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

Tor E. Jeremiassen
AT&T Bell Laboratories*
600 Mountain Ave.
Murray Hill, New Jersey 07974
tor@research.att.com

Susan J. Eggers
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195
eggers@cs.washington.edu

Abstract

We have developed compiler algorithms that analyze explicitly parallel programs and restructure their shared data to reduce the number of false sharing misses. The algorithms analyze per-process shared data accesses, pinpoint the data structures that are susceptible to false sharing and choose an appropriate transformation to reduce it. The transformations either group data that is accessed by the same processor or separate individual data items that are shared.

This paper evaluates that technique. We show through simulation that our analysis successfully identifies the data structures that are responsible for most false sharing misses, and then transforms them without unduly decreasing spatial locality. The reduction in false sharing positively impacts both execution time and program scalability when executed on a KSR2. Both factors combine to increase the maximum achievable speedup for all programs, more than doubling it for several. Despite being able to only approximate actual inter-processor memory accesses, the compiler-directed transformations always outperform programmer efforts to eliminate false sharing.

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 [TLH94, EJ91]. False sharing occurs when multiple processors access (both read and write) different words in the same cache block. Although the processors do not actually share data, they incur its costs, because coherency operations manipulate cache blocks. In a write-invalidate coherency protocol the overhead of false sharing takes the form of extra invalidations when a processor updates data and extra invalidation misses when other processors reread different data that reside in the invalidated cache block.

False sharing is caused by a mismatch between the memory layout of write-shared data and the cross-processor memory reference pattern to it. By changing the way shared data is laid out in memory to better conform to the memory reference pattern, false sharing can be eliminated. In particular, all data that are accessed by the *same* processor should be grouped together, improving processor locality. Individual data objects that are accessed by *multiple* processors should be separated and padded to the size of a cache block. Although this restructuring reduces false sharing, applying it universally may have a negative impact on spatial locality that outweighs the gain in processor locality. Therefore, it is important to carefully balance the tradeoff between processor and spatial locality, so as to maximize program performance.

To this end we have developed and incorporated into the Parafrase-2 [PGH+89] source-to-source restructurer a series of compiler algorithms [JE92, JE94] and a suite of data transformations. The algorithms analyze explicitly parallel programs; they produce information about each processor's memory reference patterns that identifies data structures susceptible to false sharing, decide whether transforming them will pay off and then choose appropriate transformations.

This paper evaluates that technique. We show through simulation that the analysis successfully identifies the data structures responsible for most false sharing misses, and makes appropriate tradeoffs between eliminating false sharing and reducing spatial locality. For example, with 128 byte cache blocks, 70% of the cache misses in our workload are due to false sharing. The transformations eliminate 80% of them, while increasing other types of misses by only 19%. The overall effect reduces the total number of cache misses by half. No single transformation is responsible for the false sharing reductions, even within a single program: all are important contributors to improved performance.

The reduction in false sharing misses has two effects on run-time performance as measured on a KSR2: reductions in execution time and improved program scalability. Of the two, improved scalability (better performance with increasing numbers of processors) is the decisive factor. Memory contention from false sharing in the untransformed programs grows more than linearly with the number of processors. The compiler-directed transformations alleviate this bottleneck, and extend scalability, often to the point where the maximum achievable speedup more than doubles. Before the point at which the performance of the unoptimized programs no longer scales, the compiler-optimized programs still have

*This work was performed while the author was at the University of Washington.

This work was supported by IBM Contract No. 18830046, ONR Grant No. N00014-92-J-1395, NSF PYI Award #MIP-9058-439, and NSF grants CCR-9200832 and CDA-9123308.

lower execution times, ranging from a modest (2%) to a more sizable (58%) amount.

We also compare the compiler-optimized approach to several programs in which considerable programming effort had been expended to improve data locality, including eliminating false sharing. Despite being able to only approximate actual inter-processor memory accesses, the compiler analysis always outperforms programmer hand-tuning, often substantially.

The next section identifies the parallel programming paradigm for which our algorithms are appropriate and describes the particular model used in our workload. Section 3 presents a brief overview of our compile time analysis, and describes the shared data transformations and heuristics for applying them. Section 4 describes our methodology and workload. Section 5 presents the experimental results which are the contribution of this paper. They are based on both simulation and execution time experiments and compare compiler-optimized programs to both unoptimized and hand-optimized programs. Related work is discussed in section 6, and section 7 concludes.

2 Model of Parallel Programming

Our analysis and transformations are appropriate for shared memory paradigms where accesses to shared data can be parameterized by variables that have different values for different processes. Examples of these variables include induction variables of *FORALL* loops in HPF [Hig93] and private variables, such as *pid* in Figure 1, in the fork/join model.

Our current implementation targets programs that use the latter: coarse-grained, explicitly parallel C programs that execute on shared memory multiprocessors. Examples of such programs can be found in the Stanford SPLASH application suite [SWG91]. These programs currently execute on small to medium scale multiprocessors, both commercially (e.g., Sequent Symmetry [LT88], SGI Challenge [GW94], SPARCcenter 2000 [M. 93], and the KSR2 [Ken94]) and in research environments (e.g., DASH [LLG+92], FLASH [LLG+94]).

The granularity of parallelism in these programs is coarse, on the level of an entire process. Our analysis assumes the number of processes equals the number of processors and processes do not migrate. (This restriction can be relaxed to allow a larger number of processes, but the analysis may then overestimate the amount of false sharing between the processors.) 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 code. 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 variable a *process differentiating variable* (PDV).

Process synchronization is performed using both locks and global barriers. Locks are used to enforce mutual exclusion, i.e., they serialize access to critical sections. Barriers separate phases of program execution. When the control flow of a process reaches a barrier, it must wait until all participating processes also reach it. Barriers are often 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.[1]

While our workload consists of programs written in C, our compile-time analysis and transformations rely on properties that are more restrictive than what the C programming model provides. The most important constraints involve pointers and separate compilation (a full description will appear in [Jer95]).

While our model allows for pointers, the full generality of pointers in C is restricted to reduce pointer aliasing of statically allocated data to that induced by pointer type parameters to functions. For example, pointers may only point to objects of the same type as in their declarations, and pointer arithmetic and indirection through arithmetic expressions are disallowed.

In order to ensure that any shared data transformation is applied universally to all accesses to a target data structure, separate compilation is restricted to only those modules that do not access shared data that may be targeted for transformation.

3 Compile-time Analysis and Transformations

Since this paper evaluates the ability of the static analysis to eliminate false sharing rather than the algorithms per se, we provide only an overview of the analysis and transformations. Section 3.1 briefly describes the compile-time analysis used to pinpoint data structures that are susceptible to false sharing. Section 3.2 illustrates how our four transformations eliminate false sharing, and Section 3.3 discusses under what conditions they are applied.

3.1 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. This compiler analysis involves three separate stages. The first uses interprocedural analysis of the control flow to determine which sections of code each process executes, and annotates the nodes of the control-flow graphs accordingly [JE92][2]. The second performs non-concurrency analysis [MR93] interprocedurally 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 [JE94]. The third stage performs an enhanced interprocedural, flow-insensitive, summary side-effect analysis [Bar78, Ban79, Mye81, CK88b] and static profiling on a per-process basis (based on the control flow determined in stage one) for each phase (determined in stage two).

Per-process references to shared data occur either as a result of the processes executing different code (and thus accessing different 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 (PDVs)[3] (stage 3) help detect the sec-

[1] HPF has direct counterparts to constructs in the fork/join model. For example, iterations of FORALL loops are "forked" implicitly; values of the FORALL induction variables could act as a PDVs; and there is an implicit barrier after a FORALL.

[2] This reference describes the general technique, but an implementation we no longer use.

[3] 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

```
private int pid;
shared barrier_t Barr1, Barr2, Barr3;
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(&Barr1);
        SubPart2(pid);
        Wait_Barrier(&Barr2);
        if (pid == 1)
            converged = TestConverged();
        Wait_Barrier(&Barr3);
    }
}
```

(b)

```
SubPart1(proc)
int proc;
⋮
cell1 = value1[proc];
cell2 = value2[proc];
⋮
```

(c)

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

ond. The side-effect analysis represents the sections of each array that each process accesses using bounded regular section descriptors[4] to describe the index expressions [HK91]. When a regular section descriptor contains a PDV in the index expressions, 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 PDVs, and applying the side effect analysis to the separate control-flow graphs yields the sections of shared data that each process reads and writes.

We improve upon traditional summary side-effect analysis in two respects. First, to improve its accuracy we allow multiple regular section descriptors [CK88a, HK91] and only merge them 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). Second, to pinpoint data structures most responsible for false sharing, we use static profiling to produce a weighting of the side-effects with respect to estimated execution frequency.

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 detects the memory access pattern of distinct phases of a program between barriers, and, more importantly, when the pattern shifts. Coupled with static profiling, it determines the dominant sharing pattern in the program and restructures shared data for that pattern.

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.

3.2 Shared Data Transformations

In order to eliminate false sharing, data must be restructured so that (1) data that are only, or overwhelmingly, accessed by one processor are grouped together, and (2) write shared data objects with no processor locality [AG88] do not share cache lines. Two transformations, originally devised for manual application, *group and transpose* and *indirection* [EJ91], address item (1); the third, *pad and align*, is well known and addresses item (2).

Group & Transpose: Group & transpose (Figure 2a) physically groups per-process data together by changing the layout of the data structures in memory. It gathers vectors in which adjacent elements are accessed by different processors into a group and then transposes it. 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.

Indirection: 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 effect by using indirection. Indirection (Figure 2b) 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.

Pad & Align: The third transformation pads and aligns on cache block boundaries data (scalars or array elements) that are falsely shared in the short term but 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

is an example.

[4] A bounded 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).

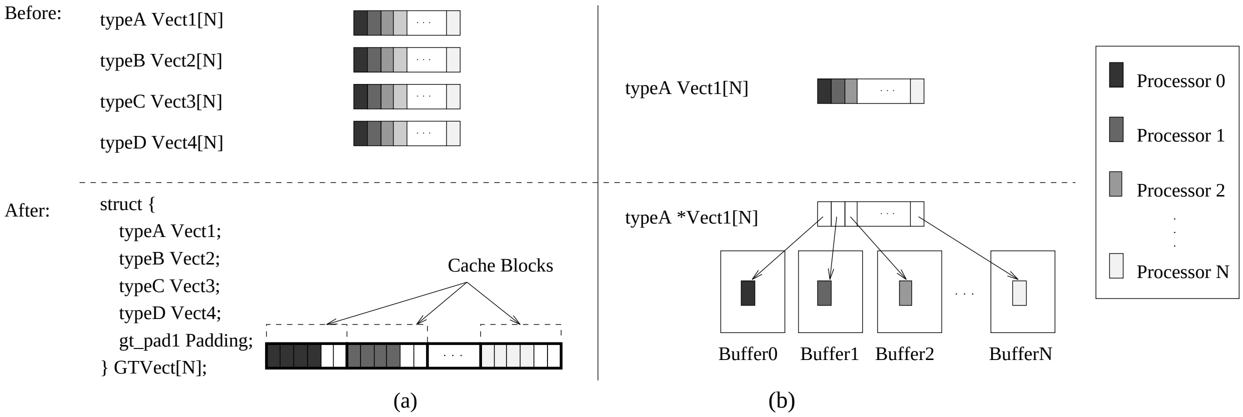

Figure 2: Illustration of (a) group & transpose and (b) indirection.

a processor accesses 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 false sharing. Pad and align has been used to eliminate false sharing in both cache blocks and pages in other work. Our application of padding differs in that we apply it only when indicated by the static analysis, as opposed to from feedback from off-line cache simulation profiles [TLH94], or based on programmer knowledge [BFS89].

Locks: Locks are also padded, to the size of the cache block, rather than allocated with the write-shared data they protect. Co-allocating locks and data [TLH94] improves spatial locality, but 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 additional invalidations, and then invalidation misses when the waiting processors reread the status of the lock. Our approach of always padding locks decreases spatial locality, but eliminates any false sharing caused by lock contention.

3.3 Transformation Heuristics

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 transformations should be applied to eliminate it. The heuristics were developed by comparing the results of the per-process side-effect analysis to profiling information from simulations that showed the number of false sharing misses per data structure. The factors used in the heuristics to make the transformation decisions are the type (read/write, shared/per-process), stride (known/unknown) and frequency of access to the elements of a data structure. In order to apply either group & transpose or indirection to a data structure, the pattern of writes to the data structure must be per-process and the pattern of reads either per-process or read-shared without spatial or processor locality. If the pattern of reads is read-shared with locality, the data structure is transformed only if the number of writes dominate the number of reads by at least an order of magnitude. This is done so that the reduction in false sharing will exceed any performance loss from reduced spatial locality. Except for locks, which are always padded, data structures are only padded and aligned on cache block boundaries when both the reads and the writes exhibit sharing without processor or spatial locality.

4 Methodology and Workload

We perform both simulation and execution-time experiments to quantify the effects of transforming shared data on the programs in our workload. False sharing reductions and other cache miss metrics were measured using trace-driven simulation. Each program was traced (both before and after shared data was transformed), 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[5]. We studied block sizes ranging from 4 to 256 bytes.

Execution times were measured on a 56-processor Kendall Square Research KSR2 [Ken94]. Each processor has a 512 KB first level cache, divided equally between data and instructions. The second level cache contains 32 MB, and uses a coherency unit of 128 bytes. The second level cache miss latency is 175 cycles, if it is serviced by a processor on the same ring, and 600 cycles if the servicing processor is on a different ring.

The KSR2 default lock data structure is large (80 bytes) and aligned on cache block boundaries. To make implementation-independent comparisons with the simulations, and to study the effect of padding and aligning locks, we used KSR2 synchronization primitives to provide a smaller (1 word), alternate implementation of locks in the untransformed versions of the programs.

Gauging the impact of the static algorithms and transformations on program performance and comparing the compiler analysis to programmer efforts to eliminate false sharing requires three versions of each program: an unoptimized version, a compiler-transformed version and a hand-optimized version. The programs we collected had been hand-optimized for locality to varying degrees. In one group, that included Maxflow [Car88], Pverify [MDWSV87] and

[5]Infinite caches can be used to approximate very large (on the order of several megabytes) second level caches [Egg91].

Program	Description	Lines of C	Versions		
Maxflow	Maximum flow in a directed graph	810	N	C	
Pverify	Logical verification	2759	N	C	P
Topopt	Topological optimization	2206	N	C	P
Fmm	Fast multipole method (n-body)	4395	N	C	P
Radiosity	Equilibrium distribution of light	10908	N	C	P
Raytrace	Rendering of 3-dimensional scene	12391	N	C	P
LocusRoute	VLSI standard cell router	6709		C	P
Mp3d	Rarefied fluid flow	1653		C	P
Pthor	Circuit simulator	9420		C	P
Water	N-body molecular dynamics	1451		C	P

Table 1: Benchmarks used in our study. Version refers to (N)ot optimized, (C)ompiler optimized, or (P)rogrammer optimized.

Topopt [DN87], no effort had been made to improve locality. For Pverify and Topopt, in particular, the programmers had constructed data structures to match their "natural" way of thinking about the semantics of the program algorithms, rather than for better memory system performance. To provide hand-optimized versions of these programs (Pverify and Topopt), we manually transformed them [EJ91].

In another group that comprised the original SPLASH benchmark suite [SWG91] (LocusRoute, Mp3d, Pthor, Water) and the SPLASH2 benchmarks (Fmm [SHHG93], Radiosity and Raytrace [SGL94]), programs had been highly optimized for locality, including eliminating false sharing. The SPLASH2 programs contained several easily identifiable data structures whose elements had been organized by processor (in our terminology, grouped and transposed), and padded. We undid these transformations to produce unoptimized versions of the programs, but made no other changes. In addition to providing a general comparison between the compiler-directed and hand-tuned optimizations, these hand-unoptimized programs enabled us to gauge the compiler's ability to detect and transform data structures the programmer had chosen. The programmer efforts to improve locality in the original SPLASH benchmarks were not as obvious. Therefore we left them as is.

5 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 and the relative contribution of the different transformations, all via simulation. The second measures the impact of eliminating false sharing on execution time and program scalability and compares the compiler approach to that of programmer hand-tuning.

Simulation Results: Figure 3 and Table 2 show the results of applying the algorithms and transformations to the unoptimized programs in our workload. In the figure, the white portion of each bar is the miss rate due to false sharing; the black portion represents the remaining misses. It also indicates what the total minimum miss rate for that block size would be if false sharing were eliminated without any effects on spatial locality.

The compiler-directed shared data restructuring reduced false sharing in all programs for all block sizes, regardless of the size of the original false sharing miss rate. (False sharing is greater with larger block sizes; and in our programs the amount of false sharing, of course, varied.) The greatest reductions occurred for Fmm, Pverify and Radiosity, where on average more than 90% of all false sharing misses were eliminated. False sharing miss rates in Maxflow, Raytrace and Topopt, were also significantly reduced, although not to the same extent. In Maxflow and Raytrace, the remaining false sharing is mostly caused by a few busy, write-shared scalars that were allocated to the same cache block. They did not appear as candidates for restructuring, because the static profiling underestimated their dynamic access frequency. The remaining false sharing misses in Topopt occur mostly in a write-shared array that is dynamically partitioned across the processes in a revolving manner. False sharing misses occur in the cache blocks that contain elements from more than one partition. Since the partitioning of the array is dynamic and revolving, the static analysis cannot detect the per-process accesses. Nor does the array appear to the compiler to have poor spatial locality, because the writes to the elements in a processor's partition occur with unit stride.

Although, overall, the transformations were very successful in eliminating false sharing misses, no single transformation was responsible for the reductions for all programs, or even for a single program. Group & transpose and padding locks were most applicable, used in 5 out of 6 programs. However, the majority of false sharing misses were eliminated by group & transpose and indirection. Unlike padding, these transformations are harder to apply using simulation profiles; static analysis can more easily ensure that only data that are accessed by the same process are grouped together. Our compiler-driven transformations provide this.

One transformation (group & transpose) improves spatial locality, while others (indirection and pad & align) decrease it. Since both sets of transformations were applied to all but one program, our results reflect both effects. For most programs the change in spatial locality (as reflected by the black portion of the bars in Figure 3) was modest, since the effects of the transformations canceled one another. The increase in misses other than those attributable to false sharing was significant only for Maxflow (it almost doubled at 128 byte cache blocks), which is restructured with two transformations, both of which increase the shared data size. However, for all programs and at all block sizes (data not shown) the reduction in false sharing more than compensated for any decrease in spatial locality, and the total miss rate fell.

Execution Time Results: Eliminating false sharing affected two aspects of overall program performance: execution time and program scalability. The difference in execu-

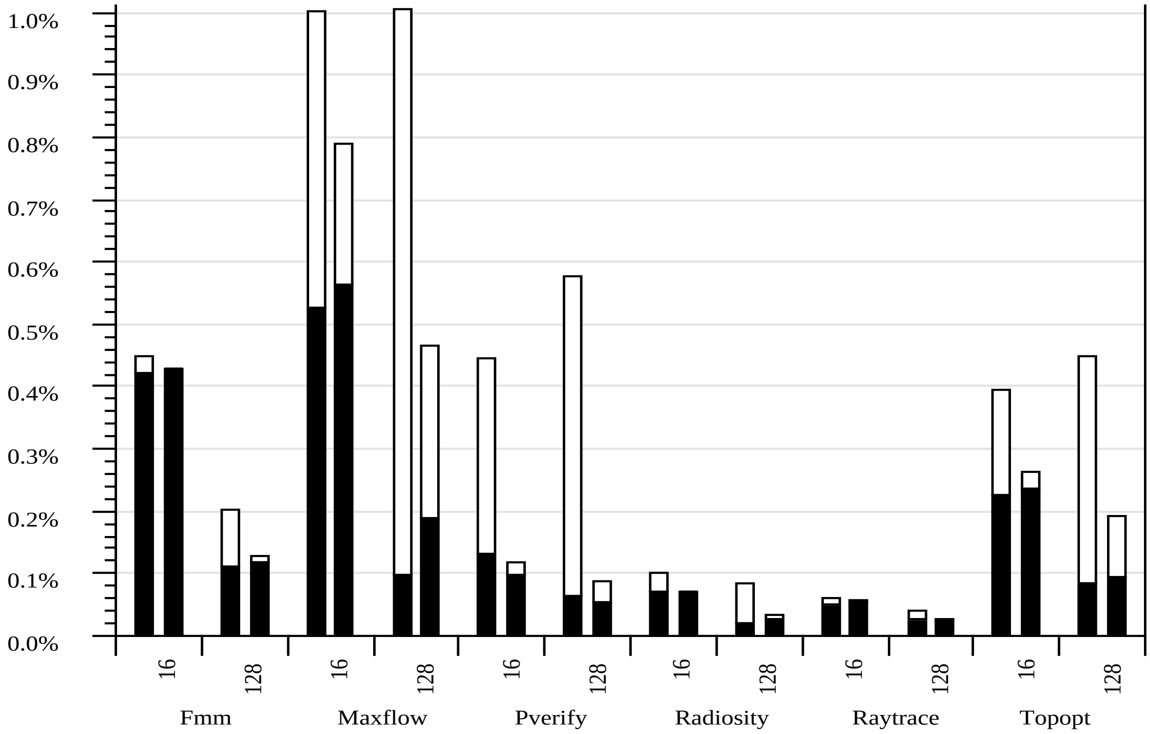

Figure 3: Total cache miss rates for unoptimized (left) and compiler-transformed (right) versions of programs for 16 and 128 byte cache blocks. (Recall that the KSR2 has a 128 byte block.) The portion of the miss rate that is due to false sharing is the white portion of each bar. Each program was run on 12 processors, except for Topopt which was run on 9.

tion time between the unoptimized and compiler-optimized versions of the programs, over the range of processors where the unoptimized version still scaled (i.e., where an increase in the number of processors produced a drop in execution time), progressively increased. Maximum improvements were modest for Fmm (3%), Radiosity (6%) and Raytrace (2%), all programs in which we undid only the easily identifiable programmer transformations to produce unoptimized versions. Reductions were better for the programs with no hand-tuning, Maxflow (50%), Pverify (58%) and Topopt (20%). Situations where the transformations had minimal performance impact occurred primarily when (1) there were few processors accessing the shared data, and either (2) the absolute miss rate value was small (Radiosity), or (3) the reduction in false sharing misses, although large, was a small proportion of total misses and therefore had little effect on the total miss rate (Fmm, Raytrace).

As the number of processors grew, so did the interprocessor contention for data structures that are falsely shared. At some threshold number of processors, which varies across the programs, the memory contention created by false sharing had such a severe impact that it reversed the speedup trend of the unoptimized versions of the programs. However, the performance of the transformed versions continued to improve, reaching maximum scalability at a greater number of processors (representative programs appear in Figure 4). (The only exception was Pverify, for which the unoptimized and compiler-optimized versions both scaled to 16 processors.) Thus, compiler-transformed versions of the programs not only run faster, but, since they scale better with the number of processors, the maximum performance they can achieve is often much higher (Table 3, columns 2 and 3). This maximum performance difference is particularly striking for Fmm, Pverify, Radiosity and Maxflow, where the transformed versions exceed the maximum speedup of the original by factors of 2.1, 2.4, 2.7 and 3.1, respectively.

Despite being based on algorithms and heuristics that can only approximate dynamic per-processor accesses and processor interaction, the compiler-directed transformations always outperformed programmer efforts, sometimes more than doubling the maximum obtainable speedup. The compiler was able to eliminate more false sharing misses in all programs. In some cases, it was simply more exhaustive in its coverage. For example, the programmer missed opportunities to apply group & transpose in Pthor, Pverify and Topopt; indirection in Pverify and Topopt; and pad & align in Radiosity and Pthor. In others, it made a better tradeoff between spatial and processor locality. For example, the programmer padded and aligned an array in Raytrace that the static analysis had concluded was not predominantly accessed on a per-process basis. Finally, the programmer sometimes left locks unpadded or associated them with the data they protected; Radiosity, LocusRoute and MP3D suffered from both.

Program	Total reduction in false sharing	Fraction of reduction by transformation			
		Group & Transpose	Indirection	Pad & Align	Locks
Maxflow	56.5%			49.2%	7.3%
Pverify	91.2%	6.4%	81.6%		3.1%
Topopt	79.9%	61.3%	18.6%		
Fmm	90.8%	84.8%			6.0%
Radiosity	93.5%	85.6%		1.0%	6.8%
Raytrace	78.3%	70.4%		3.3%	4.6%

Table 2: The false sharing miss rate reduction broken down by transformation. Numbers are averages over 8-256 byte cache blocks.

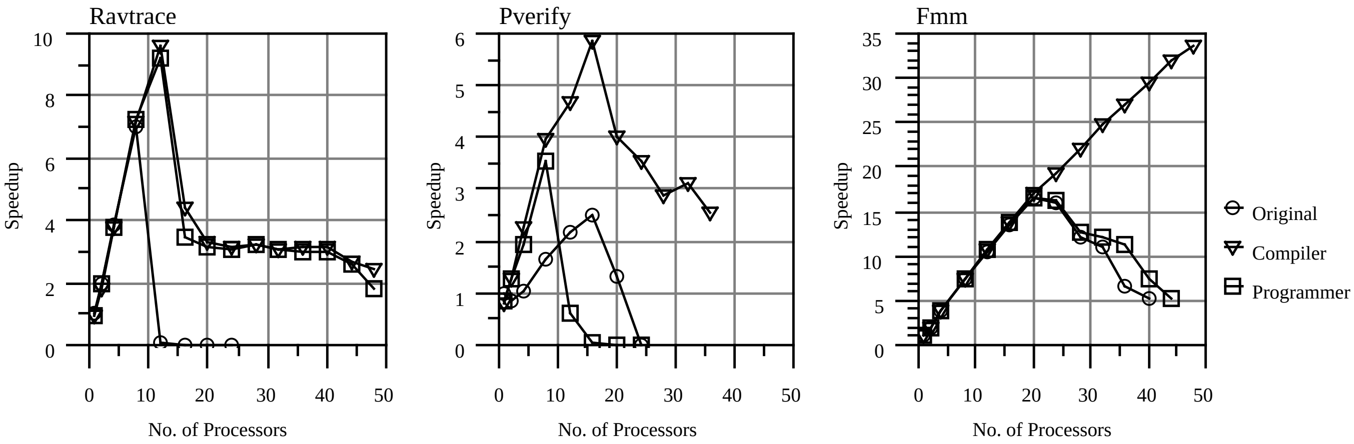

Figure 4: The scalability (speedup vs. number of processors) of unoptimized, compiler-optimized and programmer-optimized versions for 3 representative programs. Raytrace is typical of programs where the compiler and programmer approaches were comparable, Fmm, of those where programmer efforts brought little gain and Pverify, which falls in between. All data points are speedups relative to the uniprocessor execution of the unoptimized version. Note the different scales on the vertical axis.

6 Related Work

The research that is most closely related to ours is Torrellas et al. [TLH94], who reduced false sharing using a somewhat different set of transformations that were applied manually. Like us, they pad and align records and busy scalars; however, they did not use group & transpose or indirection, and they co-allocated locks with the scalars they protect rather than placing them in separate cache blocks. In addition, they used detailed, trace-driven simulation profiles, rather than static analysis, to determine which data structures suffered from false sharing and to guide the application of the transformations. On average, for 64 byte cache blocks, they reduced the number of shared misses by 10% and 13%, for 16 and 32 processor simulations, respectively. In contrast, on a slightly different workload, our transformations reduced the *total* miss rate by an average of 49% (on the unoptimized programs, also for 64 byte blocks, but with 12 processors).

Dubois et al. [DSR+93] reduced false sharing with hardware, either by delaying invalidations (at the sender, receiver or both) until special *acquire* or *release* instructions were executed, or by 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.

Several compiler approaches reorganize control structures rather than data. One group used workloads that consisted of either loops or library routines that have fine-grain parallelism [JD91, GP91, PC89]. Their studies recorded performance improvements only for the code fragments that were transformed. Therefore the results are overly optimistic with regard to the expected performance of executing entire programs. 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.

Program	Maximum Speedup and Scalability (# of processors) Original	Compiler	Programmer
Maxflow	1.4 (8)	4.3 (16)	
Pverify	2.5 (16)	5.9 (16)	3.5 (8)
Topopt	9.2 (44)	10.3 (28)	10.2 (28)
Fmm	16.4 (20)	33.6 (48+)	16.4 (20)
Radiosity	7.0 (8)	19.2 (28)	7.4 (8)
Raytrace	7.0 (8)	9.6 (12)	9.2 (12)
LocusRoute		12.3 (20)	12.0 (20)
Mp3d		2.9 (28)	1.3 (4)
Pthor		2.8 (4)	2.2 (4)
Water		9.9 (40)	4.6 (12)

Table 3: Maximum speedups for original, compiler-optimized and programmer-optimized versions and the number of processors at which they occur. Note, for LocusRoute, Mp3d, Pthor and Water only programmer- and compiler-optimized versions were available, while for Maxflow, no programmer-optimized version was available.

Wolf and Lam [WL91] and Kennedy and McKinley [KM92] do similar work, but on complete programs. They reorganize control to improve locality in the inner loops. They also detect a parallel loop, put it in the outermost legal position and tile (i.e., strip mine and interchange) it if it contains spatial locality. Their transformations remove false sharing by improving processor locality.

Two studies focused on reducing false sharing in pages rather than cache blocks. Bolosky et al. [BFS89] eliminated false sharing by coalescing objects into a larger object or padding individual objects to page boundaries, all manually. However, they do not quantify the effect of eliminating false sharing. 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.

7 Conclusion

In this paper we have analyzed the effectiveness of compile-time analysis and shared data transformations in reducing false sharing in explicitly parallel programs. Our results indicate that the static analysis successfully identifies the data structures that cause most false sharing and restructures them to eliminate it, while keeping the negative impact on spatial locality under control. No single transformation is responsible for the false sharing reductions, even within a single program: all are important contributors to improved performance.

The reduction in false sharing misses brought different performance benefits in different regions of the speedup curves. As long as the unoptimized programs experienced speedups with increasing numbers of processors, the transformed versions improved execution time by modest (as small as 2%) or more substantial (up to 58%) amounts. After the point at which the unoptimized programs no longer scaled, most compiler-transformed programs still continued to scale, resulting in more than a doubling of the overall maximum speedups, on average.

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 three grounds. The first is performance. Our particular static analyses led to transformations that were at least as successful as programmer efforts, and sometimes more. The second is portability of the program source across different cache architectures. The third is ease of programming. Hand transformations force the programmer to focus on 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, rather than the semantics of the programs. For our algorithms and transformations, all three benefits were realized with only a 5% increase in compile time.

8 Acknowledgements

We would like to thank J. P. Singh and Josep Torrellas for providing us with many of the programs in our workload, Craig Chambers, Kathryn McKinley, Anne Rogers, Jean-Loup Baer and Dean Tullsen for helpful comments on an earlier draft of this paper and John Bennett for access to additional parallel machine resources.

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, May 1988.

[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.

[BFS89] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

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

[CK88a] D. Callahan and K. 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.

[Egg91] S.J. Eggers. Simplicity versus accuracy in a model of cache coherency overhead. *IEEE Transaction on Computers*, 40(8):893–906, August 1991.

[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*, pages 37–47, 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*, pages 273–289, August 1993.

[GW94] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume I: Architecture, pages 134–143, January 1994.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Specification*. January 1993.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 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*, pages 344–358. Springer Verlag, August 1991.

[JE92] T.E. Jeremiassen and S.J. Eggers. Computing per-process summary side-effect information. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth Workshop on Languages and Compilers for Parallelism*, pages 175–91, August 1992.

[JE94] T.E. Jeremiassen and S.J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 171–180, August 1994.

[Jer95] T.E. Jeremiassen. Using compile time analysis and transformations to reduce coherency traffic on shared memory multiprocessors. Ph.D. thesis, University of Washington, to be published, 1995.

[Ken94] Kendall Square Research. *KSR/Series Princples of Operations*, revision 7.0 edition, 1994.

[KM92] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 276–283, July 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):63–79, March 1992.

[LLG+94] D. Lenoski, J. Laudon, K Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[LT88] R. Lovett and S. Thakkar. The symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 1, pages 303–310, August 1988.

[M. 93] M. Cekleov, et. al. SPARCcenter 2000: Multiprocessing for the 90's. In *IEEE COMPCON*, pages 345–353, February 1993.

[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.

[SGL94] J.P. Singh, A. Gupta, and M. Levoy. Visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, July 1994.

[SHHG93] J.P. Singh, C. Holt, J.L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing 93*, pages 54–65, November 1993.

[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.

[TLH94] J. Torrellas, M.S. Lam, and J.L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[WL91] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.