Using Compile-Time Analysis and Transformations to Reduce False Sharing on Shared-Memory Multiprocessors

by

Tor E. Jeremiassen

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1995

Approved by ______________________________

(Chairperson of Supervisory Committee)

____

____

Program Authorized
to Offer Degree ______________________________

Date ______________________________

University of Washington

Abstract

Using Compile-Time Analysis and Transformations to Reduce False Sharing on Shared Memory Multiprocessors

by Tor E. Jeremiassen

Chairperson of Supervisory Committee: Professor Susan J. Eggers
Department of Computer Science
and Engineering

On bus-based, shared-memory multiprocessors, much of the "unnecessary" bus traffic, i.e., that which could be eliminated with better processor locality, is coherency overhead caused by false sharing. False sharing occurs when multiple processors access (both read and write) different words in the same 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. However, changing the data layout 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.

This dissertation presents a compiler-directed approach to eliminating false sharing. A series of compiler algorithms and a suite of shared data transformations were developed and incorporated into a source-to-source restructurer 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. The algorithms are evaluated on the accuracy of their analysis, as well as their run-time cost.

The effectiveness of the compiler-directed approach is evaluated on a set of ten

explicitly parallel programs. Trace driven simulation shows that the compile-time analysis successfully identifies the data structures responsible for most false sharing misses, and makes appropriate tradeoffs between eliminating false sharing and reducing spatial locality. The reduction in false sharing misses positively impacts both execution time and program scalability when executed on a KSR 2. 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 outperformed programmer efforts to eliminate false sharing in these programs.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

This dissertation would not have come about had it not been for the help of many people. First and foremost I would like to thank my advisor, Susan Eggers. Without her guidance, support and great patience, I surely would have failed in this endeavor.

I am grateful to her, Jean-Loup Baer and Craig Chambers for serving on my reading committee. They gave me valuable feedback that greatly helped improve the final version of this dissertation.

Craig Chambers also helped shape the compiler-related portion of my research, as well as reading and providing comments on numerous drafts of earlier papers. For that I am grateful.

I would like to express my appreciation to the department's lab staff, in particular Paul Barton-Davis, Nancy Johnson Burr, Mark Murray and Renee Reed. They were instrumental in providing me with stable platforms for my research. Later, when our aging Sequent Symmetry suffered several hardware failures at the end of its life cycle they expended much time and energy to resurrect it so that I could salvage my results.

I am indebted to John Bennett of Rice University for providing me with the use of their Sequent Symmetry when ours finally died, enabling me to complete my studies.

My work would have been a lot harder had it not been for the trace generation tool developed by fellow graduate students David Keppel and Eric Koldinger. My discussions with them as well as Dean Tullsen and Radhika Thekkath provided me with much valuable advice and insights.

I would also like to thank J. Scott Penberthy for his friendship, support and inspiration.

I thank my family, in particular my parents Else and Tor Jeremiassen, for their wholehearted support which made the large geographical distances separating us seem so insignificant.

Last, but not least, I wish to express my most sincere gratitude to my wife, Kathleen Shipley Jeremiassen. Her love and support has kept me going through some of my toughest times.

Dedicated to the memory of my mother Else Jeremiassen.

Chapter 1

INTRODUCTION

1.1 Motivation

Shared-memory multiprocessors are considered by many to be an efficient way of obtaining the high performance potential of parallel processing, while still keeping within the traditional programming paradigm. Many of these machines are based on a single bus and have private caches for each processor. Since any given datum may be present in multiple caches, multiple copies of the same word may be updated, thus creating inconsistencies between the processors with respect to the value of the datum. This is the cache coherency problem. To solve this problem shared-memory multiprocessors implement cache coherency protocols that guarantee that writes are performed atomically, and that for each location, the sequence of values is the same for all the processors.

For multiprocessors with simple interconnects, such as a single bus, the simplest coherence solution lies in snooping cache coherency protocols [AB86], two classes of which are write-invalidate and write-broadcast protocols. Write-invalidate protocols maintain coherency by requiring a writing processor to obtain ownership of the data by invalidating all other cached copies, before updating its own. A processor that rereferences invalidated data incurs a cache miss, called an *invalidation miss*, and must obtain the most up to date copy through a cache-to-cache transfer from the most recent writer of that data. In contrast, write-broadcast protocols maintain coherency by requiring the writing processor to broadcast shared writes, so that all caches and memory are kept current with the most up to date value.

One of the major performance bottlenecks of parallel programs executing on these machines is the overhead of maintaining cache coherency, i.e., the additional bus traffic generated by the protocols. This overhead is largely determined by the pattern of memory references to write-shared data. Several studies have documented that the pattern is bimodal [AG88, CGBG88, EK89b]: under *processor locality* [AG88] or migratory sharing [WG89], a single processor makes multiple, consecutive accesses, most importantly, writes, to the words within a cache block, uninterrupted by accesses from other processors; in *inter-processor contention*, multiple processors contend for one or more words within the block, and the number of per-processor sequential writes is very low. Shared memory reference patterns have a bearing on the choice of coherency protocol [EK89b]. Write-invalidate protocols perform well when there is good processor locality, because a processor need only perform a single invalidation for a series of writes. Conversely, write-broadcast does well during periods of inter-processor contention, since the number of bus transactions is reduced.

Because of the increasing discrepancy between processor and memory speeds, reducing the number of cache misses and coherency overhead has become paramount to obtaining good performance. To that end there is a trend in modern multiprocessor design to use very large caches and large cache blocks (for instance the KSR 2 [KSR92] has 32 MB second level caches with 128 byte blocks). Large caches reduce the number of cache misses by reducing the number of conflict and capacity misses. For non-shared data and instructions, large cache blocks exploit spatial locality by prefetching data adjacent to the address that was referenced, thereby reducing first reference misses. Some of this benefit is also realized on cache coherent, shared-memory multiprocessors. However, the larger cache block sizes may also interfere with inter-processor access to data within them, since cache coherency is very often maintained on a cache block basis. Several studies have shown that large cache blocks may often increase coherency overhead, even to the point where it more than negates any benefit of the prefetching provided by the larger cache block size [LYL87, CGBG88, EK89a]. This additional coherency overhead is caused by a phenomenon known as *false sharing* [TLH90, EJ91].

False sharing in multiprocessor caches occurs when the cache block is larger than a single word, and different processors access (read and write) different words in the same cache block. It is called false sharing because, although the words in the cache block are not individually shared, the cache block as a whole is. Since cache coherency is maintained on a cache block basis, the same coherency transactions are incurred as if the words had actually been (true) shared. In a write-invalidate coherency protocol the overhead of false sharing takes the form of additional invalidations when a pro-

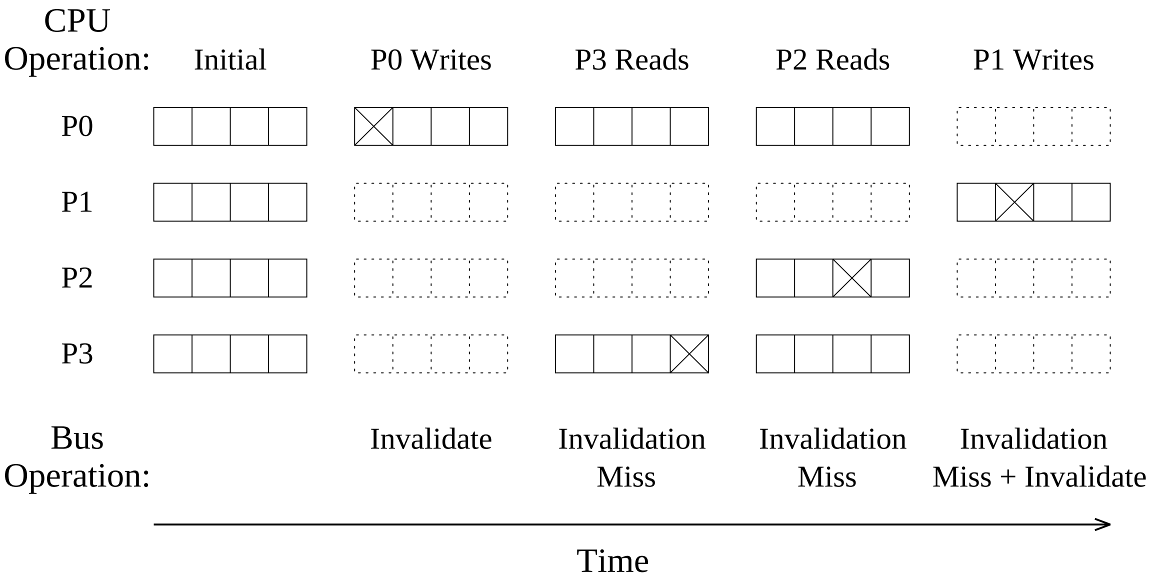


Figure 1.1: Example of false sharing misses.

cessor updates data and invalidation misses when other processors reread (different) data in the invalidated cache block. Thus, instead of exploiting the spatial locality, false sharing may cause a significant loss in processor locality. In a write-broadcast coherency protocol, the additional overhead is due to the additional broadcasts that are performed because the cache block is (falsely) labeled as shared. In this dissertation I will concentrate on architectures that utilize a write-invalidate cache coherency protocol, as this is the approach most commonly used in commercial machines.

False sharing is not inherent in any parallel program, coherency protocol, or multiprocessor architecture. Rather it is the artifact of a mismatch between the layout of write-shared data in memory, the cross-processor memory reference pattern to it, and the memory subsystem architecture. Consider the example shown in Figure 1.1, where four processors are accessing different words in the same cache block in a multiprocessor using a write-invalidate coherency protocol. Initially, a copy of the cache block is resident in the caches of all four processors. Then, when processor P0 performs a write, all copies of the cache block but its own are invalidated. This causes P3 and P2 to incur invalidation misses when they attempt to read different words in that cache block. Lastly, P1 incurs an invalidation miss before invalidating the other copies of the cache block. All of these invalidations and invalidation misses are due to false sharing. Not one is the product of actual data sharing (true sharing) between the processors. If the data had been laid out differently, so that data accessed by one processor had not been allocated together with data accessed by other processors, none of these invalidations or invalidation misses would have occurred.

The amount of false sharing on a multiprocessor varies greatly between programs. However, in some coarse-grain, explicitly parallel applications, misses due to false sharing make up between 40% and 90% of all cache misses (over block sizes ranging from 8 bytes to 256 bytes) and comprise 40% to 80% of total bus utilization [EJ91]. Eliminating false sharing from these programs holds promise for improving performance in three ways. First, the reduction in cache misses reduces running time, as less time is spent servicing the misses. Second, because fewer misses are being serviced, the utilization of the interconnect is reduced. Thus the performance of programs that are communication bandwidth limited improves. Third, the reduction in false sharing improves processor locality, thereby enabling the use of larger cache blocks, so that their prefetching benefits can be exploited, as on uniprocessors.

Manually changing the placement of data in coarse grained parallel programs 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 the research in this dissertation takes the approach of automating the elimination of false sharing. A series of compile-time algorithms that analyze coarse-grained, explicitly parallel programs have been developed and implemented in a source-to-source restructurer [PGH+89]. The algorithms determine which data structures are susceptible to false sharing, where locality may be improved, and which transformations to apply at compile-time, by analyzing a program and computing 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 [JE92]. 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 [JE94]. The third stage performs an enhanced interprocedural, flow-insensitive, summary side-effect analysis [Bar78, Ban79, Mye81a, CK88], with static estimates of execution frequencies, on a per-process basis (based on the control flow determined in stage one) for each phase (determined in stage two).

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 do not share cache lines. The data transformations eliminate false sharing by grouping together data that are predominantly accessed by the same processor, and separating data that are accessed by different processors. This is done in three ways. First, the layout of shared data structures in memory can be changed by physically grouping data together. Second, elements of shared data structures can be placed in processor specific pools of shared memory, while pointers to them are placed in their old locations. Third, shared data objects can be placed in their own cache blocks by padding their size and aligning them on cache block boundaries.

The algorithms and transformations were evaluated on a coarse-grained, explicitly parallel workload using trace driven simulation of a bus-based multiprocessor with a write-invalidate cache coherency protocol, and direct execution on a KSR 2 multiprocessor. The workload contains two different sets of applications. One set is made up of applications from the SPLASH and SPLASH2 benchmarks [SWG91]. These are programs that have undergone extensive hand-tuning to optimize cache performance. As such they represent the best case of programs that one can expect programmers to produce. The remaining applications have not undergone the same optimization process, or have had obvious programmer optimizations undone, and are therefore more likely to represent the average case parallel programming. The simulations and performance measurements produce several results. First, as could be expected, the amount of false sharing varies across the applications in the workload, and is the lowest for the hand-tuned programs. Nevertheless, all programs exhibit false sharing to the degree that it accounts for at least 40% of all cache misses for the largest cache block sizes. Second, no single algorithm nor transformation eliminates false sharing in all applications. However, when used together they are more successful than the programmer directed efforts cited above. Regardless of the amount of false sharing in the applications, they eliminated on average (across all block sizes and for the entire workload) 66% of the false sharing misses. For two programs, one from each set, they eliminated more than 90%. Third, the simulations show that even though 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. Fourth, the reduction in false sharing misses has two effects on run-time performance as measured on a KSR 2: 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 lower execution times, ranging from a modest (2%) to a more sizable (58%) amount.

1.2 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 presents the model of parallel programming of our target workload. It motivates the choice of coarse-grained, explicitly parallel programs, as opposed to medium-grained programs, such as those generated by parallelizing compilers. Furthermore, it describes the features of the model that the compile-time algorithms take advantage of to improve the precision of the analysis.

Chapter 3 is the heart of the dissertation. It describes the compile-time analysis, the algorithms and their implementation. The goal of the analysis is to approximate the cross-processor memory reference pattern of a program, by computing which data objects each process may read and which it may write. This is done in three stages of analysis. The first stage computes the possible flow of control for each process and groups processes into families based on their control flow. The second stage uses barrier synchronization points to determine which portions of a program can execute in parallel and which cannot. The third stage employs a standard, interprocedural side-effect analysis with some modifications to enhance the accuracy of the final approximation of the memory reference pattern. Results that evaluate the accuracy of the algorithms are presented, as well as an evaluation of their run-time cost.

Chapter 4 describes the set of data transformations used to restructure vulnerable data structures to eliminate false sharing. There are three transformations: *group and transpose*, *indirection*, and *pad and align*. The chapter details when each transformation can be applied, how it changes the data layout to eliminate false sharing, as well as its impact on spatial locality. This chapter also describes the heuristics that are used to determine which data structures are susceptible to false sharing and which transformations to apply.

Chapter 5 describes the methodology used to obtain the results presented in this dissertation. The applications were first analyzed by a source-to-source restructurer in which the compile-time analysis algorithms had been incorporated, and targeted data structures were transformed. The amount of false sharing was measured using trace driven simulation. A multiprocessor simulator was modified to keep track of false sharing and count the number of resulting cache misses. The run-time impact of the transformation were measured by direct execution on a KSR 2 multiprocessor.

Chapter 6 presents results that evaluate the effectiveness of automatically eliminating false sharing. The first set of results shows that the effect of the false sharing problem varies greatly from one program to another. Nevertheless, for large cache block sizes the number of false sharing misses make up about half of the total number of cache misses for all programs, and up to 90% for some. The second demonstrates how effectively the compiler-directed restructuring eliminates false sharing misses. The results show that, regardless of the amount of false sharing in the original program, on average across the workload at least 60% of the false sharing misses were eliminated, and for some programs as much as 90% to 100%.

In order to evaluate the relative effectiveness and contribution of the individual compile-time analysis algorithms and transformations in detecting and eliminating false sharing, results are broken down by transformation and aggressiveness of the static analysis. These results show that no single algorithm or transformation was universally applicable to all applications. Rather, it was their combination that reduced false sharing misses.

Since the transformations that eliminate false sharing also may enlarge the size of the data in the program their impact on spatial locality is also quantified. This gives the background for examining the tradeoff of improving processor locality at the cost of spatial locality.

In addition to the simulation studies the programs were also run on a shared-memory multiprocessor in order to gauge the overall run-time benefit of transforming data structures in the programs. Results are presented that show the speedups of the two versions (before and after transformations) on the KSR 2 [KSR92].

Chapter 7 discusses related work. It can be roughly divided up into two groups. One is work that attacks the problem of false sharing, but does so using approaches different from ours. The other is work on static analysis algorithms that is not necessarily related to false sharing.

Chapter 8 concludes this dissertation. It summarizes the research contributions as well as discusses the performance impact of eliminating false sharing. Possible future research directions are also outlined.

Chapter 2

MODEL OF PARALLEL PROGRAMMING

The target workload of our compile-time analysis algorithms and data transformations are coarse-grained, explicitly parallel programs for shared-memory multiprocessors. The programs are written in traditional high-level languages, in this case C, that use operating system calls to implement process creation and synchronization primitives. Examples of such programs can be found in the Stanford SPLASH and SPLASH 2 application suites [SWG91]. The programs solve a variety of problems, including n-body and particle simulation, circuit layout and circuit simulation. They currently execute on small to medium scale, bus and ring based multiprocessors, both commercially (e.g., Sequent Symmetry [LT88], KSR 2 [KSR92]) and in research environments (e.g., DASH [LLG$^+$92]). A new generation of these machines is emerging. Examples are the SPARCcenter 2000 [FCS$^+$93] and the SGI Challenge [GW94].

```
private int pid;
shared barrier_t MyBarr1, MyBarr2, MyBarr3;
shared int NumProcs;
:
for ( pid = 1; pid < NumProcs; pid++) {
    if ((rc = fork()) == 0) {
        Work();
        exit(0);
    } else  if (rc == -1)
        error();
}
MasterWork();
:
```

```
Work() {
    while (converged != 0) {
        SubPart1(pid);
S1:     Wait_Barrier(&MyBarr1);
        SubPart2(pid);
S2:     Wait_Barrier(&MyBarr2);
S3:     Wait_Barrier(&MyBarr3);
    }
}
```

```
MasterWork() {
    while (converged != 0) {
        SubPart1(pid);
S4:     Wait_Barrier(&MyBarr1);
        SubPart2(pid);
S5:     Wait_Barrier(&MyBarr2);
        converged = TestConverged();
S6:     Wait_Barrier(&MyBarr3);
    }
}
```

Figure 2.1: Sample program code.

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. The granularity of parallelism in these programs is very coarse, on the level of an entire process. Processes are typically only created in the beginning of the program and do not rejoin until close to program termination. Thus each thread of control executes over a large portion of the program and its control flow can be quite complex. This contrasts to programs in the medium grained parallel world, many of which are written in a dialect of Fortran and are the product of parallelizing compilers. Here processes (or light weight processes) are automatically created and destroyed at multiple places in the program. Parallelism is contained within loop nests, and the granularity of parallelism is on the order of sets of iterations.

For the purposes of the compile-time analysis, transformations and the evaluation of their effectiveness, several simplifying assumptions are made about the model of programming. First, the general model of pointers in C is restricted in the following three ways: (1) except for reference parameter pointers[1], pointers can only point to heap-allocated data structures, they cannot point to named variables; (2) a pointer only points to data objects of its declared type; (3) there is no pointer arithmetic. Second, it is assumed that the number of processes equals the number of processors. The effect is that the memory reference pattern across the processors is the same as the memory reference pattern across the processes. Therefore, it frees the compiler from considering how the cross-processor memory reference pattern would change with different process allocations, as well as the problem of choosing a good allocation. Third, it is assumed that processes do not migrate between processors, as the cache effects from such migrations may mask coherency misses and perturb the results of the performance measurements. More specifically, whenever a process migrates to another processor the new cache is cold. Therefore, the new processor will initially incur a number of compulsory misses to bring in the data set of that process. However, some of these misses will be to data that is actively (falsely) shared and would have appeared as coherency misses (true or false sharing misses) had the process remained on its original processor.

There are several features of the coarse-grained model that the compile time analysis uses to facilitate and improve the accuracy of approximating the cross-processor memory reference pattern. First, processes are created explicitly, e.g., using a *fork()* system call (as illustrated in Figure 2.1). This indicates where parallelism begins, and

[1] The static analysis can detect a large subset of the cases when parameters of pointer types actually function as reference parameters.

thus furnishes the algorithms with a starting point for their analysis. In all of the the programs in our workload, the processes are created in a single loop that iterates over the number of the processes.

Second, the processes all have the same name space, that is, the same variable and procedure names are visible to all processes. However, the address space of each process is divided into a shared and a private partition, with only the shared partition being kept coherent across the processes. Each variable name maps into either the private or shared partition, but must map into the same partition for all the processes. From a programmer perspective, the only differences between processes then are due to different values of private variables (*pid* in Figure 2.1 is an example of such a variable) or system calls that return process specific values (process identifiers). Variables that are guaranteed to have different values across the processes are denoted *process differentiating variables*. Private (to each process) variables may be assigned different values for different processes in several ways, three of which can be detected by the static analysis. One is that they can be assigned a value from the above mentioned system call. Alternately, they can be assigned the value of some monotonically increasing global variable (or simple function thereof) inside a critical section that is executed once by every process. Finally, they can be assigned the value of an induction variable in the process creation loop (as illustrated in Figure 2.1). The algorithms use these variables to differentiate processes based on differences in their control flow, and to detect per-process access of shared data structures, in particular when shared arrays are statically partitioned across the processes.

Third, process synchronization is performed using *global barriers*. They are used in programs running on shared-memory multiprocessors as a (relatively) inexpensive mechanism to enforce large sets of cross-process data dependencies that otherwise would have to be enforced by a large number of locks. When the control flow of a process reaches a barrier, it must wait until the other participating processes also reach that barrier. (The barrier is a function of the parameter in the barrier synchronization call; thus the same barrier may be reached at different statements). A barrier is global if all the processes in the program participate in the synchronization at that barrier.

Barriers are typically invoked via library or operating system calls. On bus-based multiprocessors they are commonly implemented using a structure that contains a shared counter that counts up to the number of processes participating in the barrier before the processes are allowed to continue [BP89]; this structure is called the *barrier variable*. Upon reaching the barrier, each process increments the counter and waits until the counter has reached the preset limit (which for global barriers is equal to the total number of processes), at which time the counter is reset and the processes continue execution. A barrier variable is typically passed by reference in the barrier synchronization call. Note, if different sets of processes in the same program were to pass different barrier variables for a global barrier synchronization event, the processes would be left spinning at the barrier, since neither counter would reach the limit required for synchronization. In this case no synchronization would take place, and the program would be deadlocked. Figure 2.1 shows an example of the use of barriers in a program.

Last, in our model (and in all the programs in the workload) locks are only used to enforce mutual exclusion, i.e., they serialize access to critical sections. In our false sharing analysis accesses to locks are treated just like accesses to the shared data they protect. No other synchronization primitives are used, in particular these programs do not use event variable synchronization (post, wait and clear).

2.1 Workload

The workload consists of ten parallel applications. Of these programs, four (LocusRoute, Mp3d, Pthor and Water) are taken from the Stanford SPLASH benchmarks [SWG91], and three (Fmm [SHHG93], Radiosity and Raytrace [SGL94]) from the Stanford SPLASH2 benchmarks. They are well known programs in which considerable programming effort was expended to improve data locality, including eliminating false sharing. They were included in the workload to see if the static analysis could improve upon programmer efforts to reduce false sharing. As with other heavily hand-optimized programs, it was expected that the compiler would find fewer opportunities to apply transformations that improve the performance of the programs. The results reflect this. In Maxflow [Car88], Pverify [MDWSV87] and Topopt [DN87] the programmers had made no attempt to improve locality, because of the added complexity of (multiprocessor) cache-conscious programming.

Each of the SPLASH2 programs contained several easily identifiable data structures whose elements had been organized by processor. These transformations were

Table 2.1: Workload used in this study. Available versions are (U)nptimized, (C)ompiler-optimized, or (P)rogrammer-optimized.

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

undone to produce unoptimized versions of these programs; no other changes were made. In addition, Topopt was manually optimized to eliminate false sharing in [EJ91]. This version is included in the programmer-optimized workload.

Each application is discussed separately below.

2.1.1 Fmm

Fmm implements a parallel adaptive fast multipole method to simulate the interaction of a system of bodies (n-body problem) with non-uniform distribution. For each iteration of the algorithm, the program builds a hierarchical representation of the set of bodies to simplify the calculation of interactions between sets of distant bodies.

The main loop of the program consists of six phases that are separated by global barrier synchronizations. Locks are used to guarantee mutual exclusion to the cells of the hierarchical representation, shared memory allocation and input/output functions.

The input consisted of two interacting galaxies with a total of 16K particles.

2.1.2 LocusRoute

LocusRoute is a commercial quality standard cell router. It routes wires based on a cost function to minimize the layout area. The routing is done twice to reduce the impact of the order in which the wires are routed. The circuit is divided into spatial regions, each with a task queue and a preferred process assigned. The wires are then partitioned across the regions according to their point of origin and enqueued. The processes take wires off their preferred task queue and proceed to route it. If the queue is empty, the process may use other processor's preferred task queues.

Locks are used to guarantee mutual exclusion to the task queues, cost array, routes and global counters. A single global barrier is used between the two iterations.

The input to LocusRoute was a circuit with 3817 wires.

2.1.3 Maxflow

Maxflow implements a parallel algorithm to find the maximum flow from the distinguished source to the sink in a directed graph with edge capacities. The algorithm consists of two steps. First, excess flow is pushed along the estimated shortest paths towards the sink node. Then, remaining excess flow is removed at unbalanced nodes by pushing it back towards the source node.

Six global barriers divide the program into seven phases. Locks are used to guarantee mutual exclusion to the vertices of the graph, a global task queue, and a global counter.

The input to Maxflow was a directed graph with 2000 vertices and 6000 edges.

2.1.4 Mp3d

Mp3d does rarefied flow simulation. It simulates movement of individual molecules at hypersonic speeds as they collide with each other, with the boundaries of the physical domain, and with the model object(s) being studied. The set of molecules are partitioned across the processes, meaning that the space they move through is shared and contended for.

Global barriers are used to divide the main loop into six phases. The use of locks to guarantee mutual exclusion to prevent some minor race conditions is optional. No major perturbation of the final result is expected if locks are not used; however, in this study the program was compiled with locks.

All results for Mp3d are for a $10,000$ molecule problem.

2.1.5 Pthor

Pthor is a parallel, distributed time, event-driven simulator that uses a version of the Chandy-Misra [CM81] distributed time algorithm to simulate the behavior of a digital logic circuit, given a description of the circuit and its inputs.

Pthor utilizes distributed work queues. Each element of a circuit is assigned a preferred process work queue when initialized, and is placed on that queue whenever it receives a new input event. Deadlocks occur whenever all the work queues are empty. At this point, a deadlock resolution routine is executed to generate events to activate more elements.

Locks are used to guarantee mutual exclusion to each circuit element and process work queue. Global barriers are used, but only during the resolution of deadlocks.

A circuit with $5,060$ circuit elements was used as the input.

2.1.6 Pverify

Pverify is a parallel circuit verifier that reads in the logic description of two circuits and determines if they are functionally equivalent. The program works in two phases. Phase one computes the sets of inputs that generate outputs of one or zero for the first circuit, using a depth first search of the graph representation of that circuit. In phase two it simulates the second circuit using the inputs from the first phase, comparing the outputs of the second circuit with those of the first. Both phases utilize a dynamic, stack scheduling scheme (as opposed to queue scheduling).

No barriers are used in Pverify, but locks are used to guarantee mutual exclusion to work queues and global counters.

Two small circuits were used as inputs, both with one output and 32 inputs. One consisted of 63 gates, the other of 153.

2.1.7 Radiosity

Radiosity computes the global illumination in a scene containing diffusely reflecting surfaces. That is, it computes the amount of light that hits every point of each surface as a function of the light reflected from all other surfaces in the scene. This is done by dividing each surface into patches such that the radiosity of each patch is uniform across its surface. Then the radiosity of each patch is computed as a linear combination of the remaining patches in the scene. This is repeated until the global solution converges. To minimize the number of terms in the linear combination, the algorithm uses a hierarchical method to subdivide patches, so that the largest size patches that do not impact the accuracy of the solution is used in each pair of patch interactions.

Parallelism is exploited by considering both the set of interactions of a single patch as well as each patch-patch interaction to be possible parallel tasks. Each processor implements its own task queue, which is initialized with a set of patches from the scene. As the number of subdivisions necessary for each patch varies, load balancing is provided by allowing processors to steal tasks from other processor's task queues.

Radiosity uses global barriers to separate the iterations of the algorithm. Locks are used to guarantee mutual exclusion to each patch as well as a global accumulator.

The input to Radiosity was a scene with 174 input polygons.

2.1.8 Raytrace

Raytrace renders a three-dimensional scene onto a two-dimensional image plane using optimized ray tracing. It works by firing primary rays from the viewpoint into the scene. As rays encounter objects, new rays are created. They are reflected and refracted through the objects as appropriate, and reflected to each light source to determine its contribution. A hierarchical grid is used to traverse the scene effectively.

Parallelism is afforded across rays. To maintain good load balancing, each processor implements its own task queue, but can steal tasks off the queues of the other processors. Raytrace uses one global barrier to synchronize the processes following task queue initialization. Locks are used to guarantee mutual exclusion to task queues, rays and global memory free list.

The input to Raytrace was a 10 MB scene of a car on a checkerboard floor, with a 512×512 pixel image.

2.1.9 Topopt

Topopt is a CAD tool that does topological compaction of MOS circuits. It uses a simulated annealing algorithm applied to gate and signal interchanges to minimize total area. The circuit is represented as an array, with each process having a disjoint window into the array. The windows are dynamic and move across the array in order to increase the number of possible logic gate interchanges.

No locks are used in the program. However, global barriers are used to divide up the main loop of the program into four separate phases. One of the processes is designated the master process and checks for convergence following each iteration of the loop.

The input to the program was a circuit with 561 gates.

2.1.10 Water

Water is a parallel N-body simulator that was originally a sequential program written in Fortran as part of the Perfect Club benchmarks [BCK+89]. It evaluates the forces and potentials among water molecules in liquid state. The molecules are statically partitioned across the processes according to their location in the shared data structure (not in the simulated space).

The main processing loop of the program is divided into six phases by global barriers, while locks guarantee mutual exclusion to individual molecules as well as a set of global accumulators.

The input had a problem size of 343 molecules.

Chapter 3

COMPILE TIME ANALYSIS

3.1 Introduction

This chapter contains the algorithmic contribution of the research described in this dissertation, namely, the details of the compiler algorithms that analyze parallel programs to determine which data structures are susceptible to false sharing and where locality may be improved. It also presents results that attest to the accuracy of these algorithms.

The goal of the compile-time analysis is to approximate the cross-processor memory reference pattern and determine when it interferes with the layout of shared data in memory. Since there is only one process per processor, by assumption, this can be done by computing the memory reference pattern of each of the processes in the program. In particular, since false sharing is a product of different processes accessing different portions of shared data, the analysis focuses on detecting differences in the memory reference pattern between the processes.

The per-process memory reference pattern is approximated by computing the set of data elements that each process p may read (USE_p) and write (MOD_p), the frequency with which they are accessed, and when they are accessed in relation to the execution and memory accesses of the other processes. The compiler analysis achieves this in three separate stages: per-process control flow analysis, non-concurrency analysis based on barrier synchronization, and an enhanced, interprocedural, flow-insensitive side-effect analysis.

Per-process references to shared data may 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.

---

[1] As mentioned in chapter 2, process differentiating variables are private variables that have values

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 enables the analysis to approximate the memory access pattern of distinct phases of a program (between barriers), and, more importantly, when the pattern shifts. When coupled with static estimates of execution frequencies, it estimates the dominant sharing pattern in the program and restructures for that pattern.

The remainder of this chapter details the per-process control flow analysis (Section 3.2), the non-concurrency analysis (Section 3.3), and the side-effect analysis (Section 3.4).

3.2 Process Family Control Flow Analysis

In programs written (or generated by parallelizing compilers) in a medium-grained, parallel model of programming, the parallelism is contained within regular loop nests. The control flow within each loop nest is very simple and the cross-processor data distribution (and memory access pattern) can be computed as functions of the loop induction variables. In contrast, as discussed in Chapter 2, the programs in our target workload are written in a coarse-grained model of parallel programming, where the parallelism spans very large portions of the programs and neither the memory access pattern nor the control flow of the processes is simple and regular. Therefore, to facilitate the approximation of the cross-processor memory reference pattern it is first necessary to recognize the control flow of each of the processes. This is the task of the first stage of analysis. It implements a general technique that recognizes, separates and groups the processes according to their flow of control over arbitrary control flow graphs.

In general, several processes may execute the same piece of code, i.e., have the same control flow graph, while still accessing different sections of shared data. Examples of this are programs that use a central work list that is accessed by multiple processes performing identical operations on different elements, and programs that use private variables in array subscript expressions to access different sections of shared arrays. Therefore, each control flow graph produced by this algorithm is assumed to correspond to a *family* of processes; only if it can be explicitly shown to only contain

---

that vary across the processes and are invariant throughout the lifetime of the processes. *pid* in Figure 2.1 is an example. They are discussed in detail in Section 3.2.

a single process is it marked as such. Two process families may execute large sections of identical code. However, as long as they have at least one code segment that is dissimilar, they are considered to be different process families.

In order to compute the control flow for process families, it is necessary to determine two types of control points in the program. One is the set of locations that initiates separate process families, i.e. creates parallelism. By using a coarse-grained model of parallel programming with explicit process creation, this set of control points is easily determined. The second is all points in the program where the control flow of process families diverges.

To detect these points of divergence, the algorithm must identify which control statements[2] depend on a *process differentiating predicate* (*PDP*) for which the following two conditions hold: (1) there is at least one process for which the predicate holds true, and at least one for which it is false, i.e., the predicate is the mechanism by which processes are differentiated; (2) the truth value of the predicate for a process is invariant throughout its lifetime. The latter condition guarantees that the processes in a process family will always take the same branch at the control statement in question; and the nodes dominated by the control statement along the other branch can be eliminated from that process family's control flow graph. The values of the private variables of the process family are therefore constrained by the predicate and the truth value corresponding to the process family's path at the control statement.

The process family control flow algorithm will be introduced by presenting the algorithm that computes the per-process control flow *intraprocedurally* (globally), assuming that the PDPs in the program have already been identified and marked. This is followed by an explanation of how the PDPs are detected. The intraprocedural algorithm is then extended to the interprocedural case, and finally the results of applying the entire control flow algorithm to the workload are presented.

3.2.1 Intraprocedural Control Flow Analysis

The algorithm starts out with the conservative assumption that all the processes in the program may execute across the entire parallel portion of the program starting at the points of process creation. Thus all processes initially belong to a single process

[2] The discussion assumes that a control statement has at most two paths leading from it. The approach can easily be extended to other cases, such as switch statements.

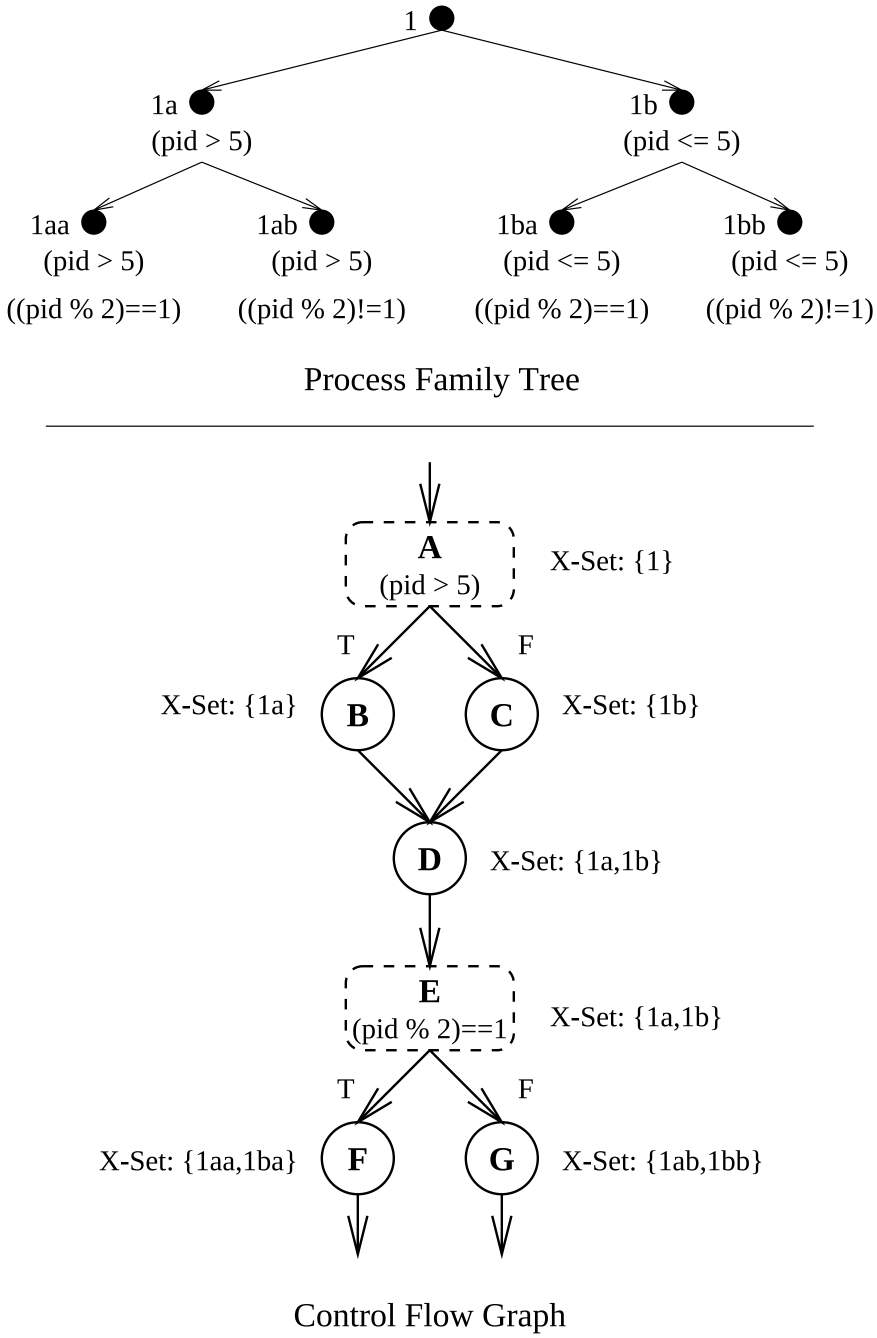


Figure 3.1: Illustration of how the control flow analysis algorithm propagates information and tags nodes in the control flow graph.

family.

Since no points of divergence have been considered, the initial process family starts out unconstrained by any predicates. The initial process family is propagated forward from the point of process creation along the edges of the control flow graph until it reaches the first point of divergence. Since the process family is unconstrained, it is necessary to assume that at least one process will take the path that corresponds to the PDP, call it Π, being *true*, the remainder will take the path that corresponds to the PDP being *false*. Since the processes in the process family no longer have the same control flow, the process family must be split; the original process family is partitioned into two sub-families. The processes of one sub-family are assumed to take the *true* path, and $< \Pi, true >$ is added to its set of constraints. The processes of the other sub-family take the *false* path, and it has $< \Pi, false >$ added to its set of constraints. If the PDP consists of either a test for equality or inequality of a process differentiating variable, one of the resulting process families will contain only a single process, and is marked as such. The two new process families are then propagated forward over the control flow graph starting at the nodes following the point of divergence. When the remainder of the points of divergence are encountered, a process family is only split if it contains multiple processes and its constraints cannot predict the path that its processes will take at these control statements. If the path of a single-process process family cannot be predicted at a point of divergence, the algorithm must conservatively assume that the process family may take either path and therefore propagates it forward along both.

The hierarchy of process families is represented using a binary tree[3], where each node represents a process family or sub-family. When a process family is split, the new sub-families are added as children to the node in the tree that represents the process family being split. Only process families that are represented by leaf nodes may be split or propagated. However, the sets of process families associated with the nodes in the control flow graph are not updated after each split of a process family. Therefore these nodes may contain process families that are represented by interior nodes in the tree. In order to prevent propagating or splitting these process families, they are considered place-holders for the set of process families represented by their leaf descendants. These process families are then propagated and/or split instead. In addition to the pointers necessary to implement the tree, each process family is

[3] A binary tree is sufficient since, by assumption, no point of divergence has more than two paths emanating from it.

associated with two data structures: a flag that indicates whether it represents a single process or multiple, and a list of constraints on the values of private variables.

The data structure associated with each node in the flow graph is a set of process families that execute that node. This set is denoted the X-Set (for eXecute). It is initially empty for all nodes in the control flow graph, but at the conclusion of the algorithm will contain the set of process families that may execute each node. The algorithm starts by initializing the X-Set of the start node with a new, unconstrained process family.

Figure 3.1 will be used to illustrate the algorithm. It shows a portion of a control flow graph with two PDPs, one at node A, the other at node E. In the figure, each node is tagged with the final X-Set information computed by the algorithm. The initial process family, here denoted (1), is propagated forward until it reaches node A. Since A contains an if statement with a PDP and process family (1) is unconstrained, the process family is split into two process families, denoted ($1a$) and ($1b$). Process family ($1a$) adds the constraint `(pid > 5)`, and is propagated forward along the true branch of the if statement (to node B), while process family ($1b$) has the constraint `(pid <= 5)` added, and is propagated along the false branch (to node C). These process families are then propagated forward along the nodes in the control flow graph until they reach the PDP at node E. Since neither of the constraints of the process families can predict the path taken at E, both process families are split, resulting in a total of four process families (($1aa$), ($1ab$), ($1ba$), and ($1bb$)). The appropriate constraints are added to each process family (`((pid % 2) == 1)` to ($1aa$) and ($1ba$), and `((pid % 2) != 1)` to ($1ab$) and ($1bb$)). Then, the process families are propagated along the corresponding branches from E (($1aa$) and ($1ba$) to node F, and ($1ab$) and ($1bb$) to node G).

When the algorithm terminates, each node of the control flow graph is marked with the set of process families that may execute it. Any node that is marked with a process family that is an internal node in the process family tree is considered to be marked by that process family's leaf descendants. For example, in Figure 3.1 the X-Set of node B in the control flow graph is shown to be $\{1a\}$. Since, the leaf descendants of node $1a$ in the process family tree are $1aa$ and $1ab$, the X-Set of node B is considered to be $\{1aa, 1ab\}$.

```
IntraSeparate(Func,StartProcFamSet)
Function_t Func; ProcFam StartProcFamSet;
--
FlowGraph     G;
Node          N,M,Start;
ProcFam       PF,PFTrue,PFFalse;
G = Func.flow_graph;
set X-Sets for all nodes in G to empty;
set X-Set(G.StartNode) to StartProcFamSet;
while changes to any X-Set occur do
  for each node N in G in depth first order do
    for each predecessor M of N in G do begin
      if (M contains a control statement with a PDP)
        branch = which_branch(N,M);
        for each PF in leaf_nodes(X-Set(M)) do
          truth_value = evaluate M.PDP with PF.constraints;
          if (truth_value != undetermined)
            if (branch and (truth_value == true))
              add PF to X-Set(N);
            else if (not branch and (truth_value = false))
              add PF to X-Set(N);
            end if;
          else /* M.PDP could not be eval'ed with PF.constraints */
            Split PF into PFTrue and PFFalse;
            PFTrue.constraints = {<M.PDP,true>} + PF.constraints;
            PFFalse.constraints = {<M.PDP,false>} + PF.constraints;
            if (branch)
              add PFTrue to X-Set(N);
            else
              add PFFalse to X-Set(N);
            end if;
          end if;
      else
        add leaf_nodes(X-Set(M)) to X-Set(N);
      end if;
    end for;
  end for;
end while;
```

Figure 3.2: Intraprocedural algorithm for computing control flow graphs on a process family basis.

The algorithm is shown in Figure 3.2[4]. It uses an iterative approach, visiting each node of the flow graph (line 5), beginning with the root, repeatedly until no change occurs in the X-Set of any node (line 4). For each node visited the algorithm considers each of the node's predecessors (line 6). There are two cases:

- If the predecessor is not a point of divergence, every process family that can execute it can also execute the current node. Therefore, all the process families that are either leaf nodes in the process family tree and members of the predecessor's X-Set, or leaf descendants (in the process family tree) of process families that are members of the predecessor's X-Set are included in the X-Set of the current node (line 28). For example, in Figure 3.1 node E inherits the X-Set of node D.

- If the predecessor is a point of divergence, i.e., it contains a control statement with a PDP (line 7), each process family that can execute it must be examined to see if it can execute the current node (line 9). Every process family, PF, that is in the predecessor node's X-Set and is a leaf node in the process family tree, or that is a leaf descendant of a member of the predecessor node's X-Set is examined for addition to the current node's X-Set. First, the PDP is evaluated in the context of the variable constraints of the process family (line 10). If the PDP evaluates to true or false, **and** if the branch (true/false) leading to the current node matches the result of that evaluation (lines 12 and 14), it is added to the current node's X-Set (line 13 and 15) (not illustrated in Figure 3.1, because the two PDPs, in nodes A and E, are different). In the case where the constraints associated with the process family are insufficient to evaluate the PDP, it is split in two as described above (lines 18–25). This happens for process family 1 at node A and process families $1a$ and $1b$ at node G in Figure 3.1.

3.2.2 Identifying PDPs

The process differentiating predicates are identified in the following way. Recall that the first condition for a predicate to be a PDP requires that it evaluates to true for some processes and false for others. The only variables that can have different values

[4] A less efficient version (no longer used) was described in [JE92]

across processes are private variables. Therefore, to be a PDP, the predicate must use private variables. The static analysis recognizes three cases of when private variables have values that vary across the processes. First, a private variable may be assigned a value from a system call that returns process specific values (process identifiers). Second, each process may have a private variable assigned a value of an induction variable for a loop that creates processes, one for each value of the induction variable. Third, a private variable can be assigned the value of a shared variable which is then incremented or decremented, all inside of a critical section that does not contain any control statements and is executed once for each process. The shared variable must not otherwise be defined in the parallel portion of the program.

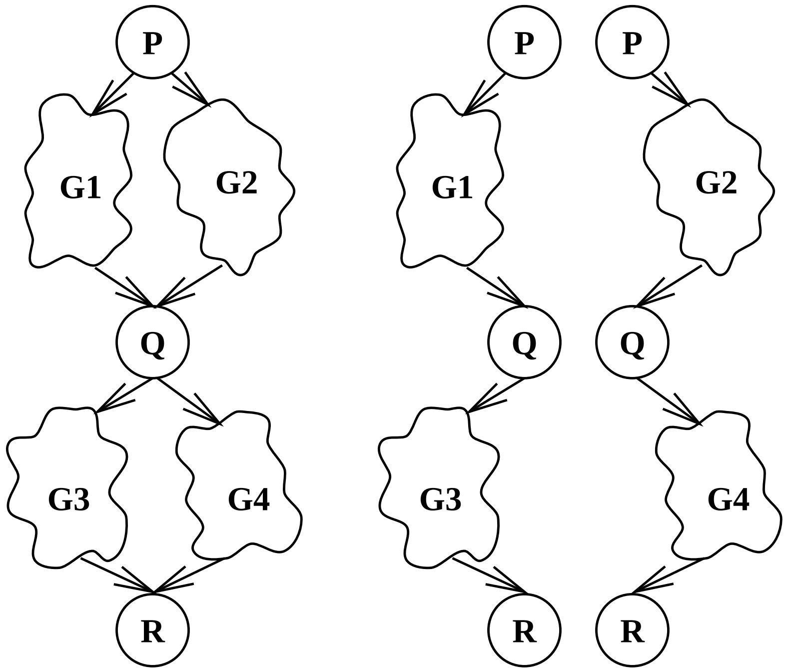
The second condition for PDPs is that the truth values be invariant with respect to the processes. If they were not, the algorithm would erroneously eliminate nodes from the process family control graphs (when the truth values of the predicates change) that the processes actually execute. To guarantee invariance, only predicates whose program variables (PDVs) are not redefined after their first use are considered.

Based on this, a PDP can be defined as a predicate on a set of variables where at least one variable is private, is not redefined after its first use, and has a value that varies across the processes. The remainder of the variables are either constants or are not redefined after their first use.

As an example, consider the control flow graph in Figure 3.3(a). Given that P and Q are if-statements whose predicates are equi-partitioning PDPs, i.e. they impose the same partition on the set of processes, and that there are two sets of processes that execute $G1$ and $G3$, and $G2$ and $G4$, respectively, the algorithm will construct the two control flow graphs in Figure 3.3(b).

In searching the list of constraints for each process family (Figure 3.2, lines 10 and 12) the algorithm only tests for syntactic equality between the PDP and a constraint, following a simple normalization of the order of operators and operands. The normalization makes sure that equivalent PDPs that differ only with respect to order and relational operators are treated as if they are identical. The operands themselves are not altered. Without such normalization, predicates such as `(pid < 5)` and `(5 > pid)` would be treated as being different. However, predicates, such as `(pid > 5)` and `(pid >= 6)`, remain different. The algorithm does not attempt to determine in general, beyond what is described above, if two PDPs are *equi-partitioning* with respect

```
if (Pid == 0) {
   statements;
} else {
   statements;
}
if (Pid == 0) {
   statements;
} else {
   statements;
}
statement;
..
```



(a) (b)

Figure 3.3: Original flow graph and flow graphs for process families after correct PDP identification

to the processes, that is, they impose the same partition on the set of processes. To do so would require a theorem-prover, something that would be prohibitively expensive.

Inaccuracies in constructing the control flow graphs can be introduced in two ways. One is when the algorithm fails to recognize PDPs. This causes a more conservative estimate for the control flow graphs of the affected process families to be made. Figure 3.4 shows the control flow graphs that would be constructed from Figure 3.3(a), given that the algorithm could recognize P, but not Q. (Should neither P nor Q be recognized as PDPs, the resulting control flow graph will be that of Figure 3.3(a)). Overestimating the control flow graph for a process family inflates the sets of data objects that are read and written by its processes, since code they will not execute is included during side-effect analysis. This results in a corresponding overestimation of process contention, since identical data accesses now appear for several process families. The consequence is that data that in actuality may not be shared appear as true shared. This would prevent any false sharing of these data structures from

```
if (Pid == 0) {
   statements;
} else {
   statements;
}
if (is_equal(Pid,0)) {
   statements;
} else {
   statements;
}
statement;
..
```

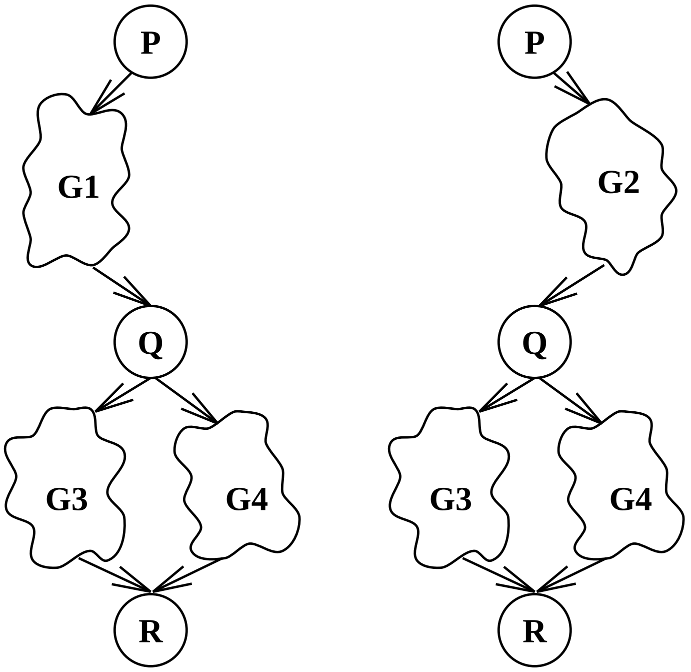


Figure 3.4: Flow graphs for the process families when a PDP (Q) is not recognized.

being detected, and therefore any transformations that eliminate false sharing would be precluded.

The second inaccuracy occurs when the algorithm fails to recognize that a PDP is equi-partitioning with another PDP that has already been considered, and generates a *ghost* process family that does not correspond to any of the processes of the program. Figure 3.5 shows the result of the analysis if the algorithm fails to detect that P and Q are equi-partitioning. Overestimating the number of control flow graphs (process families) also increases the estimated processor contention. The ghost process families access the same data as other *real* process families, thus implying sharing that does not occur in the real program.

3.2.3 The Interprocedural Case

The intraprocedural analysis described in the previous section can be extended to the interprocedural case by using an extended control flow graph that treats procedure calls as flow edges. However, the size of the extended flow graph renders this approach expensive in both time and space. Instead I present an approach that considers each

```
if (Pid > 0) {
   statements;
} else {
   statements;
}
if (Pid >= 1) {
   statements;
} else {
   statements;
}
statement;
..
```

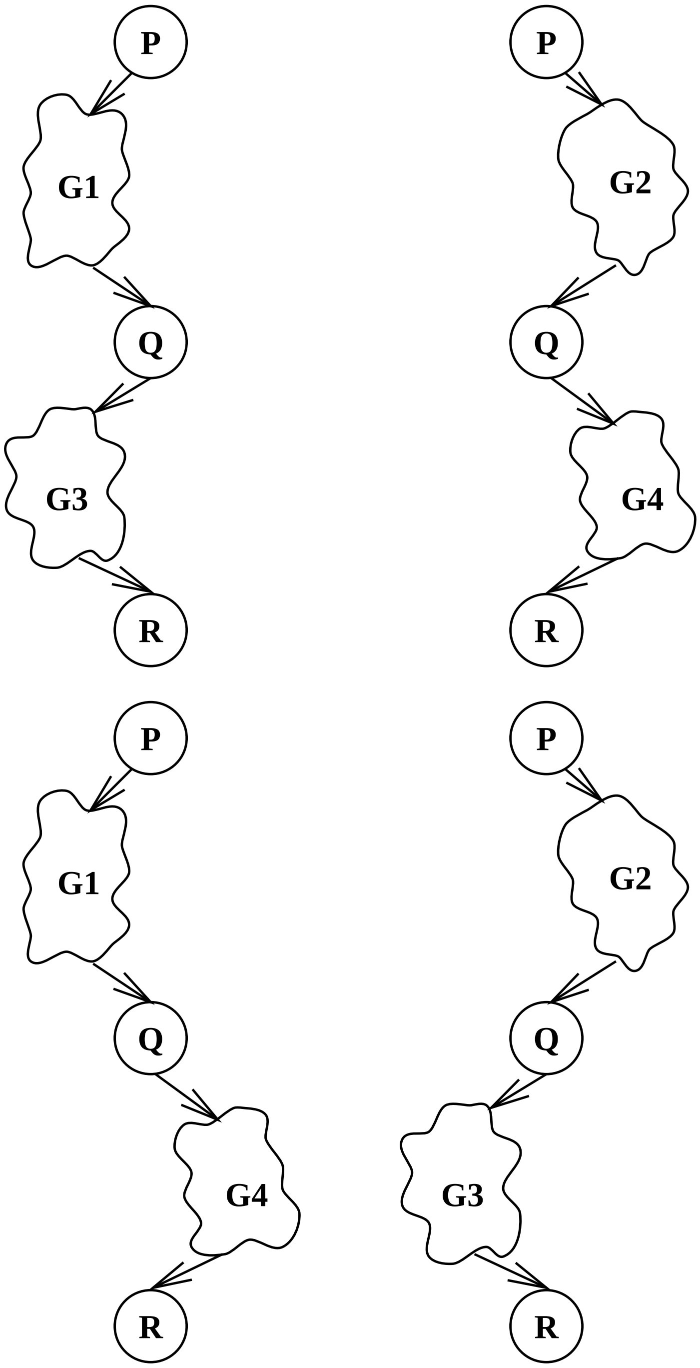


Figure 3.5: Flow graphs for the process families when the equi-partitioning of P and Q is not recognized.

procedure separately in calling order.

```
InterSeparate();
--
 1. for each strongly connected component scc in the call graph do
 2.   replace all nodes in scc by a single placeholder;
 3. end for;
    /* traverse the calling graph in topological (calling) order */
 4. for each node N in the call graph in topological order do
 5.   if (N is not an scc placeholder)
 6.     set startset to empty;
 7.     for each caller C to N do
 8.       add C.procfamset to startset;
 9.     end for;
10.     IntraSeparate(N,startset);
11.   else
12.     while changes to any procfamset occur do
13.       for each function Fscc represented by N do
14.         set startset to empty;
15.         for each caller C to Fscc do
16.           add C.procfamset to startset;
17.         end for;
18.         IntraSeparate(Fscc,startset);
19.       end for;
20.     end while;
21.   end if;
22. end for;
```

Figure 3.6: Interprocedural algorithm for computing control flow graphs on a process family basis.

Figure 3.6 depicts the interprocedural algorithm for computing the control flow graphs for the process families. The algorithm first replaces any strongly connected component of the call graph with a single placeholder node (lines 1–3), such that the resulting graph is acyclic. The algorithm then traverses the modified call graph in topological (calling) order. If a node represents a single function, the set of process families that may call the current function (`startset`) is accumulated (lines 7–9) and passed to the intraprocedural analysis for the current function (called at line 10). If a node represents a strongly connected component, the same steps are iteratively applied to the functions that are members of the strongly connected component, until the solution converges (lines 12–20).

This algorithm assumes that all processes in a process family that call a function return from that function, possibly as members of different process sub-families, derived from the (original) calling process family. This means that no function other than that where the processes were created may destroy a process.

3.2.4 Termination, Correctness and Complexity

Termination. Termination of the control flow analysis relies on bounding the number of times process families may be split. Since the number of PDPs in the program is bounded, and only process families represented by leaf nodes in the process family tree can be split, it suffices to show that if a process family is split at a control statement with a PDP, none of its leaf node sub-families can be split at that or any other control statement containing the same PDP. In other words, any process family represented by a leaf node can be the result of at most N splits, where N is the number of different PDPs in the program.

Recall that when a process family is split at a control statement with a PDP Π, the two new sub-families are constrained by Π and a truth value (one sub-family by $< \Pi, true >$, the other by $< \Pi, false >$) in addition to the constraints of the original process family (Figure 3.2, lines 19 and 20). If any of these new process families, or their leaf node sub-families, are propagated to a control statement with the same PDP, the previously added constraint will prevent them from being split. Instead, they will be propagated along the appropriate path as determined by the truth value associated with the constraint (Figure 3.2, lines 12–16). Thus, the number of process families is bounded, and the algorithm terminates.

Correctness. Given that PDPs are identified conservatively (the analysis may fail to identify a predicate as a PDP, but will not erroneously identify one that isn't), the correctness of the algorithm hinges on it not underestimating the possible control flow of any of the processes in a program. Overestimating the control flow, or creating ghost process families leads to a safe, conservative solution as described previously.

To underestimate the control flow of a given set of processes, they must be part of a process family P, which the algorithm propagates along the incorrect path from a control statement S that contains a PDP. Furthermore, there cannot be any other process family that the given set of processes can be part of instead.

The argument for correctness is done by contradiction. An underestimate of the control flow can only occur if any of the following three cases are true. The first is if the predicate of the control statement is not a PDP, but has been wrongly identified as such. That is, the processes in P may actually execute along both paths from S. However, this contradicts the given assumption that the identification of PDPs is conservative. The second case is if the predicate of S is a PDP and P is a sub-family produced by a split at S. But if there was a split at S, then there exists another process family (produced by the split at S), call it Q, that is identical to P up to S, but takes the correct path from S, i.e., it has the correct control flow. Thus, there is a process family that the given set of processes can be a part of, and therefore their control flow is not underestimated. The third case is if the predicate of S is a PDP and P is not a sub-family produced by a split at S. Then the path that P is propagated along from S is determined by the set of constraints associated with P. For this path to be incorrect, the set of constraints on P must be erroneous (too large); otherwise no process can satisfy these constraints and still take the other path. Since each constraint can only be added as the result of a split of a process family at a control statement with a PDP, it follows that P must either be a sub-family or descendant of a sub-family that was split at a control statement with a predicate that had been mis-identified as a PDP. However, this has already been covered in case one. Therefore, there cannot exist any processes for which the algorithm underestimates control flow, and the algorithm is correct.

Complexity. The intraprocedural algorithm (Figure 3.2) is implemented using a round-robin, iterative algorithm that has a worst case time complexity of $O(n^2)$ executions of the inner loop at line 9, where n is the number of basic blocks in the procedure [Hec77]. The set of constraints considered in line 10 is proportional to the number of distinct PDPs in the program. Since this number is smaller than the number of process families p, and the inner loop at line 9 iterates over the number of process families, the overall worst case time complexity of the algorithm is no greater than $O(n^2 \times p^2)$. However, most parallel programs exploit data parallelism by having different processes perform the same operations on different data. In fact, results show that no program in the workload had more than three process families. Therefore, in practice the number of process families can be bounded by a small constant, making the overall complexity no worse than $O(n^2)$.

For non-recursive programs, the interprocedural algorithm (Figure 3.6) invokes the intraprocedural algorithm once for each procedure in the program. So for q procedures, the overall worst case complexity is $O(q \times n^2)$, where n is the maximum number of basic blocks across the procedures. For recursive, programs the upper bound is $O(q^2 \times n^2)$ due to the iteration over the components of the strongly connected components.

3.2.5 Evaluation of the Process Family Control Flow Analysis

Table 3.1: Results of process family control flow analysis.

	Process families (single)		PDPs	
Program	Algorithm	Manual	Algorithm	Manual
Fmm	3 (2)	3 (2)	8	8
LocusRoute	2 (1)	2 (1)	1	1
Maxflow	2 (1)	2 (1)	3	3
Mp3d	3 (2)	3 (2)	2	2
Pthor	3 (2)	3 (2)	8	8
Pverify	2 (1)	2 (1)	1	1
Radiosity	2 (1)	2 (1)	1	1
Raytrace	2 (1)	2 (1)	2	2
Topopt	2 (1)	2 (1)	1	**2**
Water	2 (1)	2 (1)	3	3

To gauge the effectiveness of the process family control flow analysis algorithm, it was applied to each program in the workload. The ability of the algorithm to detect PDPs and divide the processes into process families according to their control flow was compared to that of manually inspecting the programs.

Table 3.1 shows the results of applying the per-process control flow analysis algorithm to the programs in the workload. For each program the number of occurrences of PDPs, process families, and the number of process families consisting of only a single process detected by the algorithm and by manual inspection are shown.

Interprocedural analysis was necessary to detect and separate the control flow of the process families in all programs with more than one PDP.

The experiments verify the per-process control flow analysis algorithm with respect to its accuracy in correctly identifying the process differentiating predicates and approximating the set of process families. For all the programs in the workload, the algorithm correctly computed the number of process families, and only for Topopt did it fail to recognize a PDP. This indicates that the algorithm can accurately separate processes correctly according to their control flow in the majority of cases. No ghost process families were created, which indicates that the limited test for equi-partitioning of PDPs is sufficient to resolve the equivalence of the predicates that appear in these programs. Similarly, the simple method by which PDPs are recognized is shown not to be a significant source of error. The only failure occurred because the PDP in question was located inside a procedure, one of whose formal parameters acted as a process differentiating variable. It was invoked from separate calls by two process families. At one call a process differentiating variable was passed as the parameter in question; at the other an integer constant was used. While the formal parameter did in actuality have different values for the different processes, the algorithm failed to recognize this because of the difference in its representation between the calls. This resulted in a more conservative estimate for the control flow graphs of the affected process families, i.e., larger control flow graphs, as illustrated in Figure 3.4.

Most coarse-grained parallel programs are to a large extent data parallel, in that different processes perform the same operations on different data. This is typically implemented either by partitioning arrays across the processes or dynamic scheduling with work queues. Except for any distinguished master or control process, one would therefore expect that the number of process families in these programs to be relatively small.

The results for the programs in the workload confirm this; they show that both the number of PDPs and the number of process families are small. Seven of the ten programs (LocusRoute, Maxflow, Pverify, Radiosity, Raytrace, Topopt and Water) have only two process families, the remaining three (Fmm, Mp3d and Pthor) have three. For each of the programs only one process family consists of multiple processes. This implies that most of the processes in the programs are similar in that they execute the same code but on different data. However, all the programs also have at least one single process process family that is distinguished by its control flow and therefore accesses shared data differently from the other processes.

3.3 Barrier Synchronization-based Non-Concurrency Analysis

Analyzing a program based on data summarized over the whole program often fails to recognize shifts in the reference pattern to shared data between different (non-concurrent) phases of the program. The barrier synchronization analysis algorithm addresses this problem by dividing the program into a set of sequentially executing phases, delineated by barriers, and computing the flow of control between them. This allows side effect analysis (Section 3.4) to be performed in each phase, thus providing a temporal component to the summary side-effect information.

This can be used in several ways when trying to minimize false sharing. If distinct access patterns are detected between barriers, shared data can be transformed dynamically (at runtime) or different coherency code that matches the access pattern of each phase can be generated. Or, as in the algorithms described in this dissertation, non-concurrency analysis can be used, along with static profiling, to detect a dominant sharing pattern in the program and restructure for that pattern. For example, in the application Topopt the barrier synchronization-based 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. The algorithm therefore decided to transform the data by process, according to its dominant usage. Summarizing over the entire program would have discouraged the transformation, because the summary information would not have reflected that the shared accesses to the data were localized to only a short phase of the program.

The barrier synchronization-based non-concurrency analysis algorithm can be used to obtain more detailed, per-phase summary information for shared data for applications other than eliminating false sharing. For example, in a parallel architecture in which processors have multiple hardware contexts [ALKK90], it can be used to schedule threads across processors to minimize interconnect operations caused by accesses to shared data. In a (single context) shared-memory machine, it can be used to enable simpler and more accurate data prefetchers [TE93].

The barrier synchronization analysis algorithm does not use the information computed in the control-flow analysis stage of the analysis; instead it conservatively assumes that each process may take any path through the program. This saves time and space, since the analysis does not have to be performed in the context of the control flow of each process.

The barrier synchronization algorithm has three major parts. First the program is divided into a set of process segments, each of which is delimited by barrier synchronization calls. These process segments are then selectively merged together into phases, based upon which segments may execute concurrently. Lastly, the statements of the program are marked with which phases each belongs to. The results of applying the barrier synchronization analysis algorithm to the workload concludes this section.

3.3.1 Barrier Synchronization Graph

In the barrier synchronization analysis algorithm a *process segment* is defined as the set of statement sequences along all barrier-free paths in the control flow graph that start at one barrier synchronization call i using barrier variable α, denoted $S_{i,\alpha}$, and end at another (possibly the same) barrier synchronization call $S_{j,\beta}$. A process segment is identified by the barrier synchronization calls that delimit it, i.e., the above process segment would be denoted $(S_{i,\alpha}, S_{j,\beta})$. As an example, consider the sample program shown in Figure 3.7. The program has eight barrier synchronization calls, numbered one through eight. In addition, a conceptual barrier synchronization call, denoted S, is assumed to exist at the beginning of the program. Using the definition described above, the program can be divided into the following process segments: $(S, S_{1,step}), (S, S_{5,step})$ $(S_{1,step}, S_{2,step})$, $(S_{2,step}, S_{3,odd})$, $(S_{2,step}, S_{4,step})$, $(S_{3,odd}, S_{4,step})$, $(S_{4,step}, S_{1,step})$, $(S_{5,step}, S_{6,step})$, $(S_{6,step}, S_{7,odd})$, $(S_{6,step}, S_{8,step})$, $(S_{7,odd}, S_{8,step})$, $(S_{8,step}, S_{5,step})$.

Process segments need not be disjoint. They may overlap, i.e., two process segments may start (end) at different barrier synchronization calls but end (start) at the same call, thus sharing the last (first) statements on the control flow paths to that end barrier synchronization call. An example is the process segments $(S_{2,step}, S_{4,step})$ and $(S_{3,odd}, S_{4,step})$ in Figure 3.7. However, one process segment cannot contain another, since this would violate the requirement for barrier-free paths.

A *phase* of a program is the set of process segments that may execute concurrently

```
..
for (pid = 1; pid < NumProc; pid++)
  if (fork() == 0) {
    SlaveWork(pid);
    exit(0);
  }
WorkStart(pid);
..
```

```
   WorkStart(pid)
   int pid;
   {
     ..
     while (not converged) do {
       comp_step_1(pid);
1:     wait_barrier(&step);
       comp_step_2(pid);
2:     wait_barrier(&step);
       if (step % 2) {
         do_odd(pid);
3:       wait_barrier(&odd);
       }
       iter++;
       converged = TestConvergence();
4:     wait_barrier(&step);
     }
   }
```

```
   SlaveWork(pid)
   int pid;
   {
     ..
     while(not converged) {
       comp_step1(pid);
5:     wait_barrier(&step);
       comp_step_2(pid);
6:     wait_barrier(&step);
       if (step % 2) {
         do_slave_odd(pid);
7:       wait_barrier(&odd);
       }
8:     wait_barrier(&step);
     }
   }
```

Figure 3.7: Sample program with barrier synchronization.

between two barrier synchronization events. The goal of the barrier analysis algorithm is to divide the program into process segments and then group process segments that may execute concurrently together in the same phase.

The algorithm relies on an important property of global barrier synchronization with respect to barrier variables. The same barrier variable must be passed by all processes for each global barrier synchronization event, otherwise a deadlock will occur (Chapter 2). Thus no two process segments that end at barrier synchronization call sites that pass different barrier variables can execute concurrently in a correct execution of a program without causing deadlock.

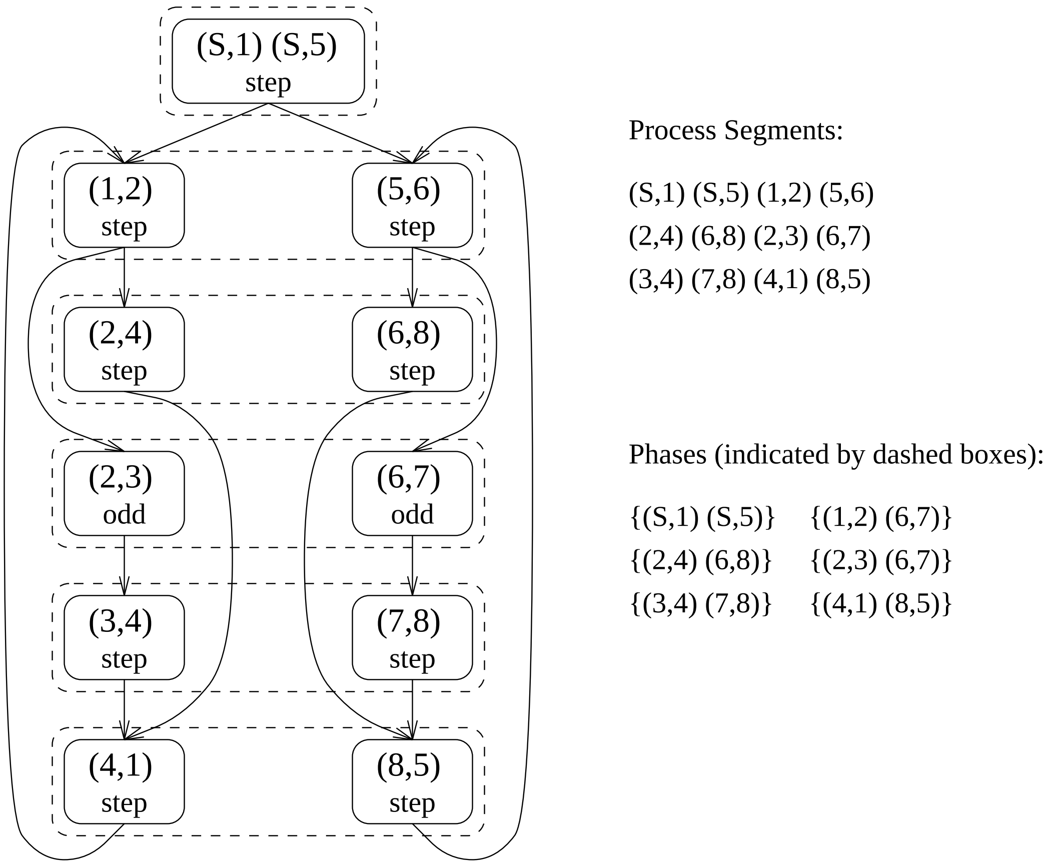


Figure 3.8: Barrier synchronization graph for program shown in Figure 3.7.

To help explain the algorithm, the notion of a *barrier synchronization graph* $G_B = (V_B, E_B)$ for a program is introduced. It is a rooted, directed graph, defined as follows:

1. Every vertex $v_{s,\alpha} \in V_B$ represents the set of process segments $T_{s,\alpha}$ that originate at the barrier synchronization call site s and terminate at barrier synchronization calls that pass the barrier variable α. A dummy start node[5] (root vertex) is inserted to point to the vertices that contain process segments originating from a conceptual barrier synchronization call at the beginning of the program. An *end barrier synchronization call* is not inserted, but instead observe that the sections of the program not covered by process segments make up a separate program phase that ends in program termination.

[5] For simplicity, this node is not shown in any illustrations of the barrier synchronization graph.

2. There is a directed edge $(u_{i,\beta}, v_{j,\gamma}) \in E_B$, if there are process segments $(S_{i,\alpha}, S_{j,\beta}) \in T_{i,\beta}$ and $(S_{j,\beta}, S_{k,\gamma}) \in T_{j,\gamma}$.

Intuitively, the nodes in the graph represent sets of statement sequences of the program that do not contain barrier synchronization calls (process segments), and the edges represent barrier synchronization events that transfer control flow from one segment to another. The phases of the program can be represented as a partition of the vertices, all of whose segments can execute in parallel. The flow of control between the phases follows easily.

The synchronization graph for the sample program in Figure 3.7 is shown in Figure 3.8. The vertices of the synchronization graph are shown in solid boxes, while dashed boxes enclose the process segments that belong to the same phase. Each vertex is annotated with the barrier variable that terminates the process segments it contains. Again, the dummy start node is not shown.

The results of the barrier synchronization analysis are conservative. If two process segments are in different phases, they cannot execute concurrently. However, the converse may not be true, i.e., if two process segments are in the same phase, they may, but need not, execute in parallel.

3.3.2 Computing the Process Segments

The first stage of the barrier analysis algorithm divides the program into a set of process segments. For each barrier synchronization call the other barrier synchronization calls that can be reached along barrier-free paths in the program are computed. Conceptually a variable $SynchVar_i$ is created for each barrier synchronization call S_i (as opposed to the barrier variable b that is passed as a parameter at S_i). Then each barrier synchronization call S_i is treated as the use of its variable $SynchVar_i$ followed by the definition of the variables $SynchVar_j$ for all j. The problem then reduces to that of computing which $SynchVar_i$ are live at the end of each barrier synchronization call. (Recall that a variable is live at a point in a program if its value can be used before it is redefined [ASU88].) Since these conceptual variables cannot be aliased, an interprocedural solution to this live *barrier* problem is possible and tractable [Mye81a]. Furthermore, live barriers can be formulated as a simple distributive data flow analysis problem [KU76], for which efficient solutions are known.

To speed up computation, procedures that do not contain any barrier synchronization calls and that do not directly or indirectly call any procedures containing barrier synchronization calls can be summarized into single nodes in the program control flow graph (thus avoiding propagating information through each statement of these procedures) without affecting the accuracy of the solution.

Following the solution to the interprocedural live barrier problem, the process segments are computed by considering each barrier synchronization call in turn. For each barrier $S_{j,\beta}$ that is live just after $S_{i,\alpha}$, the algorithm creates the process segment $(S_{i,\alpha}, S_{j,\beta})$.

Table 3.2 summarizes the results of solving the interprocedural live barrier problem for the program shown in Figure 3.7. A check mark in row i, column j indicates that barrier synchronization call j is live at barrier synchronization call i. For instance, the table shows that barrier synchronization calls 1_{step} and 5_{step} are both live at S. Thus, the algorithm creates process segments $(S, 1_{step})$ and $(S, 5_{step})$. The full set of process segments (again, without barrier variable annotations) for this program is listed in Figure 3.8.

Table 3.2: Set of barrier synchronization calls live at each barrier synchronization call.

Barrier	Live Barrier Synchronization Calls								
Synch. Call	S	1_{step}	2_{step}	3_{odd}	4_{step}	5_{step}	6_{step}	7_{odd}	8_{step}
S		$\surd$				$\surd$			
1_{step}			$\surd$						
2_{step}				$\surd$	$\surd$				
3_{odd}					$\surd$				
4_{step}		$\surd$							
5_{step}							$\surd$		
6_{step}								$\surd$	$\surd$
7_{odd}									$\surd$
8_{step}						$\surd$			

3.3.3 Partitioning Process Segments into Phases

Partitioning the process segments into sequentially executing phases is done in the second stage of the barrier analysis algorithm. It begins with the optimistic assumption that only process segments that start at the same barrier synchronization call and end at barrier synchronization calls that all pass the same barrier variable can execute concurrently. In other words, each barrier synchronization call $S_{i,\alpha}$ gives rise to phases:

$\{(S_{i,\alpha}, S_{j_1,\beta}), (S_{i,\alpha}, S_{j_2,\beta}), .., (S_{i,\alpha}, S_{jn,\beta})\}$,
$\{(S_{i,\alpha}, S_{k_1,\gamma}), (S_{i,\alpha}, S_{k_2,\gamma}), .., (S_{i,\alpha}, S_{kn,\gamma})\}$
through $\{(S_{i,\alpha}, S_{l_1,\omega}), (S_{i,\alpha}, S_{l_2,\omega}), .., (S_{i,\alpha}, S_{ln,\omega})\}$,

where $S_{j_1,\beta}, S_{j_2,\beta}, .., S_{jn,\beta}, S_{k_1,\gamma}, S_{k_2,\gamma}, .., S_{kn,\gamma}$, and $S_{l_1,\omega}, S_{l_2,\omega}, .., S_{ln,\omega}$ are barrier synchronization calls live at $S_{i,\alpha}$, and $\alpha, \beta, ..., \omega$ are barrier variables. These initial phases correspond to the vertices of the barrier synchronization graph. The initial phases for the sample program in Figure 3.7 are illustrated by the solid boxes in Figure 3.8.

	Initially	Step 1	Step 2	Step 3	Step 4	Final
Phase A:	{(S,1),(S,5)}	{(S,1),(S,5)}	{(S,1),(S,5)}	{(S,1),(S,5)}	{(S,1),(S,5)}	{(S,1),(S,5)}
Phase B:	{(1,2)}	{(1,2),(5,6)}	{(1,2),(5,6)}	{(1,2),(5,6)}	{(1,2),(5,6)}	{(1,2),(5,6)}
Phase C:	{(2,4)}	{(2,4)}	{(2,4),(6,8)}	{(2,4),(6,8)}	{(2,4),(6,8)}	{(2,4),(6,8)}
Phase D:	{(2,3)}	{(2,3)}	{(2,3)}	{(2,3),(6,7)}	{(2,3),(6,7)}	{(2,3),(6,7)}
Phase E:	{(3,4)}	{(3,4)}	{(3,4)}	{(3,4)}	{(3,4),(7,8)}	{(3,4),(7,8)}
Phase F:	{(4,1)}	{(4,1)}	{(4,1)}	{(4,1)}	{(4,1)}	{(4,1),(8,5)}
Phase G:	{(5,6)}					
Phase H:	{(6,8)}	{(6,8)}				
Phase I:	{(6,7)}	{(6,7)}	{(6,7)}			
Phase J:	{(7,8)}	{(7,8)}	{(7,8)}	{(7,8)}		
Phase K:	{(8,5)}	{(8,5)}	{(8,5)}	{(8,5)}	{(8,5)}	

Figure 3.9: Illustration of how the algorithm partitions the set of process segments into phases.

From this starting point phases that may execute concurrently are merged until no two process segments that may execute in parallel are contained in different phases. The decision to merge two phases is based on the fact that, if two process segments belong to the same phase and therefore may execute concurrently, such as $(S, S_{1,step})$

and $(S, S_{5,step})$ in Figure 3.8, then all their immediate successor process segments that terminate at barrier synchronization calls that pass the same barrier variable may also execute in parallel. Therefore the phases that contain these successor process segments are merged into one phase. Phases that contain process segments that terminate at barrier synchronization calls that pass different barrier variables are never merged, since deadlock would occur if these were executed concurrently.

Figure 3.9 illustrates how the algorithm works for the sample program in Figure 3.7. Again, barrier variables have been omitted for reasons of readability. The program has nine barrier synchronization calls (eight plus the start barrier). However, since not all the barrier variables are identical, the process segments are initially partitioned into 11 phases, A through K. Initially, only phase A is put on the work queue, since it is the only phase that contains more than one process segment, specifically the process segments $(S, 1)$ and $(S, 5)$. As can be seen from the figure, barrier synchronization calls 1 and 5 are first components of process segments in phases B and G, respectively. Since they both end at barrier synchronization calls that pass the same barrier variable, they are merged together in B, which is then added to the work queue. The presence of process segments $(1, 2)$ and $(5, 6)$ in B causes first the phase C and phase H to be merged, and then phase D and I to be merged. This process follows until phase F and K are merged to produce the final set of phases.

This stage of the algorithm, shown in Figure 3.10, is implemented using a work queue approach. First, using the live barrier information, the process segments are computed and inserted into the initial set of phases (lines 1–8). All phases P_i that contain multiple process segments are put on the work queue (lines 9 through 11). Each phase P_i on the queue is then examined in turn (line 15). For each pair of process segments $(S_{j,\alpha}, S_{k,\beta}), (S_{r,\gamma}, S_{s,\beta}) \in P_i$, the algorithm merges all phases that contain either $S_{k,\beta}$ or $S_{s,\beta}$ as the first barrier synchronization call in any of their process segments and whose process segments all end in barrier synchronization calls that pass the same barrier variable (β) (lines 17 through 29). The resulting program phase is added to the work queue (lines 28). This continues until the work queue is empty, after which the final set of phases remain.

```
 1. for each barrier call B1 in the program
 2.   if any barrier calls  are live at B1 then
 3.     create a phase Pi for each different barrier variable
 4.          appearing in barrier calls live at B1;
 5.     for each barrier call B2 live at B1 do
 6.       insert (B1,B2) into Pi.Segments and B1 into Pi.StartBarrier,
 7.         where B2 passes same barrier variable as associated with Pi;
 8.     end for;
 9.     for each Pi that contains more than one process segment
10.       Enqueue Pi on Workqueue
11.     end for;
12.   end if;
13. end for;
14. while not Workqueue empty do
15.   Dequeue Pi;
16.   MergeSet = empty set;
17.   for each barrier call Bj such that (Bi,Bj) is in Pi.segments do
18.     PSet = set of phases Pk with Bj in Pk.StartBarrier;
19.     for each distinct barrier variable BV associated with
20.         a phase in Pset do
21.       MergeSet = phases in Pset with barrier variable BV;
22.       if |MergeSet| > 1 then
23.         create a new phase Pn with barrier variable BV;
24.         for each phase Pm in MergeSet do
25.           merge Pm with Pn;
26.           destroy Pm; /* remove from Workqueue if necessary */
27.         end for;
28.         Enqueue Pn on Workqueue;
29.       end if;
30.     end for;
31.   end for;
32. end while;
```

Figure 3.10: Algorithm to compute the set of phases from a set of process segments.

Termination, Correctness and Complexity

Termination. Termination of the barrier analysis algorithm (Figure 3.10) follows from three facts. First, no more merges can be performed than there are initial phases (line 3). Second, a phase is removed from the work queue (line 15) in every iteration of the main loop. Third, only a phase that is the result of merging two or more phases (lines 23–27) can be added to the work queue (line 28). Thus, the sum of the number of phases in the program and the number of phases in the work queue decreases for every iteration of the main loop, and the algorithm must terminate.

Correctness. The result of the algorithm is a set of phases such that a process segment in one phase cannot execute concurrently with any process segment in another phase. Therefore, the algorithm is correct if it always merges phases that contain process segments that may execute concurrently. Merging phases that contain process segments that cannot execute concurrently produces a conservative, but safe approximation.

Recall that for two process segments to execute concurrently, it is necessary, but not sufficient, that they both start at barrier synchronization calls that pass the same barrier variable, and end at barrier synchronization calls that pass the same barrier variable. If not, they could not execute in parallel without causing a deadlock.

The argument for correctness is done by contradiction. Assume that after running the algorithm two process segments $x = (S_{i,\alpha}, S_{j,\beta})$ and $y = (S_{m,\alpha}, S_{n,\beta})$ are in phases P_1 and P_2 respectively, but may execute concurrently in a correct execution of the program. There are two cases, depending on whether the two process segments start at the same (i.e., $i = j$) or different (i.e., $i \neq j$) barrier synchronization calls. In the first case, they cannot end at barrier synchronization calls that pass the same barrier variable, otherwise the algorithm would have put them into the same phase during initialization (lines 5–8). However, if they end at barrier synchronization calls that pass different barrier variables, they cannot execute concurrently without causing deadlock, thus contradicting the initial assumption. In the second case it can be assumed, without loss of generality, that since x and y can execute concurrently there is a phase P_0 that directly precedes both P_1 and P_2, i.e., flow of control may flow directly from P_0 to P_1 and P_2. Thus, for P_1 and P_2 not to be merged at line 25, either their process segments end at barrier synchronization calls that pass different barrier variables, or P_0 is never put on the work queue. The former violates the initial assumption that the two process segments may execute concurrently. The latter can only occur if P_0 contains only one process segment. Otherwise it would be enqueued at line 10 (if it initially contained more than one process segment) or line 28 (if it was the result of a merge). However, since P_0 directly precedes P_1 and P_2, if P_0 contains only one process segment, call it $z = (S_{r,\gamma}, S_{s,\alpha})$, P_1 and P_2 must contain at least one process segment each that starts at barrier synchronization call $S_{s,\alpha}$, and ends at a barrier synchronization call that passes the barrier variable β. But this cannot happen, since the algorithm would initially have put these process segments in the same phase (lines 5–8), and thus $P_1 = P2$. Therefore, x and y cannot execute concurrently, and the algorithm is correct.

Complexity. The running time of the initialization phase is determined by the number of process segments in a program. The maximum number of process segments exists when every barrier is live at every other barrier. In this case the time complexity is $O(b^2)$, where b is the number of barriers in the program.

The maximum number of initial phases that can be created is $b \times v$, where v is the number of different barrier variables passed at barrier synchronization calls in the program. Since the minimum number of final phases is v (one for each barrier variable), the maximum number of merges that can be performed in the main loop (line 25) is $b \times v - v$. Using fast union-find for set merges and lookups, the overall worst case time complexity of the main loop is $O(b \times v)$. Since $v \leq b$, the overall worst case running time of the algorithm is $O(b^2)$.

3.3.4 Marking the Program

In order to accumulate side-effect information by each of the phases of the program, it is necessary first to mark each statement with the phases to which it belongs. This is done by computing which statements belong to each process segment and then which phase contains each process segment.

Recall that a process segment is the set of statements that may be executed along all barrier-free paths between two given barrier synchronization calls, and that the process segment is identified by the two barrier synchronization calls that delimit it. Thus, in order to determine which process segments a statement belongs to, it is necessary to know the set of barrier synchronization calls that are reachable along barrier-free paths from the statement and the set of barrier synchronization calls from which the statement can be reached, also along barrier-free paths. The first set is computed when process segments are identified, namely when solving the live barrier problem. The second set can be computed by solving a problem similar to reaching definitions for barriers.

The *reaching barriers* problem is solved in a similar way to that of the live barrier problem. Each barrier synchronization call i is treated as an assignment to a non-aliased, conceptual variable $SynchVar_i$. Departing slightly from the reaching definition problem for variables, the assignment to $SynchVar_i$ is treated as if it kills off all definitions of conceptual variables that reach that assignment. The problem can now be solved using known techniques for distributive data flow problems [ASU88, KU76].

Given the sets of barriers that reach and are live at a statement S in the program, the sets of phases that include this statement can be easily computed. For each barrier B_m that reaches S, compute the set of phases P_r that includes process segments (B_m, B_i) for any i. For each barrier B_n that is live at S, compute the set of phases P_l that includes process segments (B_j, B_n) for any j. The set of phases that include S is simply the intersection of P_r and P_l.

3.3.5 Experimental Results

Table 3.3: Applying the barrier analysis algorithm to the workload.

Program	Barriers	Process Segments	Phases found by algorithm	Phases found manually
Fmm	10	13	5	**10**
LocusRoute	2	2	1	1
Maxflow	6	6	6	6
Mp3d	6	7	7	7
Pthor	9	22	13	**20**
Radiosity	3	6	2	**6**
Raytrace	1	1	1	1
Topopt	10	15	5	5
Water	6	8	8	8

To evaluate the effectiveness of the barrier synchronization analysis algorithm in defining phases and process segments, it was applied to each benchmark, except Pverify which did not contain any barriers. Table 3.3 summarizes these results. The table

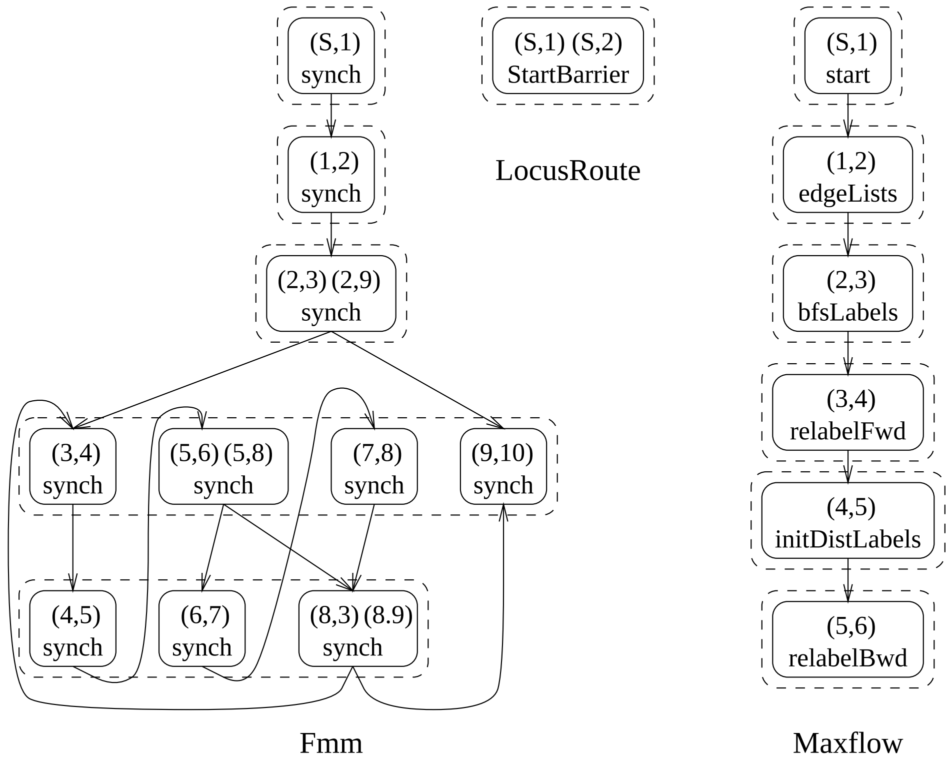


Figure 3.11: Barrier synchronization graphs for Fmm, LocusRoute and Maxflow.

contains the number of static occurrences of barriers in each program, the number of process segments in the program, phases computed by the algorithm (excluding the termination phase), and the number of phases that can be determined by manually inspecting the program (again, excluding the termination phase). The barrier synchronization graphs for the benchmarks, annotated with phases as computed by the algorithm, are shown in Figures 3.11, 3.12, 3.13 and 3.14.

The barrier analysis algorithm computes the correct number of process segments for all benchmark programs. In addition, it partitions these segments perfectly for all but three of the benchmarks. Comparing the barrier synchronization graphs with Table 3.3 shows that the algorithm divides LocusRoute (trivially), Maxflow, Mp3d, Raytrace (trivially), Topopt and Water into a maximum number of phases. That is, for these programs the results are exact, not conservative.

The results for Fmm, Pthor and Radiosity however, are conservative. Instead of finding 10, 20 and 6 phases, respectively, the algorithm only partitions the process segments into 5, 13 and 2 phases. Two characteristics shared by these three programs

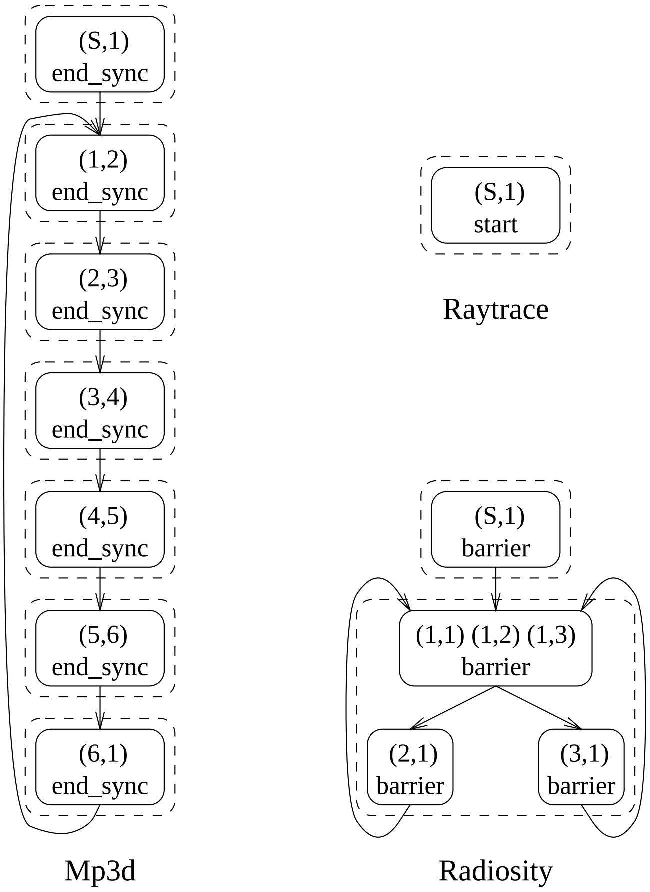
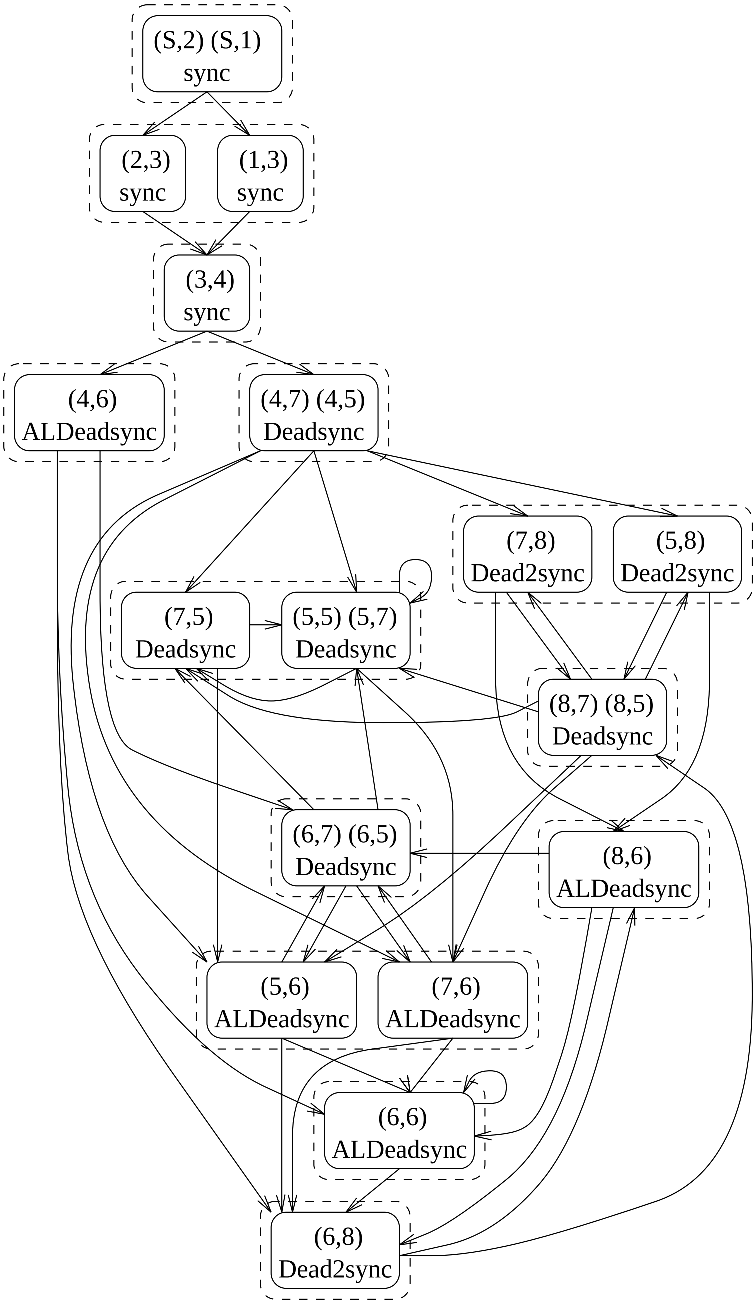


Figure 3.12: Barrier synchronization graphs for Mp3d, Radiosity and Raytrace.

combine to cause this inaccuracy. First the same barrier variable is used at several barrier synchronization call sites that in reality can never execute concurrently, e.g., two barrier synchronization that are on different paths (*true*/*false*) from an if statement, when the predicate of that if statement is a run-time constant. Second, there are cycles of different lengths in the barrier synchronization graph of each program. The former links barriers by the storage used for the synchronization variables, and not the actual synchronization operation. The latter forces the algorithm to merge phases containing process segments that end at these barriers. For instance, in Fmm the barrier variable `synch` is used at each of the ten barrier synchronization call sites in the program, even though no two of them are ever executed in parallel. The presence of two different length cycles (these are {(3,4), (4,5), (5,6), (6,7), (7,8), (8,3)} and {(3,4), (4,5), (5,8), (8,3)}) means that the algorithm must conservatively assume



Pthor

Figure 3.13: Barrier synchronization graph for Pthor.

that process segments (3,4), (5,6), (5,8), and (9,10) may execute in parallel, as may (4,5), (6,7), (8,3) and (8,9).

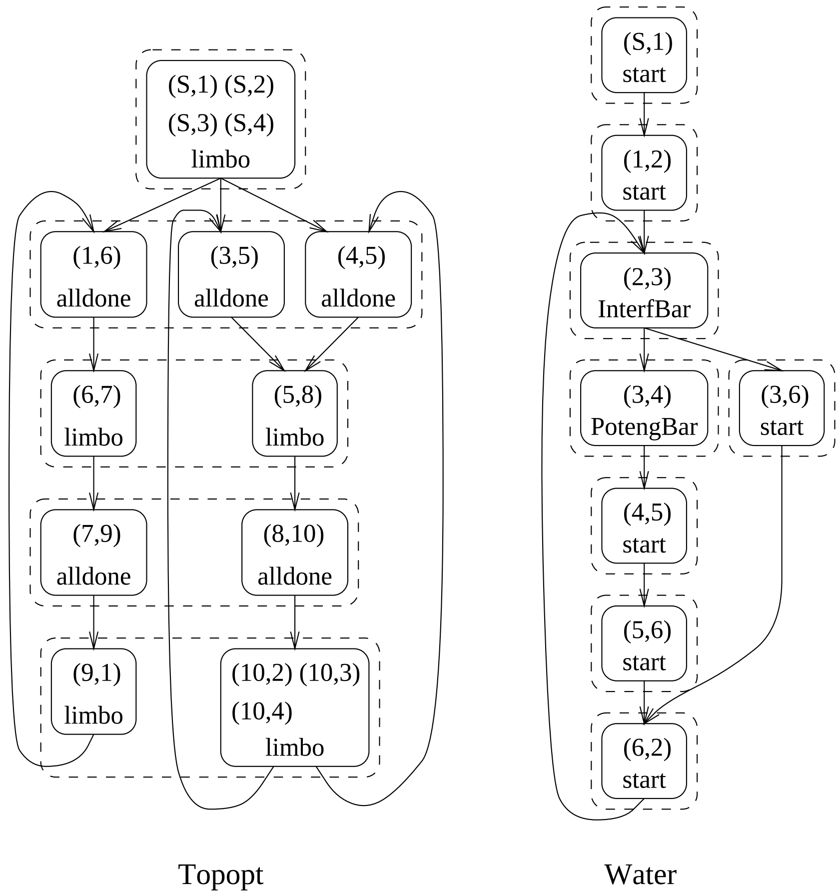


Figure 3.14: Barrier synchronization graphs for Topopt and Water.

The use of different barrier variables and the ability of the algorithm to differentiate between them were significant in obtaining the good results for both Water and Pthor. In Water, since the barrier variables at barrier synchronization calls 4 and 6 are different, the phases containing (3,4) and (3,6) were not merged. Had they been identical or aliased, the algorithm would have had to act conservatively and merge all but the two first phases into a single phase with a self loop. Similarly for Pthor, where all process segments that succeed (4,5), (4,7) and (4,6) in the barrier synchronization graph would have been lumped into a single phase with a self loop, if barrier variables were not taken into account during the analysis.

3.4 Summary Side-Effect Analysis

Following the first two stages of analysis, the control flow of different process families has been computed (stage 1) and the program has been divided into phases (stage 2). Stage 3 uses this information to compute any differences in the memory reference pattern between process families in and across the phases of the program. This is done by performing an interprocedural, summary side-effect analysis over the parallel portion of the program. Since each statement is marked both with which process families may execute it as well as which phase(s) contain it, the side-effect information is easily accumulated by phase for each process family using only a single pass over the program.

The side-effect analysis computes the set of data object each process family reads and writes in each phase of the program. A data object is either a global variable or meta-object that represents heap allocated data objects of a given type. Elements of structures are treated as variables in their own right. For arrays, additional information about the index expressions are maintained in the form of *regular sections* to specify the set of array elements accessed.

Since the techniques for side-effect analysis have been discussed extensively [Bar78, Ban79, Mye81b, CK88], this section will only dwell on those areas of the analysis that represent new techniques or novel applications of well known techniques.

3.4.1 Analyzing Array Accesses

The side-effect analysis serves two purposes. First, it computes the differences in the memory reference pattern of the processes that result from differences in their control flow. Second, it provides the basis for discovering differences in the cross-processor memory reference pattern to arrays. In the target model of parallel programming the partitioning of shared arrays is done implicitly, by different processes using different values in array index expressions for the same array. This is achieved by using process differentiating variables (see Chapter 2) as part of loop bounds or array index expressions. Thus, in order to determine if and how arrays are partitioned, it is necessary to compute the set of array elements that each process accesses.

The side-effect analysis accomplishes this by using bounded regular section descriptors [HK91] to describe and summarize which sections of an array may be read

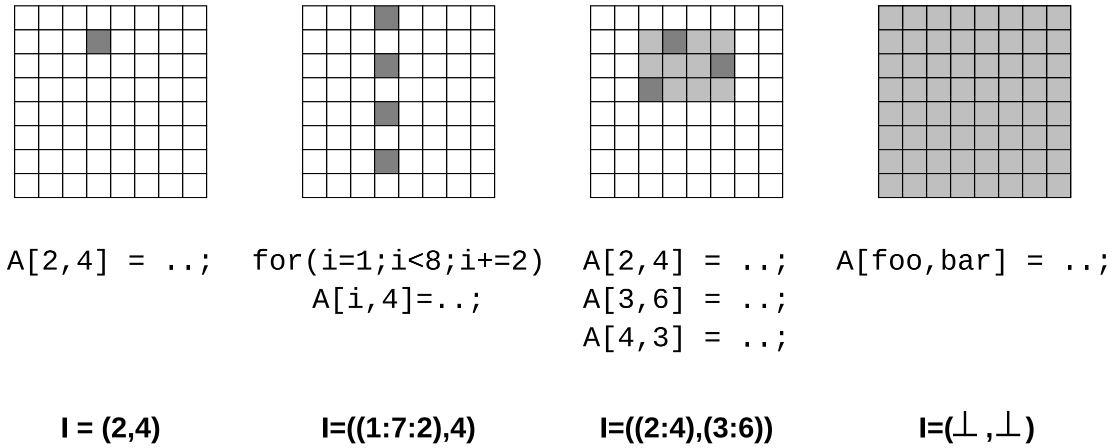


Figure 3.15: Summarizing array references.

or written at the points in the program where it is accessed. A bounded regular section descriptor is a vector of subscript positions $I = (i_1, i_2, .., i_n)$, in which each element i_j describes the portion of the array that is accessed in the jth dimension. Each vector element i_j can be in one of four forms. Initially, each element is set to $\top$, which represents the empty index set, i.e., the array has not been accessed. Second, i_j can be a single constant or a simple expression of constants, invariant variables, and process differentiating variables. In this case it specifies only one element in that dimension. This is illustrated by the first example in Figure 3.15. Third, i_j can be in the form of a triple (lb, up, st) that specifies a range of elements in the jth dimension using the same type of invariant expressions described above for their lower bound, upper bound and stride. This is illustrated by the second and third examples in Figure 3.15. If any of these expressions are unknown or invariant, they are denoted $\perp$, for unknown. The worst case for that expression must then be assumed, e.g., if the upper bound expression was unknown, the maximum upper bound would have to be used instead. Fourth, the i_j itself can be unknown ($\perp$), which represents the fact that either the index expression is too complex to represent in the framework, or that all the subexpressions (lb, up, st) are unknown. In this case all the elements of the jth dimension are implicitly specified. The fourth example in Figure 3.15 illustrates this.

In order to summarize the accesses to an individual array, it is necessary to merge the regular section descriptors, generated at each point in the program where that array is accessed, into a single descriptor. Two bounded regular section descriptors

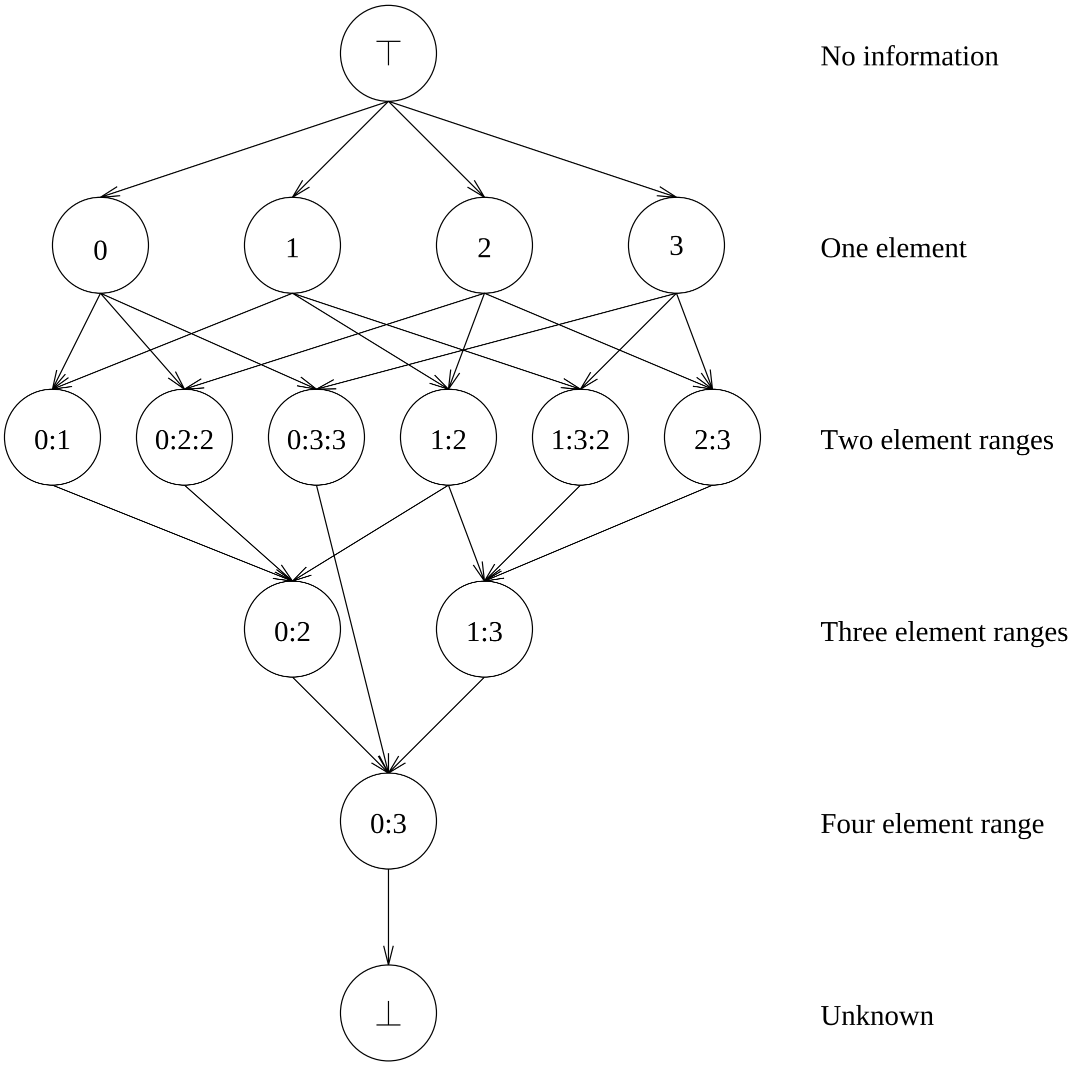


Figure 3.16: Lattice of subscript vector elements for a four element vector

are merged by pairwise merging the elements of the vector of subscript positions, i.e., merging the descriptors for each dimension. The result of merging two subscript vector elements is an element that describes the smallest section of the array in that dimension that encompasses both sections described by the two original elements. The subscript vector elements are partially ordered within a semi-lattice according to the size of the array section that they specify (as an example, Figure 3.16 shows the lattice of all possible constant entry subscript vector elements for a four element vector). Given this lattice, the merge of two subscript vector elements is defined as the meet, or greatest lower bound, of those elements in the lattice.

Summarizing the accesses to an array in a single bounded regular section descriptor serves the purpose of determining side-effects of procedures for dependence analysis well. However, the purpose of the analysis presented in this chapter is to detect per process access to shared data structures. Merging regular section descriptors as described above may interfere with this because of the loss of accuracy that results from the meet operations. For instance, consider an array in which the elements are partitioned statically across the processes using process differentiating variables for all array accesses except during a convergence testing phase, when all the array elements are shared. When the regular section descriptors for the accesses in the convergence testing part of the program are merged with those for the accesses in the remainder of the program, the meet (greatest lower bound) of the elements of these descriptors must be taken. Since not all of the regular section descriptors contain process differentiating variables as part of the elements in the subscript vectors, the merged regular section descriptor will not either. This eliminates information about the partitioning of the array and makes the array appear shared; consequently any transformation of the array to reduce false sharing is discouraged.

The solution to this problem is to use multiple bounded regular section descriptors for each array together with lazy and selective merging. Multiple bounded regular section descriptors give a more precise approximation of per-process array accesses in different parts of the program, since different access patterns need not be described by the same descriptor. Together with static estimation of execution frequencies (Section 3.4.3) it allows the system to disregard those descriptors that represent array access patterns that occur seldom and do not substantially contribute to false sharing.

There are, however, problems with allowing multiple regular section descriptors for each array to exist in the summary side-effect information. First of all, array accesses are no longer summarized, but enumerated. Second, the space requirement is significantly increased. Lastly, the more regular section descriptors there are, the more expensive any analysis that uses them becomes. Therefore it is necessary to make a tradeoff between the number of descriptors that is allowed and the accuracy of the resulting information. This is achieved by setting a small limit on the number of descriptors for each array (the results presented in this dissertation are based on using a limit of ten regular section descriptors for each array) and using a policy of lazy and selective merging.

The policy of lazy and selective merging seeks to minimize the loss of accuracy caused by merging two regular section descriptors. Unless two descriptors for the same array are identical, they are not merged until the limit on regular section descriptors for that array is reached. When the limit is reached, a descriptor is selected so that when it is merged with the new descriptor the least amount of information is lost. For instance, if there already is a descriptor for an array that contains bottom ($\perp$), and the result of merging the new descriptor with any other descriptor for that array would result in a descriptor that contains bottom, the descriptor that already contains bottom is chosen to be merged with the new one.

Once the side-effect analysis has been performed, the partitioning of arrays across processes can be detected by examining the regular section descriptors of each array. The algorithm can conclude that different elements of an array are accessed by different processes, even within a process family, in two cases. First, if the subscript vector element is not a range, but a process differentiating variable, each process accesses a different array element in that dimension. Second, if the subscript vector element is a range and the first element of the range is a process differentiating variable, and the stride is equal to the number of processes, the processes access different elements of the array in that dimension in an interleaved fashion. While both access patterns were detected in the workload, the first case alone was sufficient to detect all significant cases of false sharing in the workload caused by the static partitioning of arrays across the processes.

3.4.2 Heap-Allocated Objects and Pointers

In order to perform static analysis on programs written in a language with pointers and recursive data structures made up of heap-allocated objects, such as C, there are two separate, but related, issues that need to be resolved: how to deal with aliasing induced by pointers and how to treat (and summarize) accesses to heap-allocated data objects.

Pointer-induced aliasing of named data objects, except for pointers used to implement reference parameters, is not handled in the side-effect analysis described in this chapter. While the current side-effect analysis lacks pointer alias analysis, it has been shown that such analysis can be included to provide very accurate results at a relative low cost [LRZ93]. However, the effect of including such analysis on the final results would be negligible for the programs in the workload in this study. None of the programs use the full generality of pointers in C. Most of them conform to the constraints listed below, the remainder have been modified to do so[6]. The following three constraints on pointer usage hold across the workload. First, except for the reference parameter pointers, no pointer points to a named object, but points to heap-allocated data structures instead. Second, a pointer only points to data objects of the same data type as that associated with it in its declaration. Third, there is no pointer arithmetic[7].

Heap-allocated data objects present a problem in the side-effect analysis in that they are nameless, their lifespans are not very well defined, and their number and inter-connectivity is unknown at compile-time. They can make up arbitrarily large and complex data structures. However, since many parallel programs make extensive use of heap-allocated data structures, it is necessary to integrate them into the side-effect analysis to maximize the amount of false sharing that can be detected.

Subject to the constraints on pointer usage listed above, several techniques aim at discovering the structure and connectivity of heap-allocated data structures statically and can be used to analyze the accesses to them [Wei80, LH88, RM88, HPR89, HN90,

---

[6] This was primarily done by syntactically changing pointer accesses to array elements into array accesses with subscript expressions.

[7] Several programs, such as Cholesky and Barnes-Hut from the SPLASH benchmarks, were considered for inclusion in the workload, but were ultimately rejected, as they failed to meet these constraints without significant program modifications.

CWZ90]. However, due to their time and space cost, a simpler approach is used here. Instead of attempting to differentiate between individual heap-allocated objects of the same data type, we create a meta-object for the data type and summarize all accesses to heap-allocated objects of that data type as if they were accesses to the meta-object. In other words, during side effect analysis we treat all accesses to heap-allocated objects of any given data type as if they had occurred to a single static object. Based on this analysis, the algorithm can detect per-process family accesses to different data types, and to components of objects of the same (structure) data type. The algorithm cannot determine whether different process families access different objects of the same type. Therefore, it must conservatively conclude that all objects of a given data type are shared across all the processes.

3.4.3 Static Execution Frequency Estimates

The performance impact of false sharing that occurs in a particular data structure is directly related to how frequently that data structure is accessed. The more often it is accessed in a pattern that causes false sharing, the more false sharing cache misses occur. However, the results of standard summary side-effect analysis are only the sets of data objects that may be read or written when a procedure is executed. No information about how frequently that procedure may access the data objects is produced. Thus such an analysis would fail to discern the dominant (most frequently executed) memory reference pattern. This means that any decision to transform a data structure based on summary side-effect information alone would not necessarily result in significant performance improvements. Moreover, transformations would be precluded in the cases where data structures are accessed in memory patterns that result in true sharing as well as false sharing, even though the latter may dominate. For this reason static estimation of execution frequencies has been included in the side-effect analysis.

The static estimation weighs statements inside a loop more heavily than those outside of the loop. A factor of ten is common and is also used in the implementation for this research. Statements inside conditionals may be weighted less than those outside conditionals either by a constant factor or a probability assigned by heuristics. However, this has not been implemented in the current version of the side-effect analysis.

With the estimated execution frequencies in place, the side-effect analysis, instead of just computing the sets of elements accessed, computes sets of tuples that contain the data object and access frequency. When a data object is accessed in two different statements, the execution frequency of each statement is added to the access frequency of the tuple containing the data object. When side-effect information is propagated across procedure calls, the access frequencies of the side-effects of the called procedure are multiplied by the execution frequency of the call site.

While currently not implemented, an alternate approach would be to compile, execute and profile programs, and then feed the profiling information back into the compiler and use it instead of the static profiling information during a second compilation. This approach would provide more accurate information than possible at compile-time, provided the profiling runs of the program are on representative program inputs [Wal91].

Table 3.4: The time (in seconds) to perform different analyses in Parafrase-2 (1).

Pass	Fmm	LocusRoute	Maxflow	Mp3d	Pthor
total	14.91	67.00	3.83	7.62	139.65
callgraph construction	0.07	0.07	0.01	0.03	0.12
ref. parameter analysis	0.13	0.15	0.03	0.06	0.20
flowgraph construction	0.73	44.71	0.39	0.24	31.51
loop analysis	2.59	6.20	0.49	1.67	8.43
dependence analysis	2.06	5.59	0.91	3.23	35.60
constant folding/prop.	2.33	5.13	0.24	0.64	12.88
other (parsing etc.)	3.85	4.17	1.34	1.41	8.52
fraction of total	78.0%	98.3%	88.3%	94.8%	69.5%
process family	0.11	0.08	0.03	0.02	0.22
barrier analysis	0.24	0.33	0.04	0.05	0.79
side-effect	2.93	0.72	0.38	0.33	41.58
fraction of total	22.0%	1.7%	11.7%	5.2%	30.5%

3.5 Compilation Times

Table 3.5: The time (in seconds) to perform different analyses in Parafrase-2 (2).

Pass	Pverify	Radiosity	Raytrace	Topopt	Water
total	18.17	38.03	62.83	17.72	13.54
callgraph construction	0.05	0.09	0.13	0.03	0.03
ref. parameter analysis	0.10	1.46	0.18	0.06	0.06
flowgraph construction	2.92	0.75	5.23	4.27	1.61
loop analysis	5.15	2.29	4.48	3.47	2.63
dependence analysis	5.07	3.46	43.67	5.51	5.63
constant folding/prop.	1.48	11.18	2.41	1.41	0.95
other (parsing etc.)	2.73	8.57	5.49	2.04	2.04
fraction of total	95.8%	69.1%	97.7%	94.4%	95.2%
process family	0.05	0.09	0.08	0.04	0.03
barrier analysis	0.00	0.30	0.22	0.18	0.05
side-effect	0.72	11.37	1.12	0.77	0.57
fraction of total	4.2%	30.9%	2.3%	5.6%	4.8%

Tables 3.4 and 3.5 show the total time for Parafrase-2 to compile each program, as well as the time spent on the most important individual passes. The experiments were performed on a DEC 3000 workstation with 128 MBytes of physical memory. The order of the individual passes indicate the order in which they are performed. The last three rows show the time to perform the per-process control flow analysis, the barrier synchronization-based non-concurrency analysis, and the summary side-effect analysis, respectively.

Overall, the results indicate that the passes of analysis described in this chapter have very little compile-time cost. For LocusRoute, Maxflow, Mp3d, Pverify, Raytrace, Topopt and Water, the three passes account for an average of 5% of total compile-time. For Fmm, Pthor, and Radiosity the per-process control flow and non-concurrency analysis take less than 2% of the time of the overall compile-time. However, the summary side-effect analysis takes up an inordinate amount of time.

The reason for this is that the memory requirement for the compiler for these programs increases to cause paging activity which inflates the running time of the last stages of analysis. The excessive memory use is in part due to a memory leak in the constant folding and propagation pass. With more physical memory, or a more space economical compiler, the running time of the side-effect analysis should be significantly improved.

Chapter 4

DATA TRANSFORMATIONS

Once the compile-time analysis detailed in Chapter 3 has been performed, heuristics are used to determine which data structures are vulnerable to false sharing and which transformations should be applied to each of these data structures to eliminate or significantly reduce the number of false sharing misses. The transformations eliminate false sharing by restructuring the data so that (1) data objects that are only, or overwhelmingly, accessed by one processor are grouped together, and (2) write shared data objects with little or no processor locality do not share cache lines. Two transformations, *group and transpose* and *indirection* [EJ91], address item (1); the third, *pad and align*, is aimed at item (2).

To simplify development and exploration of different heuristics, the heuristics and transformations were applied by hand.

4.1 Group & Transpose

Group & transpose (as illustrated in Figure 4.1) physically regroups data together by changing their layout in memory. It groups together vectors that are allocated statically in which adjacent elements are always or predominantly accessed by different processes and then transposes the group. In effect, it changes a group of vectors into a vector of records, in which each field corresponds to a vector element. This way, all the elements that are accessed by the same process are allocated contiguously. If each process's data is less than the cache block size, it may be padded, so that no two processes' data share a cache block.

Group & transpose eliminates false sharing, since data that are accessed by different processes no longer reside in the same cache block. Furthermore, it improves spatial locality, despite the additional space created by padding, since the data accessed by each process is no longer spread over many cache blocks, but concentrated in a few (or only one as in Figure 4.1).

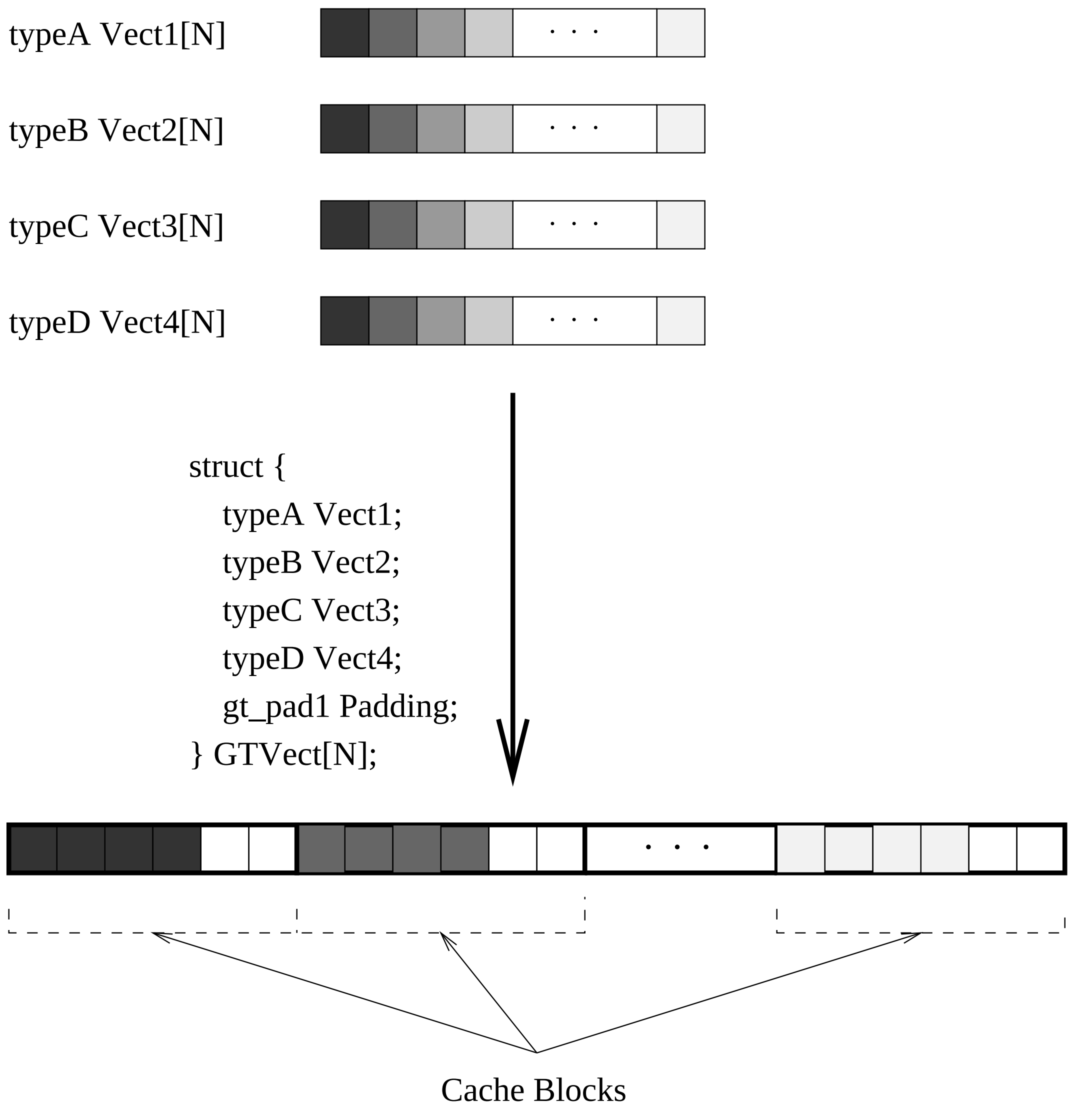


Figure 4.1: Illustration of the Group & Transpose transformation.

4.2 Indirection

For data structures where the number of data objects and their locations are unknown at compile-time, group & transpose cannot be applied, since there is no static data structure to transform. This case occurs, for example, when the affected per-process data structure is embedded into elements of heap-allocated data structures, such as linked lists or graphs. Instead, a different transformation, called indirection, is used to contiguously allocate elements of a data structure that are accessed by the same

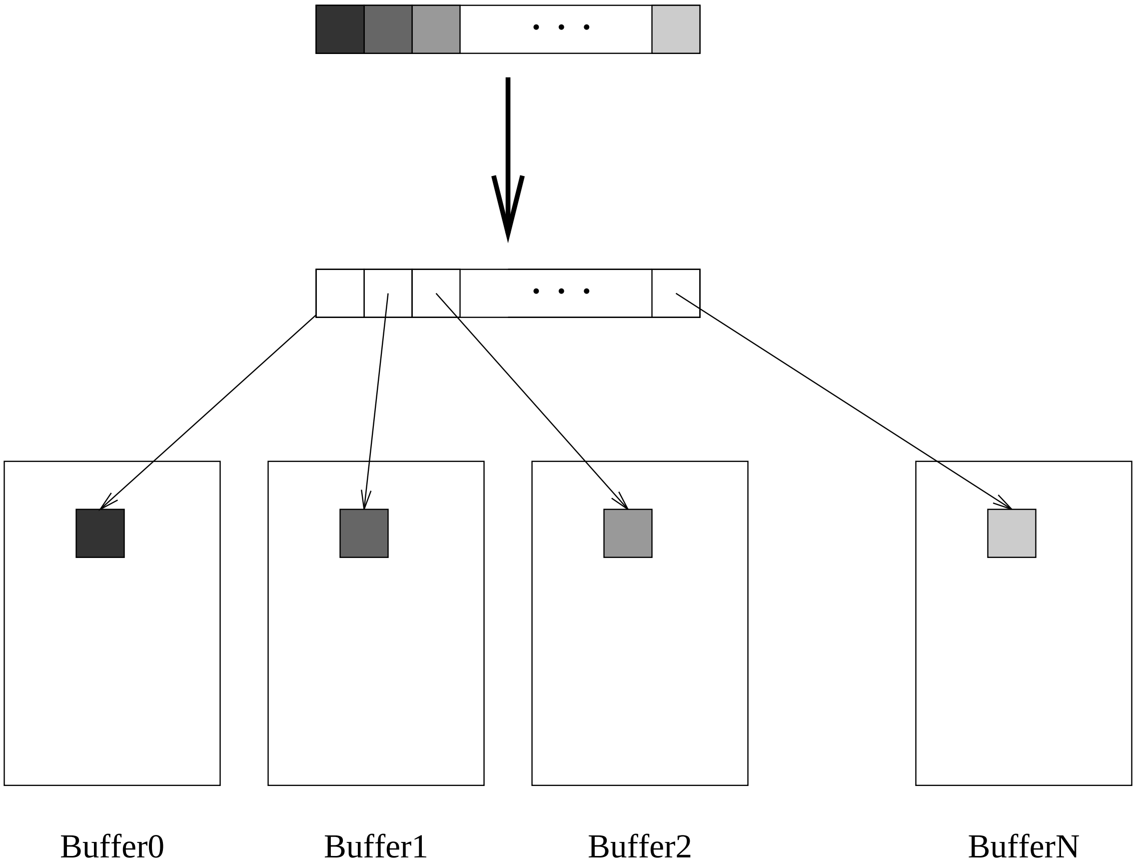


Figure 4.2: Illustration of the Indirection transformation.

process.

Indirection, shown in Figure 4.2, moves elements of a data structure that are only or predominantly accessed by the same process into a buffer allocated for that process. A pointer to each element's new location in the buffer is put into the original location of the element. This is achieved in three steps. First, the type definitions of the target elements of the per-process data structure is changed to pointers, and another level of indirection is added to the references to these data objects in the program. Second, code is inserted to create a large buffer in shared memory at run-time for each process in the program. Third, code is inserted after each statement that allocates objects of any of the affected data types. This code allocates space in the per-process buffers for the target elements and initializes the pointers in the changed data structures to point to them. An example of program code from the application Topopt before and after this transformation is shown in Figures 4.3 and 4.4 respectively.

```
typedef struct net {
     STRING netName;
     int num;
     int input;
     int output;
     int size;
     int row;
     int minp[13],maxp[13],nminp[13],nmaxp[13];
     CELL_PTR cell_list;
} *NET ;
NET np;
..
np = (struct net *) malloc (sizeof(struct net));
..
net->minp[proc] = net->nminp[proc] = INFINITY;
..
```

Figure 4.3: Example application before applying indirection.

In a data structure that suffers from false sharing because of per-process accesses (including writes) to its elements, indirection eliminates the false sharing because it prevents the data that is accessed by different processes from residing in the same cache block, and co-allocates data that is accessed by the same process. However, unlike group & transpose, it carries both a cost in time as well as space. The time cost results from the extra memory access to reference the restructured data, while the space cost is due to the additional pointers. Nevertheless, indirection provides a net benefit. Both the memory access to the pointer as well as the indirection through it are likely to be cache hits, while the memory reference to the element in the original data structure was likely to be a cache miss because of the false sharing. With the discrepancy between processor and memory speeds that exist today (and is likely to increase tomorrow), trading off a cache miss for two cache hits is quite profitable. For example, on the KSR 2 two cache hits take four cycles in the first level cache (40–48 cycles for the second level cache), while a false sharing miss (which occurs in the second level cache) can take up to 600 cycles. Similarly, any loss in spatial locality brought upon by the additional pointers is more than compensated for by the improvement in processor locality that results from the transformation.

```
void *_proc_buffer_alloc(size_t);

typedef struct net {
      STRING netName;
      int num;
      int input;
      int output;
      int size;
      int row;
      int *minp[13],*maxp[13],*nminp[13],*nmaxp[13];
      CELL_PTR cell_list;
} *NET ;
NET np;
..
_init_proc_buffers(13);
..
np = (struct net *) malloc (sizeof(struct net));
for (i = 0; i < 13; i++){
  np->maxp[i] = _proc_buffer_alloc(i);
  np->minp[i] = _proc_buffer_alloc(i);
  np->nminp[i] = _proc_buffer_alloc(i);
  np->nmaxp[i] = _proc_buffer_alloc(i);
}
..
*net->minp[proc] = *net->nminp[proc] = INFINITY;
..
```

Figure 4.4: Example application after applying indirection.

4.3 Pad and Align

Many data structures that suffer from false sharing in the short term are actually true shared by several, if not all, the processors over time. Padding the elements of these data structures to the cache block size and aligning them on cache block boundaries eliminates all false sharing, since it results in only one data object per cache block. However, the extra space increases the data set size, and may therefore increase conflict and capacity misses, as well as reduce spatial locality when processors access the entire shared area. These negative effects of padding can be avoided by applying it judiciously. In order for spatial locality to benefit write-shared data, it

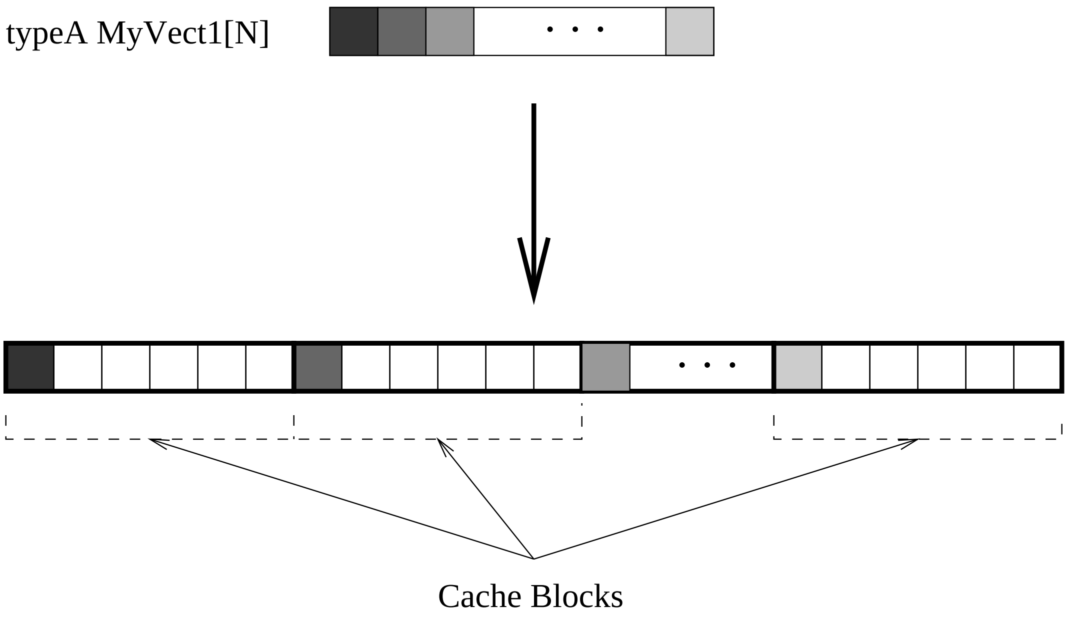


Figure 4.5: Illustration of the Pad and Align transformation.

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, pad and align can be beneficially applied to data structures that lack processor locality, i.e., where the possible loss of spatial locality is insignificant relative to the savings in coherency overhead.

4.4 Heuristics - Going from Side-Effects to Transformations

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 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. The actual transformation applied depends on the type of data structure that is to be transformed. The heuristics are detailed in Table 4.1.

Group & transpose and indirection are only applied to vectors. However, because of its lower space and execution time costs, group & transpose is the preferred transformation. Indirection is only applied in its place if the number of vectors to

Table 4.1: Heuristics for applying transformations.

Data type	Transformation	Conditions		
lock variables	pad & align			always
scalars	pad & align			write frequency $\geq 1,000,000$
vectors	pad & align			write frequency $\geq 1,000,000$
			and	no access have spatial locality
			and	not part of replicated structure
	group & transpose or indirection			number of per-process writes $\geq 1000\times$ number of non per-process writes
			and	reads have no spatial locality
		or		number of per-process writes $\geq 1000\times$ number of non per-process writes
			and	number of per-process writes $\geq 10\times$ number of non per-process reads
		or		number of per-process writes $\geq 10\times$ number of non per-process writes
			and	per-process writes occur alone in $\geq 75\%$ of the phases
			and	non per-process writes occur in $\leq 25\%$ of the phases
			and	reads have no spatial locality
		or		number of per-process writes $\geq 10\times$ number of non per-process writes
			and	per-process writes occur alone in $\geq 75\%$ of the phases
			and	non per-process writes occur in $\leq 25\%$ of the phases
			and	number of per-process writes $\geq 10\times$ number of non per-process reads

be transformed cannot be determined at compile-time, e.g., if they are components of heap-allocated data structures. In order to apply either group & transpose or indirection to a data structure, the pattern of writes to the data structure must be predominantly per-process (less so if the per-process writes occur alone in at least 75% of the phases) and the pattern of reads either per-process or without spatial locality. If the pattern of reads is read-shared with locality, the data structure is transformed only if the number of per-process writes exceed the number of non per-process 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.

Pad & align may be applied to lock and scalar variables as well as vectors. Locks are always padded; scalars are padded if they are busy; and vector elements are only padded and aligned on cache block boundaries when they are busy and both the reads and the writes exhibit sharing without spatial locality.

The heuristics were developed and tuned in an iterative process based on the comparison of the number of false sharing misses incurred by each data structure with its statically determined reference pattern. Simulations were used to determine the distribution of false sharing misses across the address space of each program. Using both symbol tables from the executables, as well as information about addresses and types of `malloc`'ed data (gathered during the tracing run), the false sharing misses were attributed to individual data structures. Based on the per-process side-effect information heuristics were developed to identify and choose transformations for the data structures that incurred the most false sharing misses. The transformations were applied according to the heuristics, and the result evaluated using simulation. Heuristics that either produced false positives, failed to indicate data structures with a substantial amount of false sharing, or led to transformations that caused cache performance degradation (e.g., by indicating pad & align too often) were discarded. This process was repeated until the set of heuristics produced the largest overall drop in false sharing misses, while keeping the negative impact on spatial locality low.

Chapter 5

METHODOLOGY

5.1 Introduction

Evaluating the merits of compiler-driven shared data restructuring requires that data be collected and evaluated for each of the constituent parts of the analysis, as well as the system as a whole. The runtime cost and accuracy of the compiler algorithms are measured by timing the runs of the compiler on the programs in the workload and comparing the output of the algorithms to that of manually inspecting the programs. The number of false sharing misses the programs in the workload incur, both before and after transformations, are measured on a simulator that models a simple, bus-based multiprocessor. This serves both to evaluate the magnitude of the problem of false sharing misses in the original programs, as well as to measure the cache effects of the compile-time analysis and transformations. Finally, any actual run-time performance improvements brought about by the reduction in false sharing misses are measured by examining the performance of directly executing the programs in the workload, before and after transformations, on a KSR 2 shared-memory multiprocessor.

The next section describes the Parafrase-2 source to source restructurer, in which the compile-time analysis algorithms have been incorporated. Section 5.3 describes the multiprocessor simulator and the simulation methodology used to measure the number of false sharing misses, while the last section describes the KSR 2 multiprocessor.

5.2 Compiler

Parafrase-2 [PGH$^+$89] is a source-to-source restructurer developed at the University of Illinois at Urbana-Champaign. The version of Parafrase-2 I used consists of a C front end (scanner, parser and semantic analyzer) and a set of selectable passes that operate on the intermediate representation built by the front end. The intermediate representation is a parse tree-based, high level representation that preserves the name space and program constructs of the source program, so that it can be regenerated with any compiler-generated modifications. This intermediate representation is traversed, added to, or transformed by the sequence of passes specified when the compiler is invoked. The false sharing analysis added to Parafrase-2 consists of four such passes. Several of the original passes were modified and enhanced for accuracy of results, language completeness and run-time performance reasons. The most significant of these modifications were:

1. Parameters that in practice worked as reference parameters were recognized and marked. Reference parameter alias analysis was improved and expanded to include C.

2. The side-effect analysis was expanded to fully include C. Interprocedural analysis was improved and support for recursive programs was added.

3. Support for interprocedural analysis was added to the dependence analysis. Worst case assumptions were removed for procedure calls and pointer accesses.

4. Dependence analysis was made more accurate by removing "impossible" dependencies.

5. Dependence graph construction was sped up significantly (up to a factor of 10) by introducing tests to detect earlier when no dependences existed.

The set of passes invoked to analyze the per-process memory access pattern of explicitly parallel programs includes much of the analysis that is part of most optimizing compilers, in addition to the passes added to implement the false sharing analysis. The sequence of analysis is described below.

The first pass is a "fixup" pass that insures the integrity of the intermediate representation, by making links between different structures of the intermediate representation that were not made during the parsing of the source program. Next, semantic and side-effect information about library functions and separately compiled units are read in and added to the symbol table and the intermediate representation[1] This is the only pass that reads in any source program information.

The remaining passes traverse the intermediate representation, analyzing the program and building and annotating additional data structures. The first of these passes constructs a call multigraph[2], where each edge is labeled by its corresponding call site in the program. It is followed by a pass that computes reference parameter alias information. As parameters are passed by value in C, it is necessary first to detect which parameters function as reference parameters. This is done by detecting which parameters contain addresses and are not redefined within the body of the procedure. Reference parameter alias analysis of is then performed as described in [CK89].

After the reference parameter alias analysis the control flow graph of each function is constructed and dominators computed. This is followed by loop nest analysis, dependence graph construction, interprocedural summary side-effect analysis implemented (also as described in [CK88]), constant propagation and induction variable analysis. The remainder of the passes are specific to the analysis of detecting false sharing. They consist of per process control flow analysis, barrier synchronization analysis, per process static profiling, and per-process summary side-effect information. These passes are described at length in Chapter 3.

5.3 Trace Driven Simulation

Most of the cache metrics reported in this dissertation were obtained using trace-driven simulation. This choice was made due to the availability of tracing tools and because memory reference traces provide a detailed picture of the memory behavior of each process, which enables the multiprocessor simulator to accurately detect and categorize cache misses. This section describes how the traces were produced, the architecture of the simulator, as well as the simulated architecture. Also, the accuracy and applicability of the simulation results with respect to real world architectures is discussed. For a discussion on the benefits and drawbacks of trace-driven simulation, see [Egg89].

---

[1] The side-effect information was produced by other runs of Parafrase-2, or if no source code was available, it was produced by hand based on man-page descriptions. Semantic information was also added by hand.

[2] A multigraph is a graph where multiple edges may exist between the same pair of vertices.

The memory reference traces were generated on a 20 processor Sequent Symmetry Model B [LT88], using a software tracing system for parallel programs called MPtrace [EKKL90]. MPtrace uses a technique called “inline tracing”, where programs are annotated to save just enough information when the program is run to reconstruct each process' sequence of memory references. Traces are generated in a three-stage process. First assembly code of the original program is analyzed and annotated with trace generation code by a preprocessing tool, and then assembled and linked into an executable. Additional information about the program, such as the addresses of labels in the executable, is also saved to facilitate the complete reconstruction of the memory reference trace from the information saved at run-time. Second, the executable from stage one is run on its normal input; the added code, together with a customized runtime system, collects and writes time-stamped trace data to disk. Third, a postprocessor uses the information saved in stage one and the trace data generated in stage two to reconstruct the sequence of memory references for each process.

Each memory reference is described in a trace record. In addition to the address, the trace record also contains information that specifies whether a reference is an instruction or data. An instruction reference is further categorized as either a regular instruction, a lock instruction, a locking sequence instruction, an unlock instruction, a barrier entry instruction, or a barrier exit instruction. Data references are classified as either reads or writes to private or shared data, and whether the data is a normal, lock or unlock operand. Note that instructions executed while busy waiting for locks or barriers are not included in the trace. Instead the simulator models all synchronization and inserts busy-waiting instructions and data references as described below.

In order to limit the size of the files containing the trace records, the traces are run through a cache filter and also compressed [Egg89]. The cache filter is used to runlength encode sequences of trace records for instructions and private data references that hit in a very small, direct mapped cache. Since these sequences of memory references will hit in any larger size cache and, because they do not reference shared data, and will not affect nor be affected by any coherency transactions, each such sequence can be replaced by a single trace record that specifies the number of its instruction and data references.

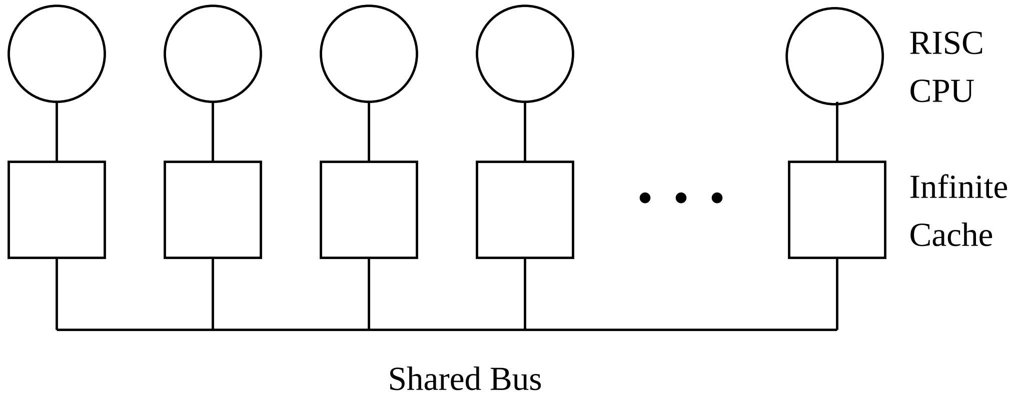


Figure 5.1: Simulated Architecture.

The simulator is a modification of Charlie [Egg89], which simulates a simple bus-based, shared-memory multiprocessor as depicted in Figure 5.1. The simulator can simulate up to twelve RISC-like processors. The processors execute each instruction in a single cycle. Each processor has a private cache connected to a shared bus. Timing of the bus was similar to the SPURbus [Gib88]. However, accesses to the bus were arbitrated in a round robin order instead of that of the NuBus protocol which gives preference to the higher priority cache controllers. The cache coherency protocol used in the simulations was the Berkeley Ownership Protocol [KEW$^+$85]. In this protocol each cache block may be in one of four states: invalid, valid, shared-dirty or private-dirty. A cache that has a cache block in state shared-dirty or private-dirty is the owner of that block. In the case where a cache block is not owned by any cache, main memory is the owner. Writes transfer ownership to the cache of the writing processor, while replacements transfer ownership to main memory. Writes cause invalidations except if the target cache block is owned in state private-dirty by the writing processor. Whenever a read miss occurs, the owner of the target cache block is responsible for supplying the requestor with the requested cache block. If a cache needs to replace a cache block that it owns, it is responsible for writing the contents of that block back to main memory.

The simulator is implemented as a set of light weight tasks that communicate via message queues and a global shared memory [Fuj83]. There are two sets of clocks in the simulator. A global clock indicates the overall time of the simulated multiprocessor, while clocks local to each task are used to time each particular function, e.g., cache lookup. Scheduling of tasks is done according to increasing local clock time.

Each processor task reads in trace records one at a time from a trace file and issues the memory references to its cache controller task. In case of cache misses or writes to shared data, the cache controller task may initiate bus transactions that are simulated by the bus task. When the time to complete the request and any bus transactions has passed, the cache controller task replies to the processor task, which allows it to proceed. The progress of each of the simulated processes is only constrained by the simulated time it takes to service the memory requests (including contending for the bus and cycles lost when the processor is locked out of the cache due to snoop operations) and by synchronization operations.

As mentioned above, the memory reference traces may contain trace records that specify either lock operations or global barrier synchronization events. These are used by the simulator to simulate the same mutual exclusion and synchronization operations that occurred in the execution of the original program, but with the timing dictated by the simulated architecture. This is done by having the simulator track and maintain the states of barriers and lock variables. When a lock data operand is encountered, the simulator first looks up the status of the lock in a table using the address of the memory reference as the key. If the status reflects that no other processor has that lock, the simulation of the locking processor is allowed to proceed within its trace. If, on the other hand, another processor already has attained the lock, the simulator injects busy waiting references into the memory reference stream of the locking processor until the status of the lock changes. At this point the locking sequence is restarted. Even though busy waiting instruction and data references are synthesized by the simulator and not consumed from the actual memory reference trace, they are accounted for when computing execution cycles for the simulation.

Global barrier synchronization events are simulated similarly to locks in that no processor may consume any references from the trace after the barrier until the all of the simulated processors have checked into the barrier.

The goal of the simulations in this research are to measure the performance degradation due to false sharing and the benefit of eliminating it, particularly in large second level caches (on the order of several megabytes). To that end all simulations assume an infinite cache. Cache block sizes were varied from 4 to 256 bytes. Following a cold start interval, approximately two million references were simulated for each processor in each simulation, and most simulations were done for twelve processors.

The simulator was modified to detect false sharing by augmenting the cache state with a tag for each word in every cache block and tracking invalidation signals and invalidation misses. Whenever a cache block is invalidated, the referenced word in the cache block is marked shared in all caches; the remaining words are marked non-shared. Additional writes to other words in the cache block are marked similarly. False sharing misses occur whenever there is an invalidation miss on a cache block, and the referenced word is marked non-shared.

While efforts have been made to make the results of the simulations as accurate as possible, there are several approximations that cause the results to deviate from that of real computer systems. First, the traces where collected on a Sequent Symmetry equipped with Intel 386 processors, while the simulator is assumed to simulate a multiprocessor with RISC style processors. Because of the special purpose and limited size of the register set on the Intel 386, it would presumably access memory more often, and therefore the traces would be expected to contain more memory references than if a modern RISC processor had been used. However, because the compiler does not determine when the values of shared data may be updated, it must conservatively assume that shared data may be changed by any process at any time. Therefore, shared data needs to be treated as volatile, and can therefore not be allocated to registers across multiple uses. Thus the increase in memory references should be mostly restricted to private (to the process) data. Because this research focuses on false sharing of shared data, and because private data cannot replace shared data in the infinite caches, no measurable perturbation of results other than increased bus utilization should occur.

Second, while the simulator assumes single-cycle execution of every instruction, there are instructions that take longer, such as floating point instructions. The effect is that bus utilization is overestimated; however, cache miss rates remain mostly unaffected, and thus so does false sharing.

Lastly, while using an infinite cache may reduce the overall cache miss rate below what is realizable with real caches, they have been shown to approximate very large second level caches well [Egg91]. Moreover, the use of an infinite cache eliminates capacity and conflict misses, which are direct functions of the cache size. Therefore all possible coherency misses, and false sharing misses in particular, are exposed. In addition, since there are no conflict misses, the application of data transformations

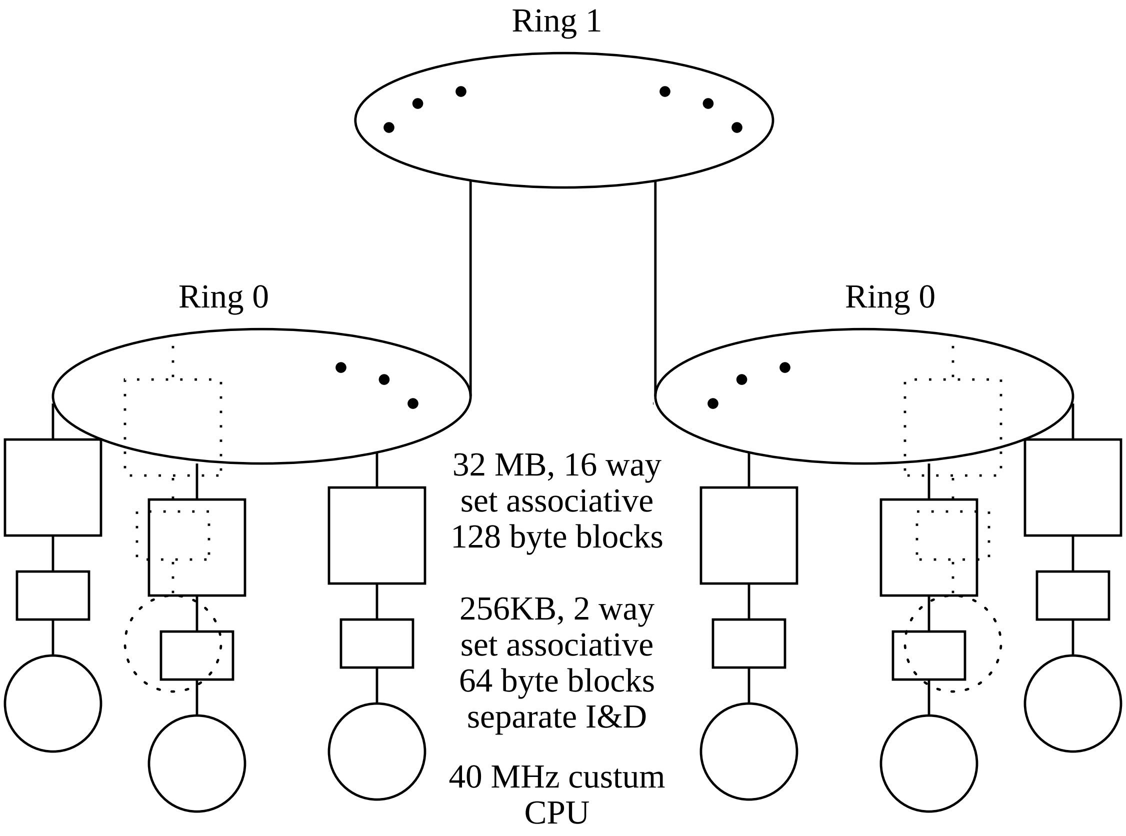


Figure 5.2: KSR 2 Architecture.

to eliminate false sharing cannot generate or remove any mapping conflicts particular to a given cache organization that otherwise could have obscured their effect.

5.4 KSR 2

The KSR 2 [KSR92] is a shared-memory, all cache, multiprocessor made by Kendall Square Research. Figure 5.2 shows the architecture of the KSR 2 multiprocessor. It uses custom dual issue, superscalar processors connected to a ring-based, two level, hierarchical interconnect. Each base level ring can contain up to 32 processors, and the second level ring can be connected to as many as 32 base level rings. The machine used in this study had 56 processors divided into rings of 32 and 24 processors. Each processor has separate 256 KByte, first level, 2-way set associative instruction and data caches. These caches have 2 KByte cache blocks each of which is divided into 64 byte subblocks. The cache block serves as the allocation unit, while data is fetched in the unit of a subblock. Not all subblocks need be present in any particular cache block. The second level cache is unified, 16 way set associative, and contains 32 MByte, divided into 16 KByte allocation units, each consisting of 128 byte subpages. Cache coherency is maintained at the subpage level, which is also the unit of transfer. The cache coherency protocol is invalidation-based with explicit subpage ownership. Each subpage may be in the following states: invalid, read-only, owned non-exclusive, owned exclusive. Additional atomic states are only used for synchronization. Misses to a subpage are always serviced by the owning cache. In the case of writes, the subpage is invalidated and ownership is transferred to the requesting cache in exclusive state. In the case of reads, the owning cache supplies the requestor with a read-only copy of the subpage, and changes the state of the subpage to owned non-exclusive. Writes cause invalidations except when the subpage is owned in exclusive state by the cache of the writing processor. The latency for cache hits is 2 cycles for the first level cache, and 20-24 cycles for the second level cache. Second level cache misses have a latency of 175 cycles if they can be serviced within the same ring, and 600 cycles otherwise.

In addition to measuring program execution time, the KSR 2 multiprocessor has a number of hardware performance counters that allow a greater variety of metrics to be collected for each processor. Of particular interest for this research are the number of cache misses at each level and the total amount of time spent waiting for memory accesses.

Chapter 6

RESULTS

This chapter presents experimental results that demonstrate the impact of the compile-time analysis and transformations on the workload. The results attest to the ability of the analysis to identify data structures that incur false sharing misses; demonstrate the effectiveness of the transformations in eliminating them; and evaluate the impact on run-time performance impact.

The results are divided into three sections. Section 6.1 presents the results from applying the compiler to the set of unoptimized programs, while Section 6.2 compares the performance of the compiler with that of programmer hand-tuning. Section 6.3 summarizes the results for the programs for which all three versions were available: unoptimized, programmer-optimized and compiler-optimized.

The first two sections are structured similarly. First, results from trace-driven simulations are presented. These simulations measure the number of cache misses and characterize them either as false sharing or non-false sharing-related misses. The cache miss rates of the programs are compared before and after transformations to measure how well they eliminated false sharing. The relative contribution of each transformation is also presented.

Second, the performance impact of the transformations on the workload on the KSR 2 are measured by comparing the relative speedup of the programs before and after applying the transformations.

6.1 Compiler on Unoptimized Programs

6.1.1 Simulation Results

Since false sharing is a product of the layout of data in shared memory and the cross-processor reference pattern to it, the number of false sharing misses naturally varies significantly from one program to another. However, for all programs, as the results in this section confirm, the false sharing miss rate increases with the cache block size.

The compiler-directed shared data transformations were effective in reducing false sharing in all programs for all block sizes, regardless of the size of the original false sharing miss rate. The greatest relative reductions were achieved for Fmm, Pverify and Radiosity, the greatest absolute reductions for Maxflow, Pverify and Topopt.

Following are the results and analysis of the simulations for each of the programs in the workload.

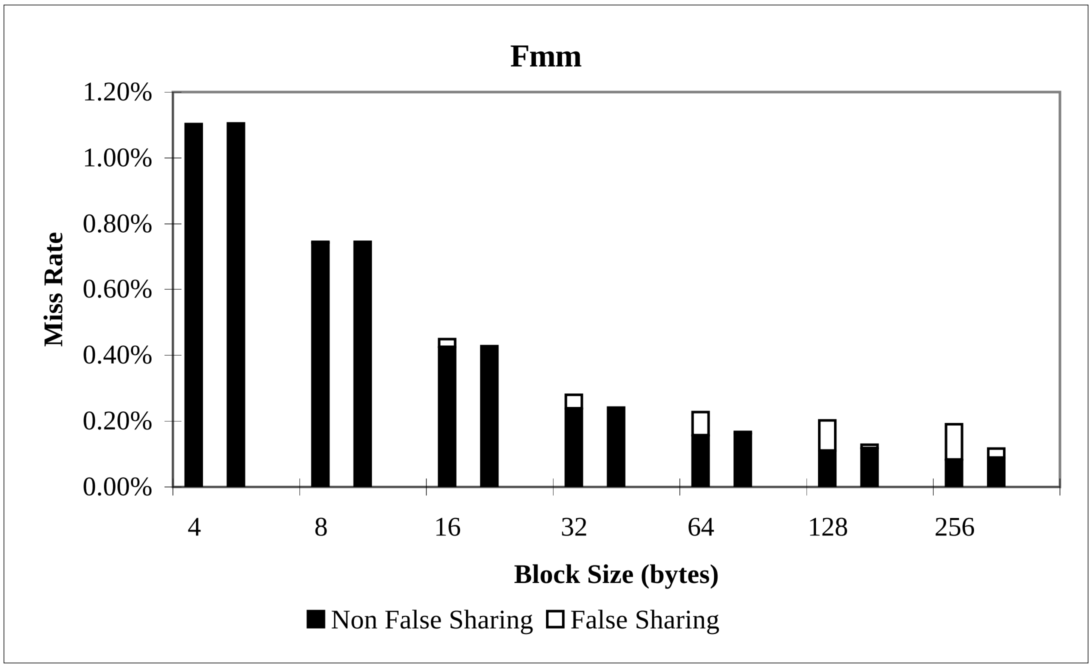


Figure 6.1: Total and false sharing miss rates, before and after applying transformations – Fmm, unoptimized.

Fmm

Figure 6.1 shows the total cache miss rate for Fmm, before and after applying the compiler-driven transformations, for cache block sizes varying from 4 to 256 bytes. The white portion of each bar represents the portion of the miss rate that is 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 figure illustrates that for the smallest cache block sizes, false sharing is not a significant source of cache misses for Fmm. However, for 64 byte cache blocks and larger, between one and two out of every three cache misses is due to false sharing. Most of these misses (84.8% average over 8 to 256 byte cache blocks) are the result of accesses to nineteen global vectors whose elements are allocated across the processes. That is, each vector element is only accessed by a single process, but is allocated in the same cache block as elements accessed by other processes. The compiler detected this access pattern based on the recognition of process differentiating variables (see Chapter 2) and their use in the index expressions of these vectors. Since all of these vectors were statically allocated and accessed similarly, that is, the same process accessed element i in each of the vectors), they were all transformed using group and transpose. In addition to restructuring this set of vectors, the lock variables in the program were padded and aligned on cache block boundaries. Together, these transformations succeeded in eliminating an average of almost 91% of the false sharing misses across multi-word cache block sizes, with no significant negative impact on spatial locality.

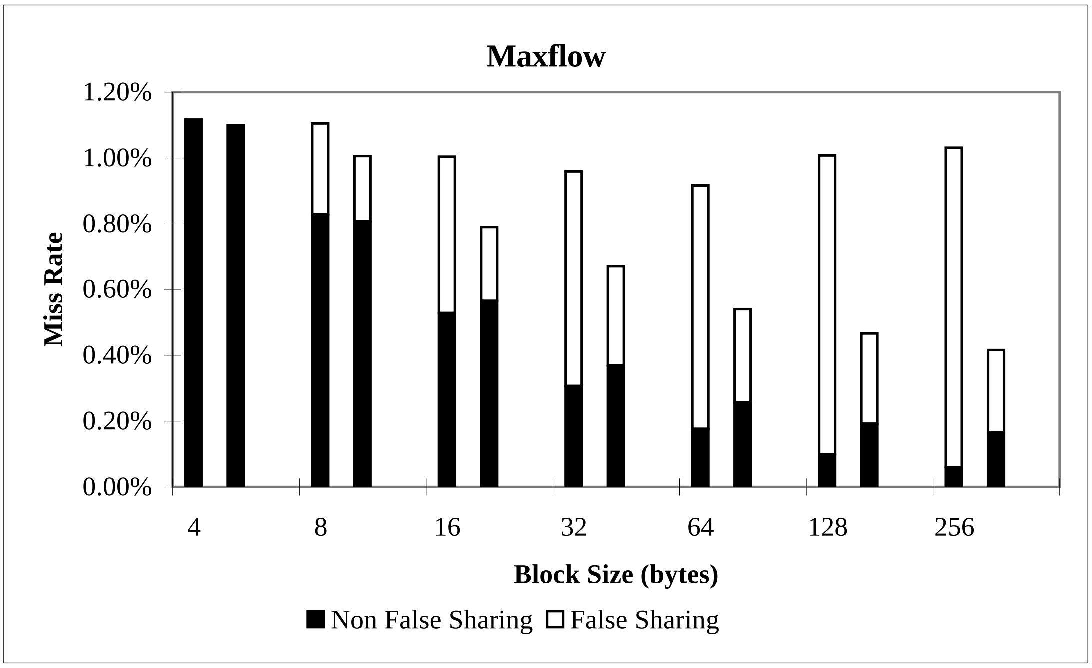


Figure 6.2: Total and false sharing miss rates, before and after applying transformations – Maxflow.

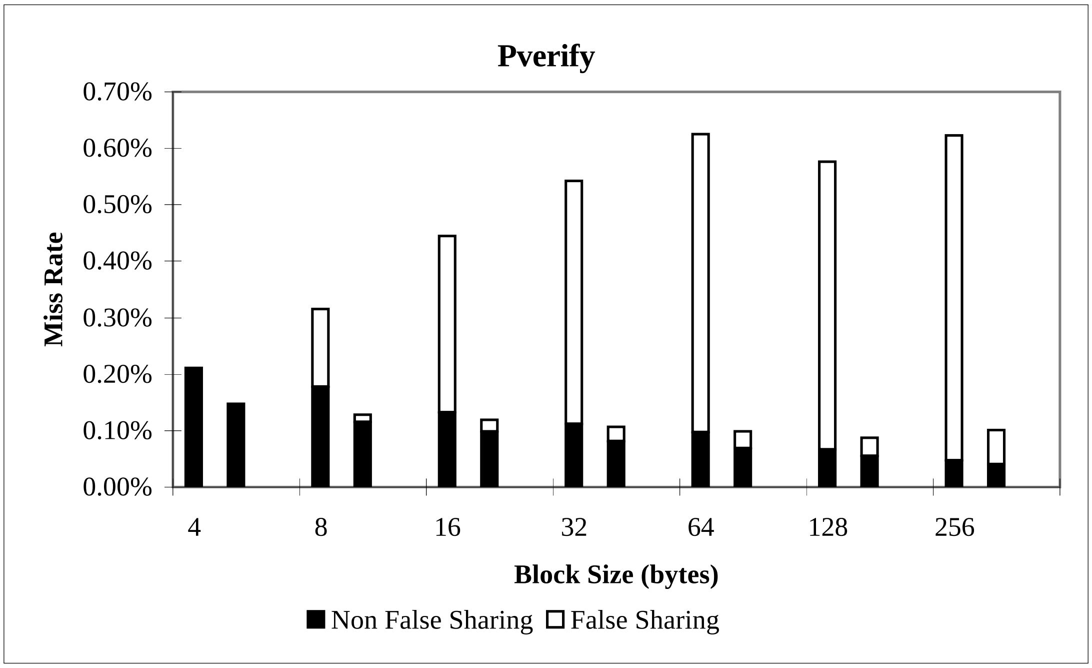


Figure 6.3: Total and false sharing miss rates, before and after applying transformations – Pverify.

Maxflow

Figure 6.2 shows that that false sharing is a much greater problem in Maxflow. In fact, the number of false sharing misses in Maxflow grows faster than the prefetching benefit of larger cache block sizes. Of the false sharing misses, almost half are incurred by a single vector that implements process work queues, and in which each element is only accessed by a single processor by using a process differentiating variable in the index expression. About a tenth of the false sharing misses are due to lock accesses, while most of the remaining are caused by a few busy, write-shared scalar variables that are allocated to the same cache blocks.

Based on the use of the process differentiating variable, the compiler analysis correctly identified the process work queue vector as a target for transformation. Since this was the only vector to transform, group & transpose could not be applied. Instead, the elements of the vector were padded and aligned on cache block boundaries, as were the lock variables. The busy scalars were not transformed (padded and aligned), because the static profiling underestimated their dynamic access frequencies, making the cost of transformation, i.e., the loss in spatial locality due to padding, appear greater than the benefit of eliminating the false sharing misses. Nevertheless, the transformations that were applied eliminated on average 56.5% of the false sharing misses. However, this came at the cost of a reduction in spatial locality, as shown by the change in the size of the black portion of the bars. All in all, the reduction in false sharing misses far outweighed the decline in spatial locality, resulting in a net reduction of the overall cache miss rate by an average (across multi-word cache blocks) of 35.7%.

Pverify

Pverify (Figure 6.3) is a program in which the false sharing miss rate is so large that it determines the shape and trend of the overall miss rate. In fact, this occurs to such a degree that the overall cache miss rate is tripled as the cache block size increases from 4 to 256 bytes.

Approximately 80% of the false sharing misses occur as results of accesses to two vectors that are embedded in (as components of) a structure that is allocated and replicated dynamically. Each such structure represents a wire connecting a pair of gates in the circuits given as input to the program. The vector elements are used to hold state and partial results for the processes, and are write-shared among the processes over the lifetime of the program. However, each vector element is predominantly accessed by only one process, thus causing false sharing misses when multiple elements reside in the same cache block. The other data structures that incur significant numbers of false sharing misses are two global vectors accessed in a per-process manner (6%), and lock variables (3%).

The compile-time analysis determined that the vectors embedded in the dynamically allocated structures were write shared. However, the use of multiple regular section descriptors for these vectors, coupled with static profiling information, showed that the processors predominantly accessed separate portions of the vectors, using process differentiating variables. Similarly, the two global vectors were transformed based on the use of process differentiating variables in their index expressions.

Since the vectors most responsible for false sharing were components of a structure that was replicated dynamically, they could not be restructured using group & transpose. Instead, indirection was applied to their elements. Group & transpose was, however, applied to the two global vectors, and pad & align was applied to the lock variables. The result was a 91.2% average (over multi-word cache block sizes) reduction in false sharing misses, and a proportional reduction in the overall miss rate, with even a slight improvement in spatial locality.

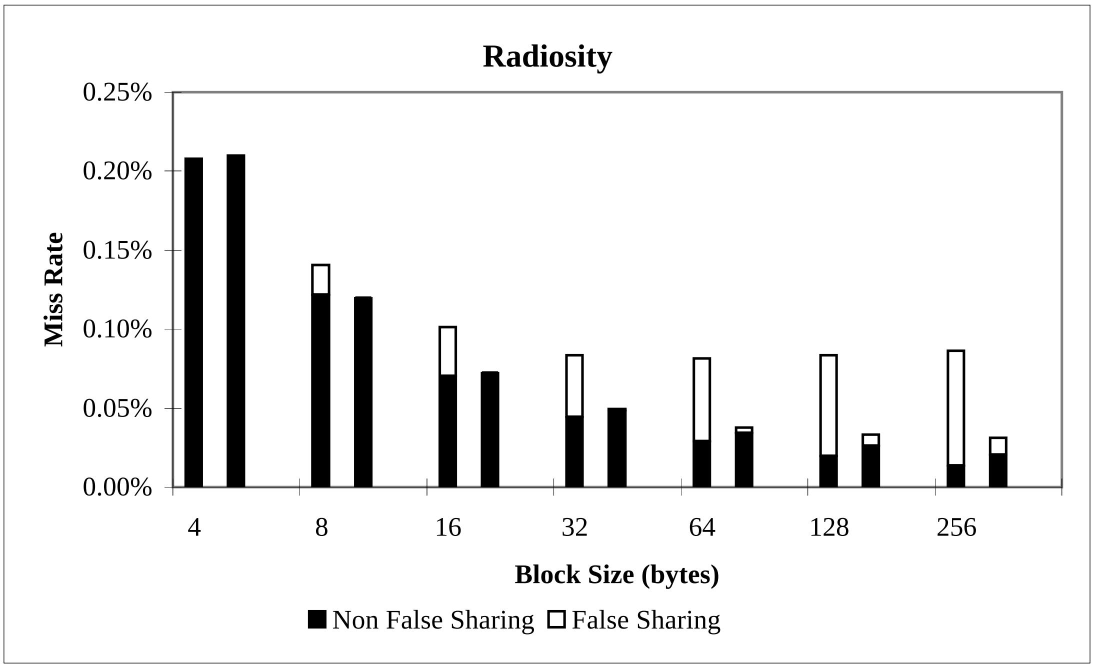


Figure 6.4: Total and false sharing miss rates, before and after applying transformations – Radiosity, unoptimized.

Radiosity

Although the overall miss rate in Radiosity (Figure 6.4) is fairly small (less than 0.1% for 16 byte cache blocks and larger), many of the remaining misses are due to false sharing. Several sets of global vectors used in process task management, small object management and the computation of visibility values are accessed in a per-process manner. Together they are responsible for about 85% of the false sharing misses. The compiler analysis correctly identified these vectors as likely to incur false sharing misses, and they were transformed using group & transpose. In addition, the compiler also pointed out several busy scalars to be transformed using pad & align. However, these scalars were allocated among lock variables that had already been

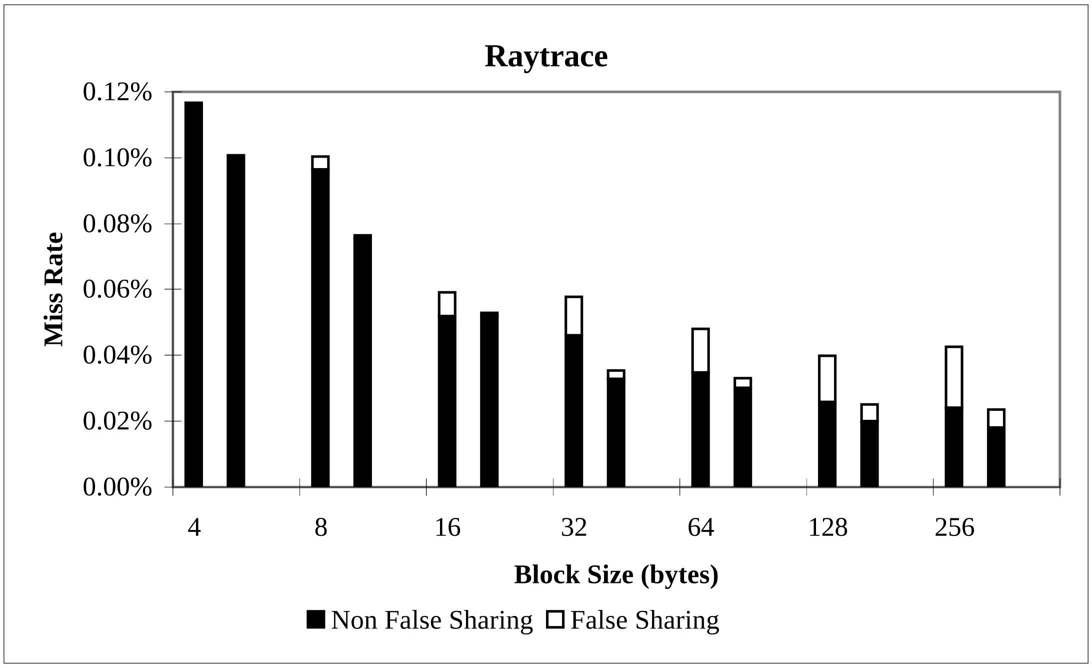


Figure 6.5: Total and false sharing miss rates, before and after applying transformations – Raytrace, unoptimized.

padded and aligned. Therefore, the additional padding afforded to the scalars had only a marginal benefit, since it served mostly to separate items that were already padded and aligned on cache block boundaries. In total, the transformations reduced the false sharing miss rate by an average of 93.5% and the overall miss rate by an average of 43.6%.

Raytrace

False sharing is less of a problem in Raytrace (Figure 6.5) than in most of the other untransformed programs. Nevertheless, for the largest block sizes, about one out of every three cache misses is due to false sharing. Again, almost three-quarter of the false sharing misses are caused by a small set of vectors that are referenced in a per-process pattern. These vectors are used to implement a shared work pool and per-process stacks for reflected and refracted rays. The per-process reference pattern was recognized by the compiler, and all but one of these vectors were transformed using group & transpose. The last vector was a member of a (statically allocated)

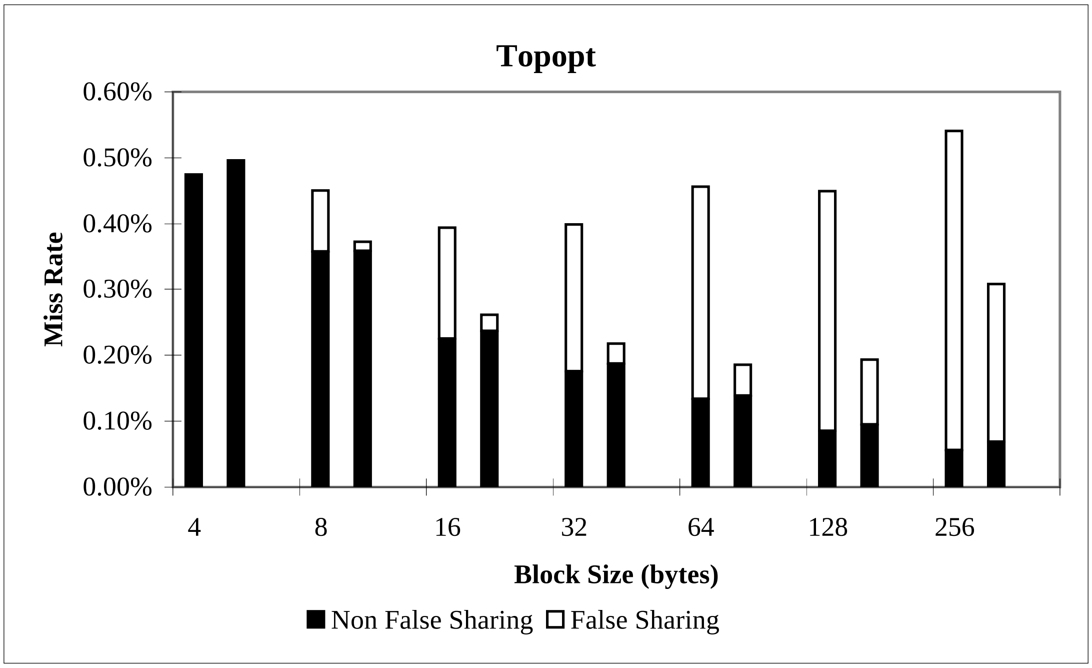


Figure 6.6: Total and false sharing miss rates, before and after applying transformations – Topopt.

structure, so instead its elements were padded and aligned. Padding and aligning the lock variables reduced the false sharing miss rate by an additional five percent, for a total average reduction in the false sharing miss rate of 78.3% and an average reduction in the overall cache miss rate of 34.5%.

Topopt

In Topopt (Figure 6.6), as in Maxflow and Pverify, the overall cache miss rate is dominated by false sharing misses. The primary source of false sharing is a set of four global vectors that describe the left and right boundaries as well as the maximum height of each per-process partition of the circuit representation. These vectors are write-shared, but only in one out of four phases in the main loop of the program. In the remainder of the program they are accessed in a per-process manner, causing an average of 61.3% of the false sharing misses. A second source of false sharing is a set of four adjacent vectors that are embedded in a dynamically replicated structure that represents an element of the input circuit. All the vector elements except one in each of two of these vectors are used as per-process accumulators, the last two are write-shared across the processes and are busy. These vectors are responsible for 18.6% of the false sharing misses. The majority of the remaining false sharing misses occur in a single array that is dynamically partitioned across the processes, where the partitioning changes in a revolving manner at run-time. As a result of the compiler analysis, the first set of four vectors were transformed using group & transpose. Since the second set of vectors were components of dynamically replicated structures, indirection had to be applied instead, with the write-shared elements being allocated to separate cache blocks. The last array was not transformed, because the static analysis had to conclude that its elements were true shared due to the dynamically changing reference pattern.

Table 6.1: Average (over multi-word block sizes) reduction in false sharing misses by different transformations.

Program	Total reduction in false sharing	Fraction of reduction by transformation			
		Group & Transpose	Indirection	Pad &Align	Locks
Fmm	90.8%	84.8%			6.0%
Maxflow	56.5%			49.2%	7.3%
Pverify	91.2%	6.4%	81.6%		3.1%
Radiosity	93.5%	85.6%		1.0%	6.8%
Raytrace	77.8%	70.4%		3.3%	4.6%
Topopt	79.9%	61.3%	18.6%		

Summary

Although, overall, the transformations were 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, since source code level analysis is required to ensure that only data that are accessed by the same process are grouped together. The static analysis for the compiler-driven transformations provides 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) 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, the reduction in false sharing more than compensated for any decrease in spatial locality, and the total miss rate fell.

6.1.2 Execution-Time Results

Eliminating false sharing affected two aspects of overall program performance: execution time and program scalability. The difference in execution 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 only the easily identifiable programmer transformations were undone 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 was small (Radiosity), or (3) the reduction in false sharing misses, although large in relative terms, 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 inter-processor contention for data structures that were 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 version of four of the six programs (Fmm, Maxflow, Radiosity and Raytrace). The performance of the transformed version of these programs continued to improve, reaching maximum

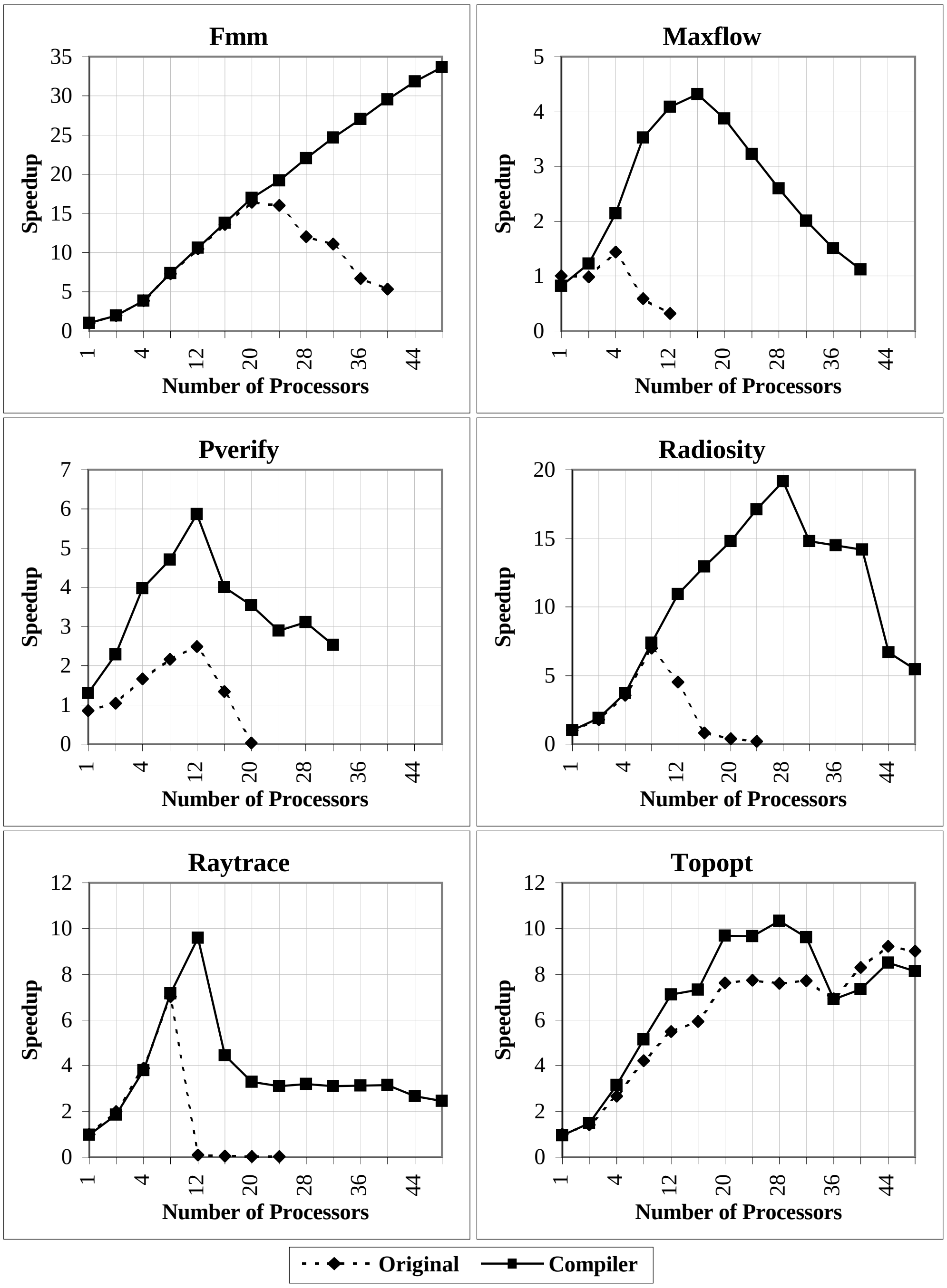


Figure 6.7: Speedup and scalability (speedup vs. number of processors) of unoptimized and compiler-optimized versions of the workload. Speedups are relative to the single processor execution of the unoptimized version of each program.

scalability at a greater number of processors (as shown in Figure 6.7). 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 higher (Table 6.2). This maximum performance difference is particularly striking for Fmm, Maxflow, Pverify, and Radiosity, where the transformed versions exceeded the maximum speedup of the original by factors of 2.0, 3.1 2.4 and 2.7, respectively.

Table 6.2: Maximum speedups for original and compiler-optimized versions and the number of processors at which they occur.

Program	Maximum Speedup and Scalability (# of processors)			
	Original		Compiler	
Fmm	16.4	(20)	33.6	(+48)
Maxflow	1.4	(4)	4.3	(16)
Pverify	2.5	(12)	5.9	(12)
Radiosity	7.0	(8)	19.2	(28)
Raytrace	7.0	(8)	9.6	(12)
Topopt	9.2	(44)	10.3	(28)

Even though the maximum performance of the compiler-optimized version of Topopt is greater than the unoptimized version, Figure 6.7 shows that for 40, 44 and 48 processors the unoptimized version has slightly better performance. This is in the range of processors where the program is no longer executing within a single ring on the KSR 2. Data from the KSR 2 performance monitor show that the difference in performance is due to the compiler-optimized version generating a greater number of requests across the top level ring. This additional traffic is due to the application of the indirection transformation (recall indirection was applied to a set of vector elements, including elements that were write-shared and busy). The per-process buffers used to hold the data of the original data structures (see Section 4.2) are allocated by the master processor in the initial (sequential) part of the program. When processors from more than one ring are used, those that are part of the other ring incur very long latency misses to this data (600 cycles versus 175 cycles), as well as the indirection. Even though false sharing causes more cache misses in the unoptimized version than in the compiler-optimized version, it has the side-effect of increasing the “ring locality” of the falsely shared data. In other words, because the cache block is falsely shared among the processors, some of which are on the local ring, the likelyhood that it can be found (supplied by a cache-to-cache transfer) without going over the top level ring is improved. This in particular benefits the accesses to the write-shared vector elements that incur cache misses as a product of the sharing pattern of the program.

6.2 Compiler versus Programmer Optimized Programs

The programs presented in this section have all been subject to extensive programmer optimizations to eliminate false sharing and improve memory hierarchy performance. The amount of false sharing is substantially less in these programs than in the unoptimized workload. Still, the compiler analysis improves upon every programmer-optimized program. The improvements range from marginal to substantial.

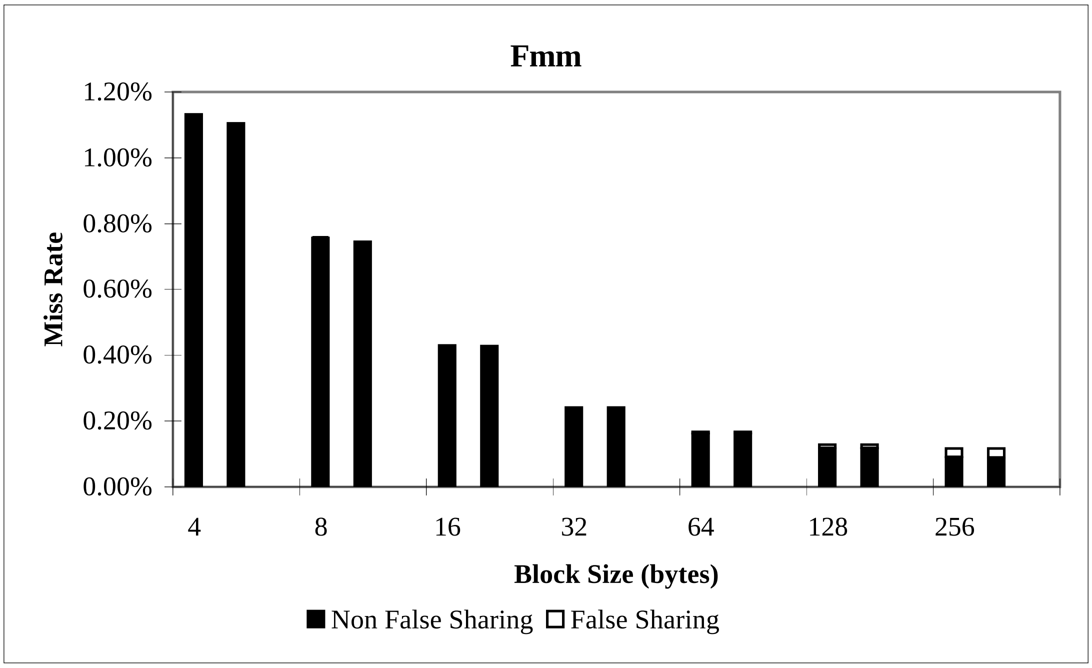


Figure 6.8: Total and false sharing miss rates for programmer and compiler-optimized versions of Fmm.

For Fmm, Radiosity and Raytrace, the programmer-optimized version is compared to the compiler-optimized version of the unoptimized program, thereby providing a direct comparison between the compiler-directed and programmer efforts. Since there were no unoptimized versions available for LocusRoute, Mp3d, Pthor and Water, the programmer-optimized versions of these programs had to be used as the input to the compiler. Thus, they indicate the improvements possible over hand-tuned code.

6.2.1 Simulation Results

Fmm

There is almost no difference in the miss rates between the programmer and compiler-optimized versions of Fmm (Figure 6.8). In the programmer-optimized version group & transpose was applied to one more vector than in the compiler-optimized version. However, locks were padded and aligned in the compiler-optimized version only. Therefore, while the amount of false sharing is similar in the two versions, it occurs in different data structures. The run time results (Section 6.2.2) will show that the false sharing in these two data structures does scale differently with the number of processors.

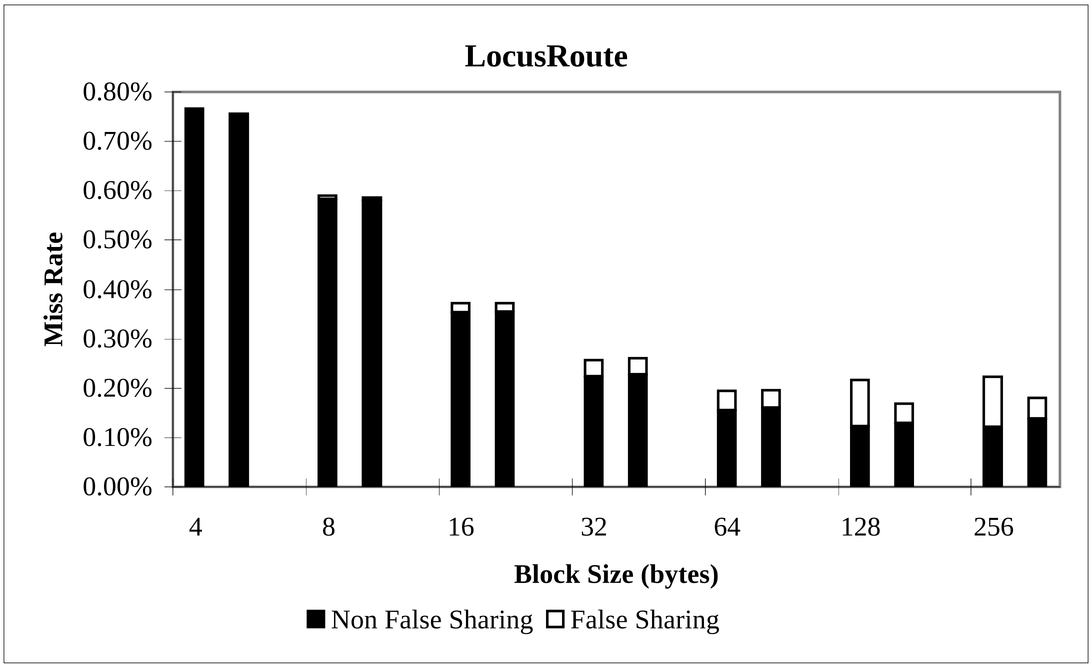


Figure 6.9: Total and false sharing miss rates for programmer and compiler-optimized versions of LocusRoute.

LocusRoute

In LocusRoute (Figure 6.9) the only difference in optimization between the two versions of the program was that pad & align was applied to the locks in the compiler-optimized version but not in the programmer-optimized version. Thus, the miss rates for the two versions are very similar for all but the largest cache block sizes, where the transformations in the compiler-optimized version more than halved the false sharing miss rate of the programmer-optimized version. On average across multi-word cache blocks, the compiler-directed transformations reduced the false sharing miss rate by 24% relative to the programmer-optimized version. The padding did not significantly affect spatial locality, so the reduction in false sharing translated into a 6.8% average reduction in the overall cache miss rate compared to the programmer-optimized version.

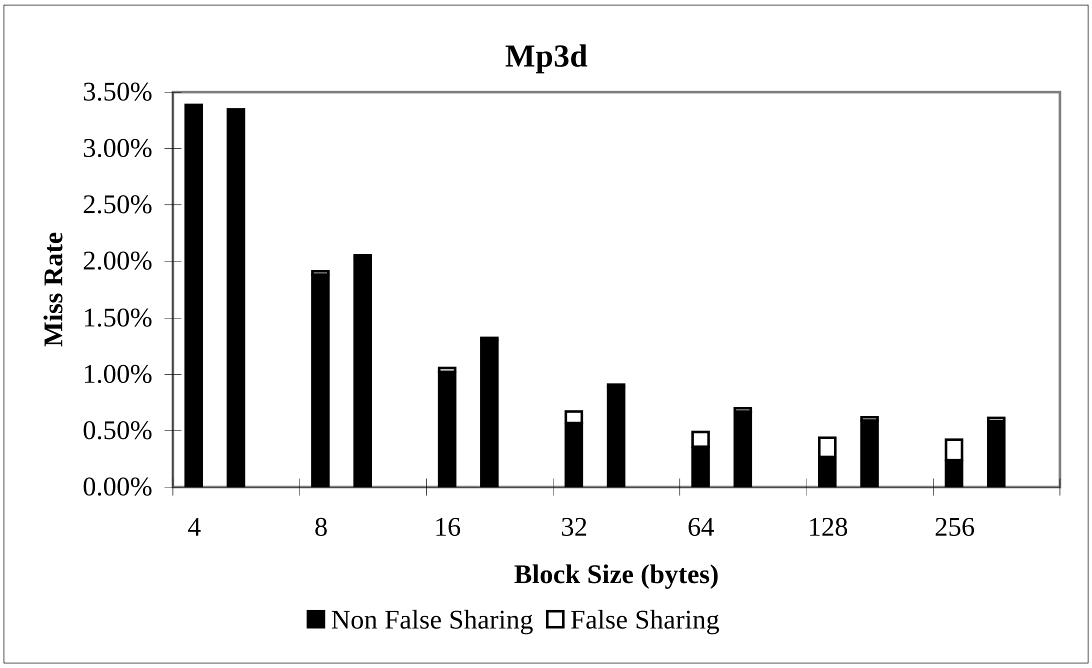


Figure 6.10: Total and false sharing miss rates for programmer and compiler-optimized versions of Mp3d.

Mp3d

On average, two thirds of all false sharing misses in Mp3d (Figure 6.10) can be attributed to lock variables. Since all the locks are padded and aligned in the compiler-optimized version, these false sharing misses are eliminated. However, there are a large number of locks in Mp3d, and few of these locks exhibit any significant contention, particularly for small numbers of processors. Therefore the total amount of padding applied to the data structures in Mp3d was large, and the cost in spatial locality of the larger data set more than outweighed the reduction in false sharing misses. Consequently, the overall miss rate was an average of 33% higher relative to the the programmer-optimized version. However, the run-time impact of reduced spatial locality and false sharing scale differently with increasing number of processors. In fact, Section 6.2.2 will show that performance is impacted more severely by the latter than the former. Thus, the compiler-directed transformations constitute a better space-time tradeoff than those performed by the programmer.

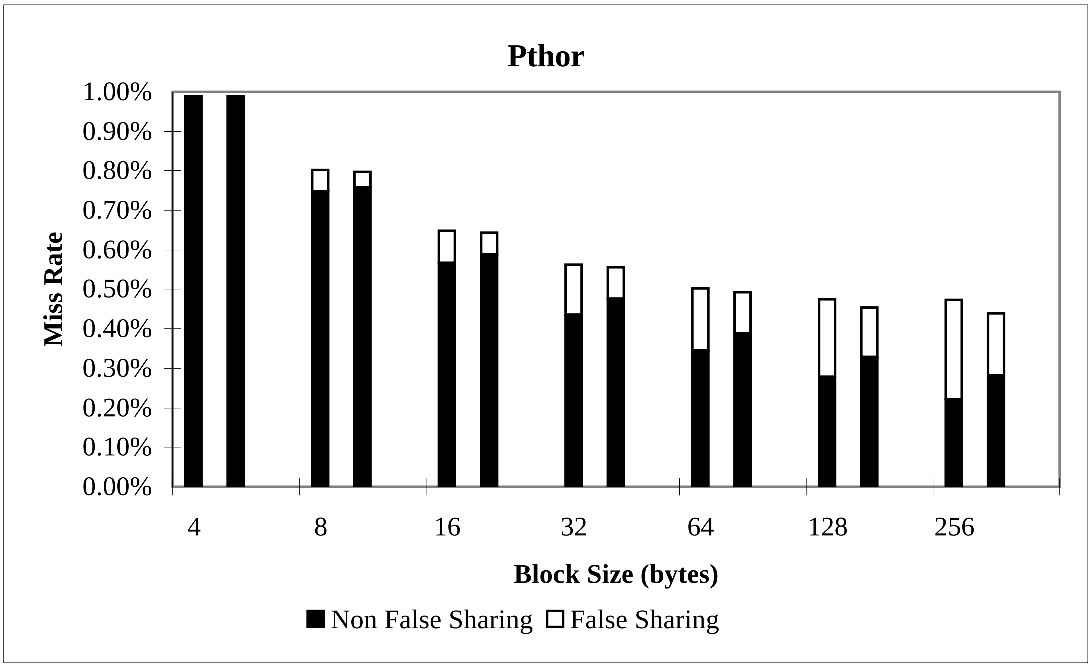


Figure 6.11: Total and false sharing miss rates for programmer and compiler-optimized versions of Pthor.

Pthor

In Pthor (Figure 6.11) the compiler analysis identified several data structures to be transformed that accounted for an average of 33% of all false sharing misses in the programmer-optimized version. Group & transpose was applied to two vectors that were accessed in a per-process manner. However, these vectors were not accessed very frequently, so the effect on the total number of false sharing misses was negligible. Pad & align was applied to lock variables, as well as four busy vectors that had high frequencies of writes with no spatial locality. The four vectors accounted for 26.1% of all false sharing misses, while the locks were responsible for 7.5%. However, due to the cost in spatial locality the overall cache miss rate of the compiler-optimized version was reduced by an average of 2.5% compared to the programmer-optimized version.

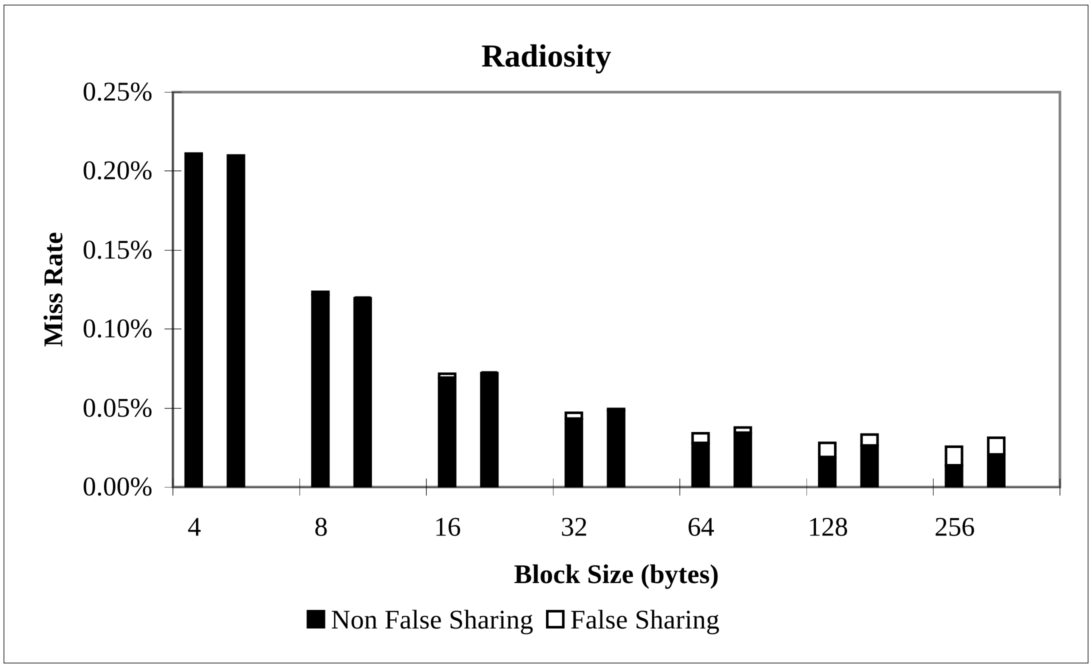


Figure 6.12: Total and false sharing miss rates for programmer and compiler-optimized versions of Radiosity.

Radiosity

The compiler and programmer-optimized versions of Radiosity (Figure 6.12) both have group & transpose applied to the same set of vectors. In addition, the compiler-optimized version also pads & aligns lock variables and a set of busy scalars. The differences in miss rates between the two versions reflect this. The false sharing miss rate of the compiler-optimized version is 53% lower than that of the programmer-optimized version. On the other hand, the additional padding worsened the spatial locality, making the overall miss rate 9% higher. However, since the miss rates for both versions are less than 0.05% for 32 byte cache blocks and larger, the absolute differences are minuscule. Moreover, as was the case for Mp3d, the performance impact of decreased spatial locality and false sharing scale differently with the number of processors. Section 6.2.2 will show that reducing the run-time impact of false sharing far outweighs any small decrease in spatial locality.

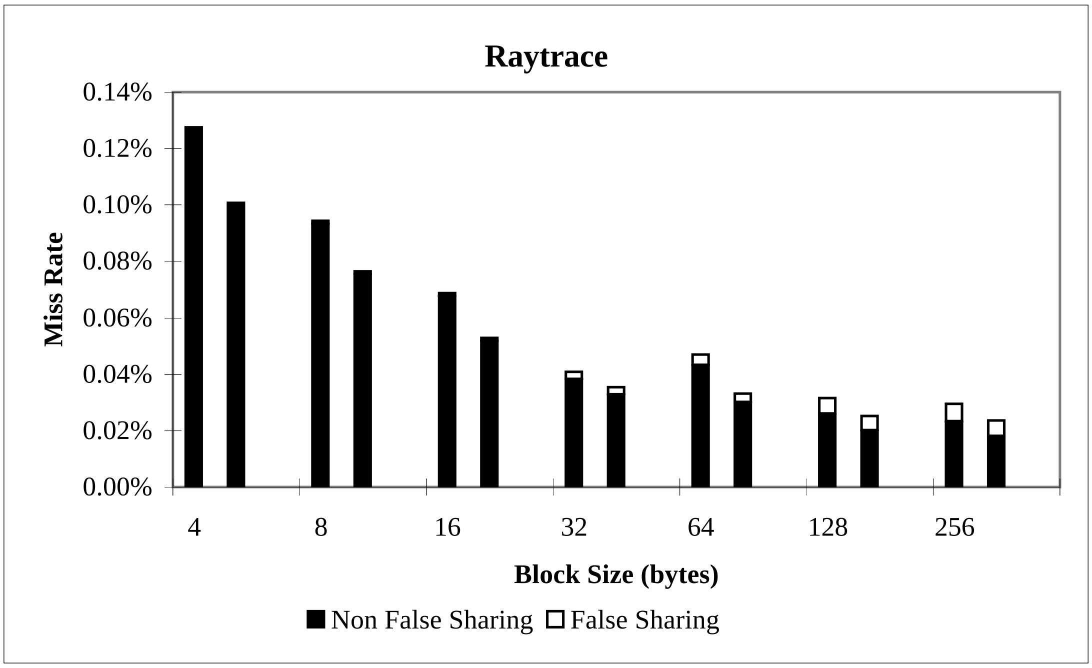


Figure 6.13: Total and false sharing miss rates for programmer and compiler-optimized versions of Raytrace.

Raytrace

The programmer and compiler-optimized versions of Raytrace (Figure 6.13) apply almost the same transformations. The differences are that the compiler-optimized version has pad & align applied to all lock variables, and the programmer-optimized version has applied group & transpose to one more vector, that is used in the management of the shared work pools, than the compiler-optimized version. The result is that the total and false sharing miss rates for the compiler-optimized version are on average 21% and 9% lower, respectively, than for the programmer-optimized version. However, as for Radiosity, the miss rates for both versions are very small, so that the absolute differences are almost insignificant.

Topopt

The difference between the compiler-optimized and programmer-optimized versions of Topopt (Figure 6.14) is that the programmer-optimized version has indirection applied to only two out of the four replicated, heap-allocated vectors that are trans-

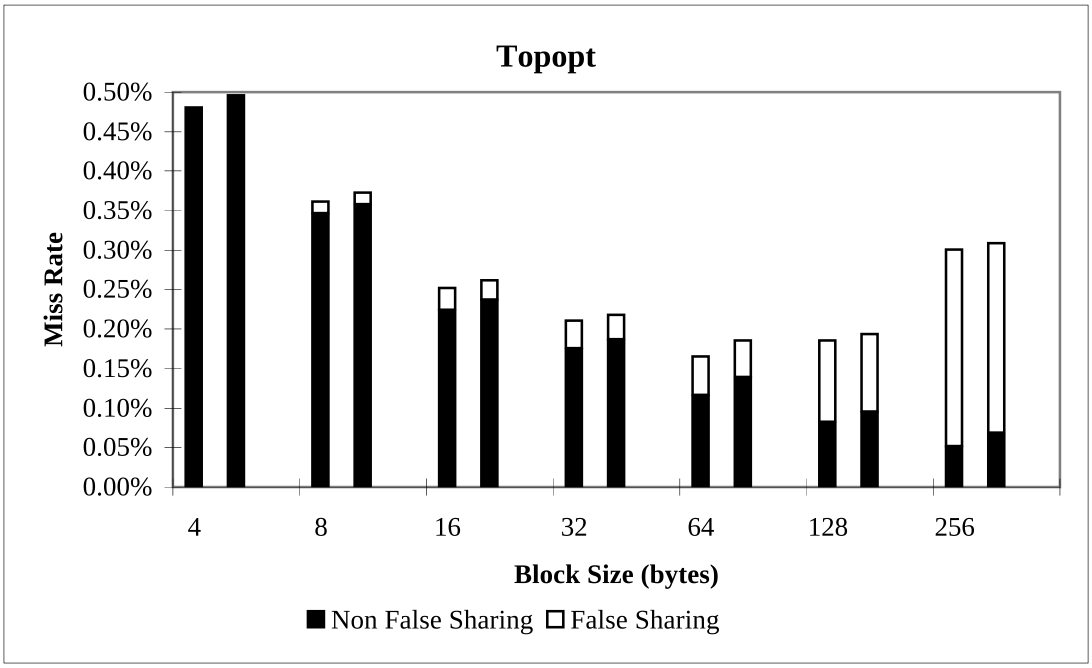


Figure 6.14: Total and false sharing miss rates for programmer and compiler-optimized versions of Topopt.

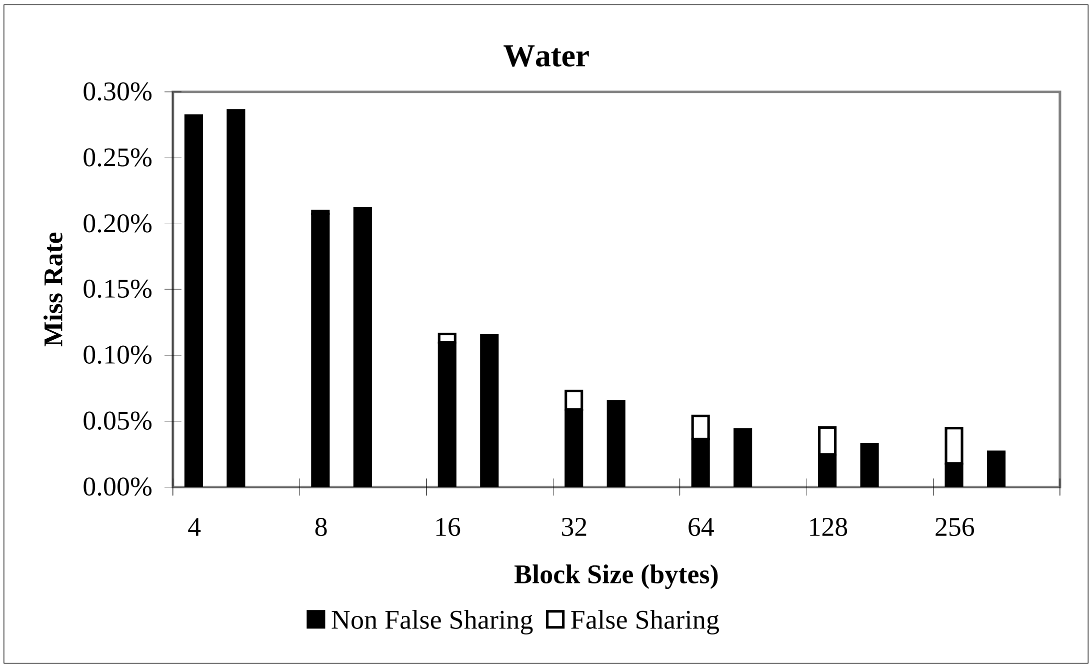


Figure 6.15: Total and false sharing miss rates for programmer and compiler-optimized versions of Water.

formed in the compiler-optimized version. The vectors that are not transformed are those that contain one element that is write-shared among the processors (see Section 6.1). The additional transformations result in the compiler-optimized version having a 6% lower false sharing miss rate than the programmer optimized version. However, due to the additional space requirements, the overall cache miss rate is 5% higher.

Water

All of the false sharing in Water (Figure 6.15) is due to lock variables. Therefore, all false sharing misses that occur in the programmer-optimized version are eliminated in the compiler-optimized version when the lock variables are padded and aligned. However, unlike Mp3d, this transformation barely affects spatial locality. This is reflected by the overall miss rate for the compiler-optimized version being 16% lower than it is for the programmer-optimized version.

6.2.2 Execution-Time Results

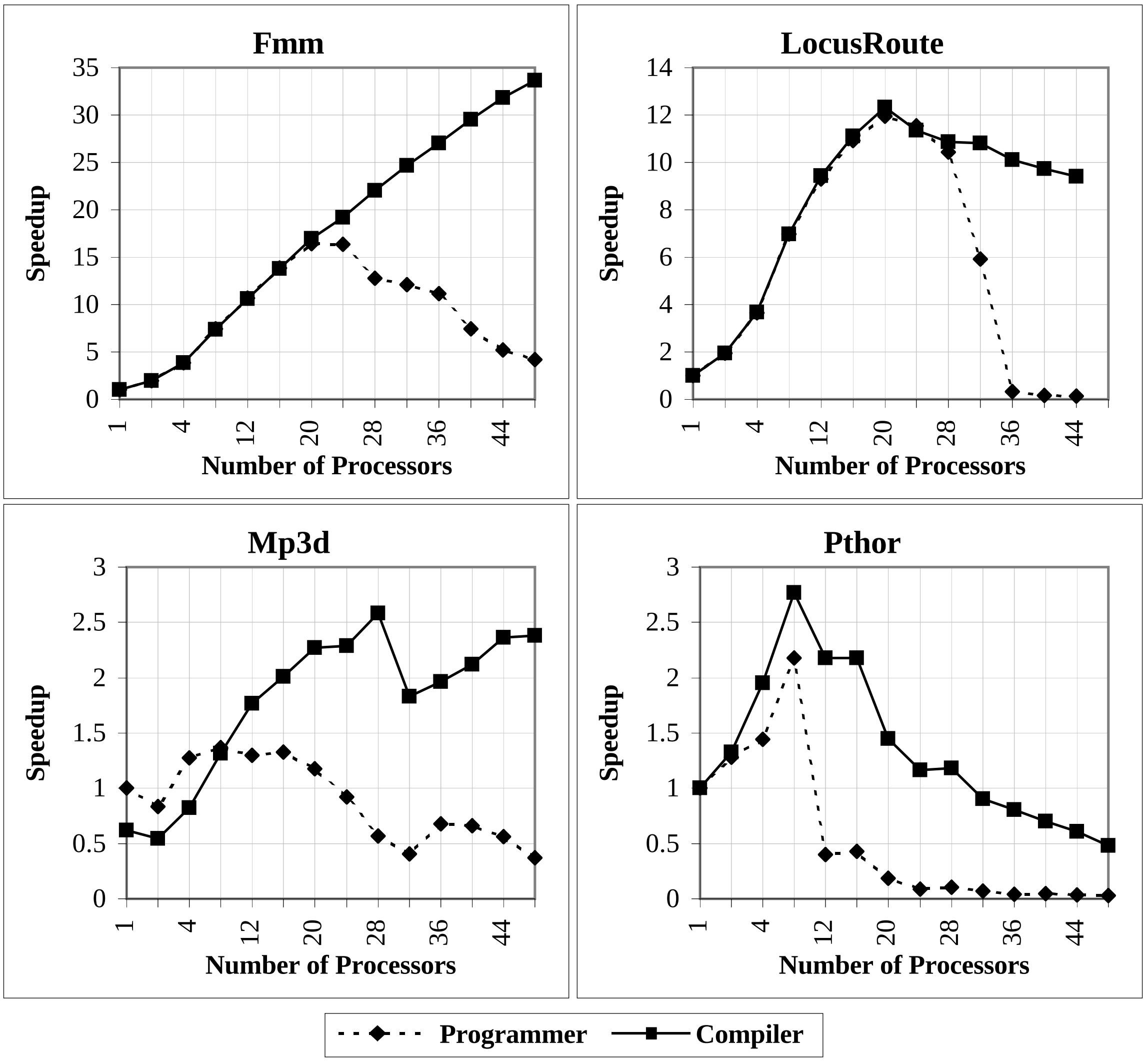


Figure 6.16: Speedup and scalability (speedup vs. number of processors) of programmer and compiler-optimized versions of the workload.

Figures 6.16 and 6.17 show the speedup[1] and scalability of the compiler-optimized and programmer-optimized versions of the workload as a function of the number of processors. The results are summarized in Table 6.3.

[1] Speedup is relative to the uniprocessor execution of the unoptimized version of the program for Fmm, Radiosity, Raytrace and Topopt, and the programmer-optimized version for LocusRoute, Mp3d and Water.

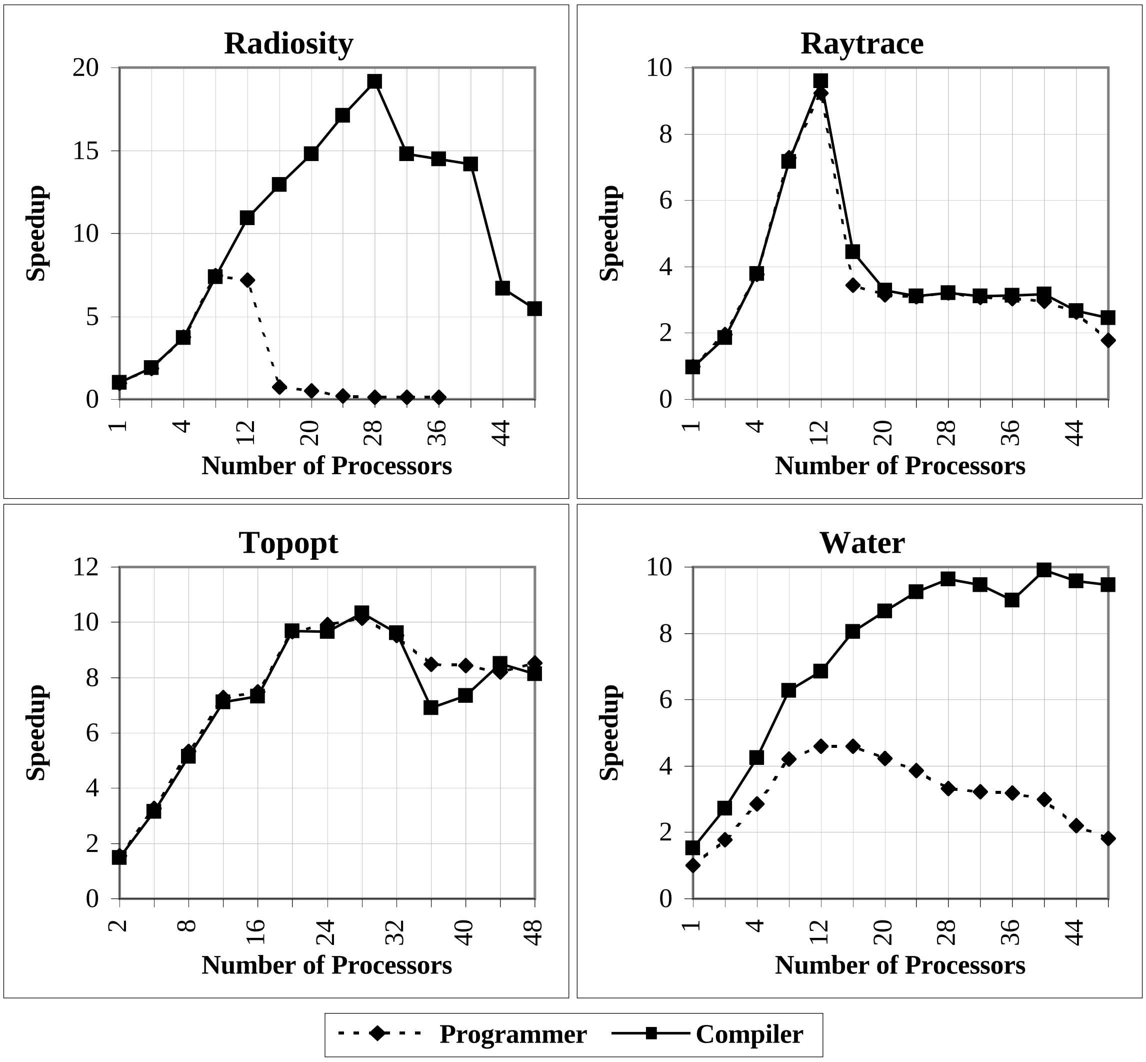


Figure 6.17: Speedup and scalability (speedup vs. number of processors) of programmer and compiler-optimized versions of the workload.

For some programs the difference in maximum speedup is barely discernible (LocusRoute, Raytrace and Topopt), while for others they differed by factors of 2 or more (Fmm, Radiosity and Water). However, in all cases the compiler-optimized version of the program performed better than the programmer-optimized version.

The programs fall roughly into two groups, depending on whether the speedup trend of the two versions are similar (LocusRoute, Pthor, Raytrace and Topopt), or whether they diverge (Fmm, Mp3d, Radiosity and Water). For the former the limitation on speedup is determined more by the size and parallelism afforded by the problem, than limitations brought on by false sharing. This is supported by the fact that any divergence in performance of the two versions occurs only after the maximum speedup has been achieved.

For the latter group the memory contention brought on by false sharing limits the performance of the programmer-optimized version of the programs. This is particularly clear for Water, where simulation showed that the transformations eliminated all false sharing misses. For Fmm, Mp3d and Radiosity, the results illustrate that different components of the cache miss rate as well as the false sharing miss rates to different data structures scale differently with increasing number of processors. Therefore, the cache performance for the programs at 12 processors, as determined by simulation, does not accurately reflect that for higher numbers of processors on a real machine.

The difference in scalability between different components of the miss rate, or between the false sharing miss rates to different data structures, can be explained by considering the impact of additional processors on the memory reference pattern. Ignoring conflict misses, the non-false sharing related miss rate is determined by the size of the problem (compulsory and capacity misses) and the degree of sharing between the processors (coherency misses). Since the transformations affect the former, but not the latter, only the effect of additional processors on the miss rate relating to the size of the problem is relevant to this discussion.

At a high level, parallelism in parallel programs is usually afforded by either distributing the data set or the set of operations performed on each element of the data set, across the processors. In either case the number of accesses to the data set per processor drops as more processors are applied. Therefore, the number of non-false sharing misses per processor that are related to the size of the data structure, does not increase as the number of processors grows, but in many cases decreases. This is not the case for false sharing misses. As the number of processors grows, so does the contention for each falsely shared cache block, and subsequently the number of false sharing misses incurred by each processor.

How the false sharing miss rates to different data structures scale with the number of processors depends on how each falsely shared cache block is accessed. On the one hand, if a vector is accessed in a per-process manner, the number of processors that

Table 6.3: Maximum speedups for programmer and compiler-optimized versions and the number of processors at which they occur.

Program	Maximum Speedup and Scalability (# of processors)			
	Programmer		Compiler	
Fmm	16.4	(20)	33.6	(+48)
LocusRoute	11.9	(20)	12.3	(20)
Mp3d	1.4	(4)	2.6	(28)
Pthor	2.2	(8)	2.8	(8)
Radiosity	7.4	(8)	19.2	(28)
Raytrace	9.2	(12)	9.6	(12)
Topopt	10.2	(28)	10.3	(28)
Water	4.6	(12)	9.9	(40)

can falsely share each cache block is bounded above by the number of vector elements per cache block. Thus the number of false sharing misses to these data structures tends to level off once that number of processors has been reached. On the other hand, all processors often access the same set of locks. This means that the number of false sharing misses incurred by the (unrelated) data structures that are allocated with these locks do not level off at some number of processors, but continues to increase. Furthermore, locks tend to guard critical sections used to serialize accesses to shared data. As the number of processors grows (while the input size is kept constant), the ratio of time spent in the (sequential) critical sections increases. If the cache blocks that contain the locks used to guard these critical sections are falsely shared, the additional cache misses they incur prolong the critical sections, increasing the sequential portion of the programs thereby reducing the maximum speedup achievable.

Fmm illustrates the difference in scalability of false sharing miss rates to different data structures well. While the miss rates for both the programmer and compiler-optimized versions were almost identical for 12 processors, locks were only padded in the compiler-optimized version. The run-time performance of both versions were similar up to 20 processors. When the number of processors increased beyond that, their speedup curves diverged. The additional overhead of false sharing of the cache blocks containing locks became a bottleneck in the programmer-optimized version;

but no performance bottleneck was evident for the compiler-optimized version for the range of processors used in the experiments.

The impact of additional processors on the tradeoff between spatial locality and false sharing misses is reflected in the results for Mp3d and Radiosity. Simulation showed that for 12 processors, the compiler-optimized version of these programs exhibited fewer false sharing misses (66% and 53% respectively), but higher overall miss rates (33% and 9% higher respectively) than the programmer-optimized version. Still, as the number of processors used increased, the compiler-optimized version outperformed the programmer-optimized version by almost a factor of two for Mp3d and more for Radiosity.

6.3 Putting it All Together

The performance of the four programs (Fmm, Radiosity, Raytrace and Topopt) for which the unoptimized, programmer-optimized and compiler-optimized versions all were available, is depicted in Figure 6.18. The programs fall into two categories, depending on whether the performance of the programmer-optimized version shadows that of the unoptimized (Fmm and Radiosity) or the compiler-optimized version (Raytrace and Topopt) of the program. There is no middle ground.

Even though, as shown in the previous section, both programmer efforts and compiler-directed transformations are effective in eliminating false sharing misses, these results confirm that some false sharing misses are more crucial to performance than others. In fact, the results for Fmm and Radiosity show that despite significant programmer hand-tuning, little was gained over the unoptimized version. On the other hand, for Raytrace and Topopt, the differences in the miss rates were not reflected in the performance measurements. In other terms, eliminating false sharing from some data structures matters little for overall program performance if those that constitute the main bottleneck remain unchanged.

This highlights the difference between the programmer efforts and the compiler-driven approach. The compiler was more successful in eliminating false sharing misses than the programmer, but more importantly, detected and eliminated those that constituted the performance bottlenecks more often than the programmer did.

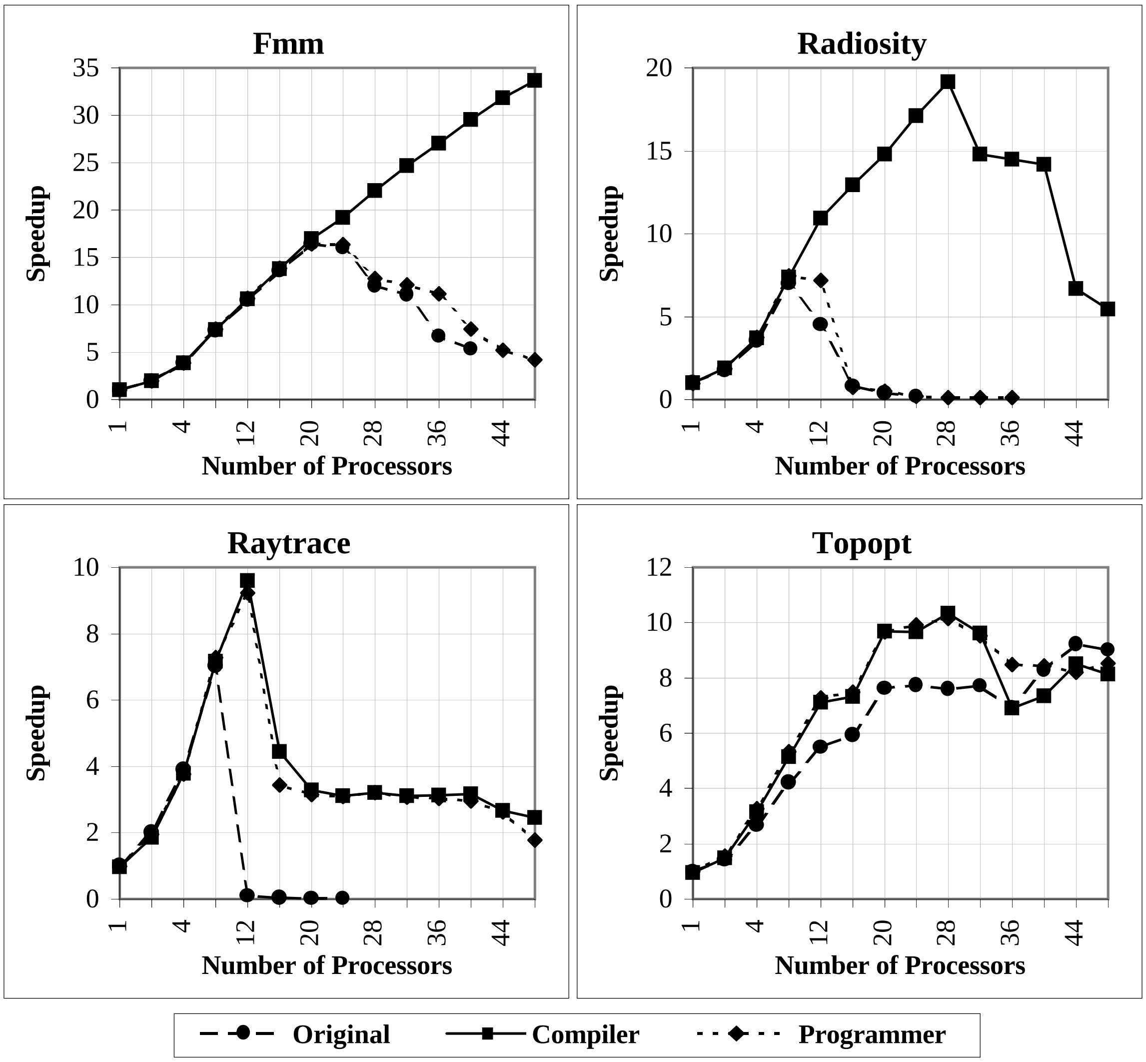


Figure 6.18: Speedup and scalability (speedup vs. number of processors) of unoptimized, programmer and compiler-optimized versions of the workload.

Chapter 7
RELATED WORK

Related work falls into two broad categories. The first is work that addresses the main thrust of this dissertation, namely that aimed at reducing false sharing and its performance impact. The second is work that analyzes the synchronization pattern of parallel programs. This chapter will describe both in turn. In addition, it will survey some work that has used the research detailed in this dissertation.

7.1 Reducing False Sharing

This section describes related studies that reduce false sharing, either in cache blocks or in pages, and its performance impact. These studies can be divided into two groups according to their main course of action. The first changes a program's shared memory reference pattern to better fit the cache architecture and coherence protocol. The other augments the cache architecture and/or the coherence protocol to reduce or eliminate any detrimental effects of false sharing.

7.1.1 Reducing False Sharing Using Program Transformations

A number of studies have used program transformations to reduce false sharing. However, they all differ from this dissertation in one or more important respects. First, they apply a different set of transformations. Some use only padding, while others transform the control structure of the program, e.g., loop transformations, instead of changing the shared data layout. Second, some apply their transformations manually, either based on programmer knowledge or profiles derived from simulation, rather than using a compiler-directed approach. Third, the workload in several studies only consists of individual loop nests; those that use complete programs use either fewer or smaller programs, or programs from a different parallel programming model than the workload in this dissertation. Lastly, several studies concentrate on false sharing in operating system pages, while this dissertation focuses on false sharing in cache blocks.

The simplest technique to eliminate false sharing is to manually apply padding to prevent the allocation of unrelated data objects to the same cache block. This technique was used in two studies of false sharing in operating system pages: Bolsoky et al. [BFS89] and Priol and Lahjomri [PL92].

Bolosky et al. [BFS89] studied techniques for automatic page placement on NUMA architectures. They reduced false sharing by manually applying padding around susceptible objects, based on their knowledge of their benchmark applications. No results attesting to the amount or change in false sharing were reported. However, they suggested that a compiler-driven approach be developed to reduce the necessity for programmer intervention.

Priol and Lahjomri [PL92] evaluated the performance of a shared virtual memory system running on an distributed memory multiprocessor. Studying one algorithm that produces an ortho-normal basis from a set of independent vectors, they found that false sharing in pages was an obstacle to program speedup. They successfully eliminated the false sharing by manually padding vectors to align them on page boundaries and selectively using broadcasts of page updates.

Several studies have focused on reducing false sharing in loops in medium-grained parallel programs, either by changing array data layout or by changing the control structure of loop nests. Peir and Cytron [PC89] partitioned loops to minimize interprocessor communication when processing recurrences. Their mechanism for partitioning utilizes loop unrolling and dependence vectors. Partitions are then scheduled on different processors.

Gupta and Padua [GP91] examined the cache performance of sequential programs that had been automatically parallelized at the loop level. They reduced false sharing by strip-mining the loops to the size of the cache block (in words). This ensured that sets of iterations that access array elements that reside in a single cache block are scheduled as one chunk, making it much less likely that independent iterations of the parallel outer loop would falsely share cache blocks. The decline in miss rates ranged from 4% to almost 60%, as the cache block size was increased to 128 bytes. No execution times were reported.

Lilja [Lil94], as an extension of [GP91], studied the interaction of parallel loop scheduling and prefetching with respect to memory subsystem performance. A new "adaptive" prefetching strategy was proposed. Under adaptive prefetching, the number of words fetched into the cache on each miss is a function of the number of iterations (size of work) scheduled on that processor. This reduced both false sharing as well as the number of unnecessary word fetches. Using a unit word size cache block, adaptive prefetching was compared to four fixed size (1, 2, 4, and 8 words) prefetching strategies, each combined with either fixed chunk scheduling or guided self scheduling. The workload consisted of six loop parallel numeric benchmarks. The results showed that the combination of adaptive prefetching and guided self scheduling reduced network traffic, cache misses and the number of invalidations more than any other combination.

Ju and Dietz [JD91] constructed an interference graph to model the constraints on loop structures and layouts of arrays in memory. Each vertex in the graph represens either an individual loop nest or an array. There is an edge between a loop vertex and an array vertex if and only if the array was accessed in that loop. Each edge is labeled by a cost table, which records the expected coherency cost for each possible combination of loop nest structure and data layout for that edge. Given this graph, a search is performed to to find loop nest structures and data layouts that minimize the sum of the edge cost functions, i.e., the overall coherence cost of the program. For a small program consisting of two loop nests, the results showed that their technique successfully reduced execution time by as much as 80% when running on 8 processors.

Granston [Gra93] developed a loop transformation theory to identify and eliminate page level false sharing in shared virtual memory systems. The theory is based on the analysis of the iteration spaces of parallel loops. The transformations select blocking and array alignment factors to minimize overlap between sets of pages accessed by different processors. However, as it has not been implemented, no results are available.

Two studies do not mention false sharing directly per se, but nevertheless reduce it by improving processor locality: Wolf and Lam [WL91] analyzed individual loop nests to improve locality by transforming the code using loop interchange, reversal, skewing and tiling. They developed a mathematical formulation of reuse and locality in terms of array indices that were affine functions of the loop indices. Based on their mathematical framework they presented an algorithm that searched the space of possible loop transformations for each loop nest to determine the loop structure that maximized the reuse in the inner loop. They presented preliminary results for two programs: LU-decomposition and SOR. While the non-tiled versions of these programs exhibited speedups of 2 or less on an eight processor machine, the transformed versions achieved near linear speedup.

Kennedy and McKinley [KM92] explored the tradeoffs between effectively utilizing parallelism and improving inner loop locality for scientific Fortran codes. They used a smaller set of loop transformations, and potentially less accurate analysis than [WL91]. However, the simplicity of the analysis exhibited better time complexity ($O(n^2)$ vs. exponential in loop nest depth). Instead of searching the space of possible loop permutations, they used an algorithm that directly computed the preferred loop nest structure.

A different approach altogether was taken by Bianchini and LeBlanc [BL92]. They studied the effectiveness of using software caching techniques to eliminate false sharing on cache coherent multiprocessors. Their workload consisted of one program, parallel shellsort, an algorithm with a dynamically changing cross-processor memory reference pattern. Their application level coherency scheme consisted of changing the algorithm in two ways. First, each process copied the the elements it had to sort into a local buffer, performed an insertion sort on these elements, and then copied them back into the shared data structure. Second, the initial distribution of data was changed so that block transfers could be used during the initial phase of the algorithm. This approach was then compared to a naive implementation and two that reduce false sharing: indirection and padding. Due to the loss in spatial locality, padding had little benefit over the naive implementation. Indirection, however, was much closer to the performance of software caching, particularly as the number of processors increased. There were two reasons why software caching yielded better performance. First, since the data distribution changed dynamically, indirection only eliminated false sharing in the first phase of the algorithm. Second, but less important, every access to shared data using indirection suffered the overhead of an additional memory reference. All optimizations depended on a high degree of programmer involvement and the use of application and architecture specific knowledge.

The most closely related research, both in terms of type of workload and the selection of transformations, is that of Torrellas et al. [TLH94]. They reduced false sharing in a set of coarse-grained parallel programs by applying a set of five transformations manually. Their transformations are: "split scalar", "heap allocate", "expand record", "align record" and "lock scalar". Split scalar and expand record are special cases of pad & align applied to busy write-shared scalars and array elements of record types, respectively. Heap allocate allocates data from process specific heap regions, instead of one global heap, on the assumption that data that is allocated by a slave process is most likely to be accessed by that process. Align record lays out an array of records at a distance from a cache block boundary that minimizes the number of cache blocks that the average record spans. As such it only applies to records that exceed the cache block size. Lastly, lock scalar allocates active scalars that are protected by a lock together with that lock so that the scalar is prefetched whenever the lock is acquired. Except for lock scalar, these transformations change the data layout by padding and/or changing the alignment only. No variations of group & transpose or indirection were used, i.e., transformations that group together data that are predominantly accessed by the same process. This dissertation found these transformations to be the most effective in reducing both the false sharing and overall cache miss rates.

To determine when data structures suffered from false sharing and to guide the application of the transformations, Torrellas et al. used detailed, trace driven simulation profiles, instead of static analysis. 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, the transformations described in this dissertation reduced the *total* miss rate by an average of 49% (on the unoptimized programs, also for 64 byte cache blocks, but with 12 processors).

7.1.2 Hardware Solutions to Reduce False Sharing

Dubnicki and LeBlanc [DL92] suggested an adjustable block size cache architecture to reduce false sharing, while taking advantage of spatial locality when present. Their approach used a power-of-two buddy scheme to split and merge the data in cache blocks based on their recent reference history[1]. The data in cache blocks are merged and split upon ownership changes based on the state information of the cache blocks. In principle, the data of two adjacent cache blocks are merged when the state shows that the data in both cache blocks have been accessed recently by the owning pro-

[1] Note, that the number of blocks in the cache is not changed, only the amount of data stored in each cache block.

cessor. Conversely, the data in a cache block is split when only one half of the cache block has been accessed by the owner, i.e., the data in the cache block is falsely shared. To implement this scheme three additional pieces of state information have to be associated with each cache block: (1) a field that indicates the size of the data held in the cache block; (2) reference bits for each half of the cache block; and (3) a counter used to indicate the recent reference history of the cache block. In addition the address tag field must be large enough to hold the tag for the smallest possible cache block. A fully associative cache is assumed, or alternately, the presence of multiple, direct-mapped sub-caches, each with a different line size. Their results showed that their technique was successful in reducing both the memory waiting time, as well as the number of words transferred compared to any fixed block size cache for all but three of the programs in their workload. However, they did not consider the hardware costs of providing a large second level fully associative cache, or multiple direct mapped caches. Furthermore, their results were obtained using simulation of infinite caches; therefore the cache wasting effect of storing small pieces of data in large cache blocks could not be assessed.

To reduce the number of false sharing misses, while still taking advantage of the prefetching benefit inherent in larger cache blocks, Chen and Dubois [CD93] suggested the use of a partial block invalidation cache coherency protocol. In this protocol each cache block (address block) is subdivided into a number of invalidation blocks. While the whole address block is fetched on a read miss, only the affected invalidation block is invalidated on a write. This exploits the prefetching of the larger address block while reducing the false sharing read miss rate to that occurring within the invalidation blocks. However, ownership is still maintained on a address block basis. Therefore, false sharing can occur on writes when different processors write to different invalidation blocks within the same address block, since each such write requires an ownership change. Their results reflected this. The application with the smallest ratio of reads to writes (2.9) saw the smallest improvement in the false sharing miss rate. The other two applications (read to write rations of 5 and 75, respectively) reduced the false sharing miss rate to a small fraction of what it would have been had the invalidation block been the same as the address block.

Dubois et al. [DSR+93] evaluated two hardware assisted techniques for reducing false sharing misses. The first delays the effect of invalidations by buffering them at the sender, receiver, or both, until special *acquire* or *release* instructions are executed. Because of the delayed invalidations these techniques are only applicable to applications that conform to the release consistency model [GLL^{+}90]. The second is a partial block invalidation protocol with word size invalidation blocks. Their results showed that the performance of delaying invalidations at either the sender or the receiver (but not both) was dependent on the memory reference pattern of the application. However, buffering invalidations at both the sender and the receiver, and the word invalidate coherency protocol performed consistently well across their target workload. The former reduced the false sharing miss rate by 85% to 100%; the latter totally eliminated false sharing misses. 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 a valid bit per word in the cache block, causes more invalidations when the writes exhibit spatial locality, and suffers from the same ownership contention as [CD93].

7.2 Program Synchronization Analysis

Some work has been done on analyzing synchronization in parallel programs to detect race conditions. The analysis mainly focuses on event variable synchronization, i.e., `post` and `wait` or locks, rather than barrier synchronization; however, few show experimental results attesting to the effectiveness of the algorithms.

Taylor [Tay83] presented an algorithm that analyzed rendezvous in concurrent Ada programs. The algorithm computed the set of possible concurrency states for a program and determined the possible sequences in which they may occur. A concurrency state is an n-tuple of synchronization statements, where n is the number of tasks in the program. Each element of the n-tuple represents a synchronization statement that may be executed next by the corresponding task. As the number of synchronization states for a program is exponential in the number of tasks, so was the algorithm.

McDowell [McD89] and Helmbold and McDowell [HM91] also enumerated possible concurrency states in the presence of event variable synchronization and locking. They reduced the number of states either by exploiting similarity between tasks to generate fewer states or by folding multiple states into a single state. However, the worst case number of states remained superpolynomial.

Callahan, Kennedy and Subhlok [CKS90] introduced a *synchronized control flow graph* that combined control flow and event synchronization information. The synchronized control flow graph can be used to determine when data dependencies are preserved in parallel programs and when possible race conditions exist. Their model of parallel programming was based on parallel case and parallel do constructs, and their analysis was focused on loop nests. In contrast, the non-concurrency analysis presented in this dissertation is based on analyzing barrier synchronization in programs with process level parallelism. Moreover, the analysis is performed over the entire program.

Masticola and Ryder [MR93] presented a framework for non-concurrency analysis of Ada tasks. They iteratively used four different refinement strategies to produce conservative approximations of sets of basic blocks that could not possibly execute concurrently. The barrier synchronization analysis algorithm bears some similarity to their strategy of *pinning*. Pinning analysis determines sets of basic blocks that *must* execute concurrently based on synchronization events in the program. For instance, if two tasks T_1 and T_2 synchronize using statements S_1 and S_2 respectively, the set of statements following S_1 up to the next synchronization event in T_1, must execute concurrently with the statements following S_2 up to the next synchronization event in T_2. However, there are some key differences between pinning and the non-concurrency analysis described in this dissertation. First, pinning is aimed at pairwise process synchronization, as opposed to global process synchronization. Second, pinning is used to compute *must* information for a subset of the basic blocks, while the barrier analysis algorithm computes *may* information for the entire program. Third, the solution given by the barrier analysis algorithm is applicable to programs written under a different model of parallel programming. Finally, the barrier analysis algorithm is simpler and therefore more efficient.

7.3 Applications

The research described in this dissertation has already been applied in two studies [TE93, TE94].

Tullsen and Eggers [TE93] studied the effectiveness of prefetching on bus-based multiprocessors. They found that the limit to effective prefetching (and performance in general) is invalidation misses on shared data, something that traditional (uniprocessor-based) prefetching algorithms do not address. However, using the techniques presented in this dissertation they were able to eliminate false sharing, a significant source of invalidation misses, in their workload to the point where the performance of the (simpler) traditional prefetching algorithms equaled that of more complex, multiprocessor-based algorithms.

Thekkath and Eggers [TE94] studied placement of threads based on inter-thread sharing on a parallel architecture in which processors have multiple hardware contexts [ALKK90]. They found that for programs that had been optimized for locality, either manually or using the compiler-directed approach presented in this dissertation, the coherency-traffic was low enough compared to the number of shared-memory references that load balancing was the dominant issue.

Chapter 8

SUMMARY AND CONCLUSIONS

This dissertation has investigated the use of static analysis and data transformations to improve performance of parallel programs by eliminating false sharing on shared-memory multiprocessors. A series of compile-time algorithms that analyze coarse-grained, explicitly parallel programs were developed and implemented. The algorithms pinpoint which data structures are likely to incur false sharing misses, where locality may be improved, and which transformations to apply. Guided by a set of heuristics, three data transformations were used to group together data that were accessed by the same processor and separate data accessed by multiple processors.

Two sets of programs were used to evaluate the effectiveness of the transformations in eliminating false sharing misses and how they affected run time performance. The first set were programs where either little or no programmer effort had been expended on cache-conscious programming, or where obvious programmer transformations had been undone. These programs exhibited significant amounts of false sharing, so much that for the larger block sizes, misses due to false sharing dominated the overall miss rate. They were included to illustrate both the ability of the compiler to detect and eliminate false sharing misses as well as the performance improvements possible. The second set of eight applications were programs that had undergone extensive hand-tuning to optimize for memory hierarchy performance. They exhibited significantly less false sharing than the first set of programs. However, even in these programs, false sharing misses accounted for at least 40% of all cache misses for the largest block sizes. This set of programs were included to see if the compiler-analysis could improve upon hand-optimized programs.

Results showed that the compile-time analysis accurately pinpointed data structures that suffered from false sharing and that the transformations effectively eliminated the false sharing misses. Across the unoptimized workload, the average reduction (over multiword cache block sizes) in the false sharing miss rate was 81%. Even though two of the transformations applied decrease spatial locality, the overall miss rate for these programs was reduced by an average of 42%. No single transformation was responsible for the false sharing reductions, even within a single program; all were important contributors.

The reduction in false sharing had two effects on run-time performance as measured on a KSR 2: reductions in execution time and improved program scalability. Of the two, improved scalability was the decisive factor. The compiler-directed transformations eliminated false sharing as a bottleneck, and extended scalability, often to the point where the maximum achievable speedup more than doubled. This illustrates well that false sharing can be a significant hindrance to obtaining good performance of parallel programs on cache coherent, shared-memory multiprocessors; and that compile-time shared data transformations can radically improve program performance.

Even though the amount of false sharing in the programmer-optimized workload was smaller than in the unoptimized workload, the compiler improved upon the false sharing miss rate of these programs, some substantially. The execution time results showed that the compiler-optimized versions of the programs always performed better than the programmer-optimized. For some the difference was marginal; for others the maximum achievable speedup more than doubled. In addition, when the compiler and programmer restructured different data structures, the compiler made a better choice, selecting data structures that were more of a bottleneck to scalable performance. Therefore, even though simulation showed the cache miss rates for the two versions to be similar, the compiler-optimized version outperformed the programmer-optimized version when additional processors where used.

With the trend toward larger caches, larger coherency units, and longer memory latencies, false sharing will have an increasingly large, negative performance effect. Regaining the performance will necessitate either a significant programming effort to improve locality or the use of a compile-time system like the one described in this dissertation. The latter is preferable on three grounds. The first is performance. For every program in the workload, the static analysis was no less successful than programmer optimization in eliminating false sharing, and significantly better for some. Moreover, the compiler-directed transformations led to greater performance improvements than the programmer efforts. The second is program portability. The programmer-optimized programs must necessarily be customized to the particular cache architecture at the source code level. The compiler-directed approach has no such drawback, as the architectural dependencies are hidden in the compiler. 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 program. For a nominal cost (under 10%, assuming sufficient physical memory) in compile time, the compiler-directed approach realized all three benefits.

While the algorithms presented in this dissertation were developed to eliminate false sharing, their ability to divide a program up into phases and approximate both the per-process memory reference pattern within each phase and how it shifts across phases has several potential uses.

First, they can be applied to more general memory system allocation decisions, i.e., whether to allocate shared data in local or global memories, to caches or local memories and whether to cache it or not. For instance, in NUMA architectures, data that is accessed predominantly by one processor should be allocated in a memory local to that processor. If the reference pattern shifts, good performance can be achieved by migrating the data closer to the processor that will be accessing it.

Second, for systems with multiple cache coherency protocols, the algorithms can choose the coherency protocol that best coincides with the sharing pattern for particular data structures. If the protocol can be set on a page basis, data structures that exhibit the same sharing pattern can be allocated to the same page and the coherency protocol customized for that page. If the cross-processor memory reference pattern shifts from one program phase to another, a different coherency protocol could be used after the shift. Pages could also be marked as non-coherent, i.e., no hardware coherency protocol, and the compiler could insert coherency instructions where necessary.

Third, knowledge of a program's cross-process memory reference and synchronization pattern can facilitate allocation of threads on multi-threaded multiprocessors based on the level of sharing between sets of processes or determine that sharing is uniform enough that sharing-based placement in unwarranted.

Last, the algorithms can improve prefetching on multiprocessors in two ways. First, by eliminating false sharing misses existing prefetching algorithms become more effective as the lifetimes of the cache lines become more predictable. Second, improved knowledge of the cross-processor sharing pattern can allow prefetching algorithms to make better decision as to which data to prefetch. For instance, data that is actively contended for and write-shared by multiple processors are likely to have short lifetimes in the cache. Therefore, little benefit would be expected by prefetching them. However, data that is write-shared across phases exhibit migratory sharing, and are much better candidates for prefetching.

Bibliography

[AB86] J. Archibald and J-L. Baer. An evaluation of cache coherence solutions in shared-bus multiprocessors. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[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.

[ALKK90] A. Agarwal, B-H. Lim, D. Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[ASU88] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 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.

[BCK+89] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, C. Fox, P. Messina, D. Walker, C. Ilsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD Report No. 827, Center for Supercomputing Research and Development, Urbana, Illinois, May 1989.

[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.

[BL92] R. Bianchini and T.J. LeBlanc. Software caching on cache-coherent multiprocessors. In *Proceedings fo the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 521–526, June 1992.

[BP89] C.J. Beckmann and C.D. Polychronopoulos. The effect of barrier syncrhonization and scheduling overhead on parallel loops. In *International Conference on Parallel Processing*, volume II, pages 200–204, August 1989.

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

[CD93] Y.S. Chen and M. Dubois. Cache protocols with partial block invalidations. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 16–23, April 1993.

[CGBG88] D.R. Cheriton, A. Gupta, P.D. Boyle, and H.A. Goosen. The vmp multiprocessor: Initial experience, refinements and performance evaluation. In *15th Annual International Symposium on Computer Architecture*, pages 410–421, May 1988.

[CK88] 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.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, Texas, January 1989.

[CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of Event Synchronization in a Parallel Programming Tool. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 21–30, March 1990.

[CM81] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[DL92] C. Dubnicki and T.J. LeBlanc. Adjustable block size coherent caches. In *19th Annual International Symposium on Computer Architecture*, pages 170–180, May 1992.

[DN87] S. Devadas and A.R. Newton. Topological optimization of multiple level array logic. In *IEEE Transactions on Computer-Aided Design*, pages 915–941, 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.

[Egg89] S.J. Eggers. Simulation analysis of data sharing in shared memory multiprocessors. Technical Report No. UCB/CSD 89/501 (Ph.D. thesis), University of California, Berkeley, March 1989.

[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.

[EK89a] S.J. Eggers and R.H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.

[EK89b] S.J. Eggers and R.H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *16th Annual International Symposium on Computer Architecture*, pages 2–15, May 1989.

[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.

[FCS+93] J-M. Frailong, M. Cekleov, P. Sindhu, J. Gastinel, M. Splain, J. Price, and A. Singhal. The next-generation SPARC multiprocessing system architecture. In *IEEE COMPCON*, pages 475–480, February 1993.

[Fuj83] R.M. Fujimoto. SIMON: a simulator of multicomputer networks. Technical Report UCB/Computer Science Dpt. 83/140, University of California, Berkely, September 1983.

[Gib88] G. Gibson. SpurBus specification. Technical report, University of California, Berkely, December 1988.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 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.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs.* North-Holland Pub. Co., New York, 1977.

[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.

[HM91] D.P. Helmbold and C.E. McDowell. Computing reachable states of parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 76–84, May 1991.

[HN90] L.J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *ACM Transactions on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[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.

[KEW+85] R. Katz, S. Eggers, D. Wood, C.L. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[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.

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

[KU76] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *JACM*, 23(1):159–171, January 1976.

[LH88] J.R. Larus and P.N. Hilfinger. Restructuring lisp programs for concurrent execution. *Parallel Programming: Experience with Applications, Languages and Systems*, 23(9):100–110, September 1988.

[Lil94] D.J. Lilja. The impact of parallel loop scheduling strategies on prefetching in a shared memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):573–584, June 1994.

[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.

[LRZ93] W. Landi, B.G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[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.

[LYL87] R. L. Lee, P. C. Yew, and D. H. Lawrie. Multiprocessor cache design considerations. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 253–262, June 1987.

[McD89] C.E. McDowell. A Practical Algorithm for Static Analysis of Parallel Programs. *Journal of Parallel and Distributed Computing*, 6:515–536, June 1989.

[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.

[Mye81a] E. Myers. A precise inter-procedural data flow algorithm. In *Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[Mye81b] Eugene Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eight ACM Symposium on Principles of Programming Languages*, pages 219–230, Williamsburg, VA, 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.

[PL92] T. Priol and Z. Lahjomri. Experiments with shared virtual memory on an iPSC/2 hypercube. In *International Conference on Parallel Processing*, volume 2, pages 145–148, August 1992.

[RM88] C. Ruggieri and T.P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[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.

[Tay83] R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, 1983.

[TE93] D.M. Tullsen and S.J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *20th Annual International Symposium on Computer Architecture*, pages 278–288, June 1993.

[TE94] R. Thekkath and S.J. Eggers. Impact of sharing-based thread placement on multithreaded arhictectures. In *20th Annual International Symposium on Computer Architecture*, pages 176–186, April 1994.

[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.

[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.

[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.

[Wei80] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[WG89] W.D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[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.

Born outside of Oslo, Norway in 1966, Tor Erik Jeremiassen received his Bachelor of Science in Computer Science with High Honors and Special Honors in Computer Science from the University of Texas in May 1989. He received his Master of Science in Computer Science in December 1992, and his Doctor of Philosophy in Computer Science in June 1995. Both from the University of Washington.