidations both at the sender and the receiver and invalidating cache subblocks consistently perform well. The former reduced false sharing misses by 85% to 100%; the latter totally eliminated them. These reductions were achieved at the cost of increased memory traffic and additional hardware complexity. The first approach requires a change in the instruction set architecture, as well as hardware to implement invalidation buffers at each processor node. The second requires an invalid bit per word in the cache block, and causes more invalidations when the writes exhibit spatial locality.

Granston [Gra93] presented a theory to identify and eliminate page-level sharing between processors that occur in parallel do-loops. The transformations select blocking and alignment factors that cause minimal overlap between sets of pages accessed by different processors. No results have been reported, as the theory and transformations have yet to be implemented.

Ju and Dietz [JD91] restructured a program fragment of several loops accessing array elements. Their restructuring algorithm applies loop transformations (such as loop distribution) and data layout transformations (accessing arrays in row or column major order), according to a coherency cost function. The restructuring provided a 25% improvement in execution time of the loops for a 64 KB cache.

Gupta and Padua [GP91] also examined sequential programs that were automatically parallelized at the loop level. They strip-mined the loops to the size of the cache block and assigned each strip to a different processor. The decline in miss ratios ranged from 4% to almost 60%, as block size was increased to 128 bytes. No execution times were reported.

Peir and Cytron [PC89] partitioned loops to minimize inter-processor communication when processing recurrences. Their mechanism for partitioning utilizes loop unrolling and dependence vectors. Partitions are then scheduled on different processors.

For the compiler-based approaches, the workload consisted of either loops or library routines that have

fine-grain parallelism. Their studies recorded performance improvements only for the code fragments that have been transformed. Therefore the results were overly optimistic with regard to the expected performance of executing entire programs.

8 Conclusion

In this paper we have analyzed the effectiveness of using compile-time analysis and shared data transformations to reduce false sharing in coarse-grained, explicitly parallel programs. For the most part our techniques were successful in eliminating false sharing misses. For the programs in our workload they reduced the false sharing miss rate by an average of 64%. In two programs that had not previously been programmer-enhanced for locality the average reduction in the false sharing miss rate was 86%. The reduction in false sharing miss rates were for all but one program followed by proportional reductions in the shared data miss rate. The exception proved to be due to a loss of spatial locality caused by isolating locks from the data they protected.

We also examined how well the reduction in false sharing misses translated into improvements in execution time. The programs were run on two different multiprocessors that represent different points in the memory subsystem design space. The KSR1 has large caches, block size and miss penalty; the Sequent Symmetry has a much smaller configuration. The results indicated that execution time is very sensitive to the memory subsystem architecture, as well as the original amount of false sharing. The two programmer-unoptimized programs running on the larger machine benefitted much more from compiler-directed transformations than other cache/program combinations. The primary effect on the programmer-optimized programs was to improve their scalability, relative to the untransformed versions. On the smaller machine no significant speedups were measured.

Our results indicate that with the trend toward larger caches, larger coherence units, and longer memory latencies, false sharing will have an increasingly large (negative) performance impact. Regaining the performance will necessitate either a significant programming effort to improve locality or the use of a compile time system like ours.

This paper argues for the latter, on two grounds. The first concerns ease of programming. Compiler-based solutions allow the programmer to focus on the semantics of the application, divorced from details of the caching structures and coherency operations, the shared data structures' layout in memory and the often nonintuitive (particularly over the temporal domain) cross-processor memory accesses to them. The second relates to the gains in multiprocessor performance that the first brings about. Our particular static analyses led to transformations that were more successful than programmer efforts for all applications in our workload. These benefits were realized with only a 5% increase in compile time.

References

[AG88] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under mach. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 215–225, 1988.

[Anoa] Anonymous. Reference omitted for anonymous reviewing.

[Anob] Anonymous. Reference omitted for anonymous reviewing.

[Anoc] Anonymous. Reference omitted for anonymous reviewing.

[Ban79] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Sixth Annual Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. *CACM*, 21(9):724–736, September 1978.

[Car88] F. J. Carrasco. A parallel maxflow implementation. CS411 Project Report, Stanford University, March 1988.

[CK88a] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, (5):517–550, 1988.

[CK88b] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Conference on Programming Languages Design and Implementation*, pages 57–66, June 1988.

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

[DSR+93] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *20th Annual International Symposium on Computer Architecture*, pages 88–97, June 1993.

[EJ91] S.J. Eggers and T.E. Jeremiassen. Eliminating false sharing. In *International Conference on Parallel Processing*, volume I, pages 377–381, August 1991.

[EKKL90] S.J. Eggers, D.R. Keppel, E.J. Koldinger, and H.M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, May 1990.

[GP91] M. Gupta and D. A. Padua. Effects of program parallelization and stripmining transformations on cache performance in a multiprocessor. In *International Conference on Parallel Processing*, volume 1, pages 301–304, August 1991.

[Gra93] E. D. Granston. Toward a compile-time methodology for reducing false sharing and communication traffic in shared virtual memory systems. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Sixth Workshop on Languages and Compilers for Parallelism*, August 1993.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[JD91] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fourth Workshop on Languages and Compilers for Parallelism*. Springer Verlag, August 1991.

[KSR92] Kendall Square Research. *KSR-1 Principles of Operation*, 1992.

[LLG+92] D. Lenoski, J. Laudon, K Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3), March 1992.

[LT88] R. Lovett and S. Thakkar. The symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[MDWSV87] H-K. T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli. Logic verification algorithms and their parallel implementation. In *Proceedings of the 24th Design Automation Conference*, pages 283–290, July 1987.

[MR93] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 129–138, May 1993.

[Mye81] E. Myers. A precise inter-procedural data flow algorithm. In *Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[PC89] J.K. Peir and R. Cytron. Minimum distance: A method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8):1203–1211, August 1989.

[PGH+89] C. Polychronopoulos, M. Girkar, M. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 39–48, August 1989.

[SWG91] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[TLH90] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 266–270, August 1990.

[Wal91] D. W. Wall. Predicting program behavior using real or esitmated profiles. In *Conference on Programming Language Design and Implementation*, pages 59–70, June 1991.