Characterizing the Memory Behavior of Compiler-Parallelized Applications

Evan Torrie
torrie@cs.stanford.edu
Computer Systems Lab.
Stanford University
Stanford, CA 94305-4070

Margaret Martonosi
mrm@ee.princeton.edu
Dept. of Electrical Eng.
Princeton University
Princeton, NJ 08544-5263

Chau-Wen Tseng
tseng@cs.umd.edu
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Mary W. Hall
mhall@isi.edu
USC Information Sciences Inst.
4676 Admiralty Way
Marina del Rey, CA 90292

Abstract

Compiler-parallelized applications are increasing in importance as moderate-scale multiprocessors become common. This paper evaluates how features of advanced memory systems (*e.g.*, longer cache lines) impact memory system behavior for applications amenable to compiler parallelization. Using *full-sized* input data sets and applications taken from standard benchmark suites, we measure statistics such as speedups, synchronization and load imbalance, causes of cache misses, cache line utilization, data traffic and memory costs.

This exploration allows us to draw several conclusions. First, we find that larger granularity parallelism often correlates with good memory system behavior, good overall performance, and high speedup in these applications. Second, we show that when long (512 byte) cache lines are used, many of these applications suffer from false sharing and low cache line utilization. Third, we identify some of the common artifacts in compiler-parallelized codes that can lead to false sharing or other types of poor memory system performance, and we suggest methods for improving them. Overall, this study offers both an important snapshot of the behavior of applications compiled by current parallelizing compilers, as well as an increased understanding of the interplay between cache line size, program granularity, and memory performance in moderate-scale multiprocessors.

1 Introduction

Historically, parallel programming has been the domain of a relatively small group of highly knowledgeable and dedicated supercomputer users. Recent architectural advances, however, have propelled moderate-scale parallel computers into widespread use as general-purpose numeric compute servers. Increased reliance on *compiler-parallelized* applications will not only be a natural fallout from this transition to widespread moderate-scale parallel architectures; it will also likely be one of the driving forces that help bring it about.

Although parallelizing compilers are not always successful, evidence indicates they are finally reaching the stage where they can parallelize many interesting scientific codes. Their increasing success opens parallel computing to a broad spectrum of users, since parallel programs can be developed with little effort beyond what is required to develop sequential programs. In

a realm where parallel computers are widely available, compiler-parallelized applications are likely to be the workload of choice for most users. However, relatively little is known about their characteristics.

In particular, memory system behavior has been shown to have a significant impact on the performance of scalable multiprocessors [7, 10, 16]. Because of the increasing disparity between processor and memory speeds, memory systems have been evolving towards longer cache lines in order to hide memory latency. Researchers have studied how this trend affects carefully tuned hand-parallelized programs [22, 26]. In this paper, we examine the memory system behavior of a new class of applications—those amenable to compiler parallelization. Our goal is to evaluate how these programs are impacted by advanced memory systems.

This paper makes several contributions:

- We provide a detailed examination of the memory behavior of compiler-parallelized applications. Extending on previous work in [24], we characterize the application behavior on both PRAM (parallel random access memory) and more realistic memory models. We use an extensive collection of *real* applications taken from well-known benchmark suites such as SPEC, NAS, and PERFECT. We evaluate these benchmarks on full-sized data sets, using metrics such as speedups, memory overheads, causes of cache misses, cache line utilization and useful data traffic.
- Based on these measurements, we show that the *granularity* of application parallelism is an important determinant of application memory behavior, and ultimately of application performance. For many applications, we find granularity is a function of data set size.
- Finally, our research has led to a better understanding of the artifacts (*e.g.,* small inner parallel loops) that can cause excessive false and true sharing. We give examples demonstrating that advanced compiler techniques such as array privatization and interprocedural parallelization can increase parallelism granularity, improving memory system performance.

Overall, this study provides a snapshot of the interactions between current compilers, parallel architectures, and applications. Its results can benefit both architects and compiler writers for multiprocessors. Architects can observe how an important class of programs with characteristics different from hand-parallelized programs will behave in relationship to trends in architecture design. Compiler writers can apply these quantitative measurements to improve the behavior of compiler-parallelized applications and avoid problematic patterns. In particular, the effect of parallelism granularity on memory system behavior is vital for both architects and compiler writers to keep in mind when attempting to exploit fine-grain parallelism on advanced memory systems.

In the following sections, we describe the compiler, simulation methodology, and applications used in our experiments. We present our measurements for these programs, then examine the behavior of the compiler in greater detail before concluding.

2 The SUIF Parallelizing Compiler

For our study we used the SUIF parallelizing compiler [25] to generate parallel versions of our applications. SUIF takes as input sequential Fortran or C programs, producing as output parallel C programs that execute according to a master-worker model. For each program, SUIF

performed identical set of analyses and optimizations. SUIF contains most of the features found in commercial parallelizing compilers such as KAP; these techniques include data dependence analysis, supporting scalar analyses such as constant propagation, induction variable recognition and scalar privatization and reduction recognition. Additionally, SUIF performs both array privatization and interprocedural analysis. Section 7.3 will demonstrate the importance of these advanced features. While the SUIF compiler also incorporates new techniques for improving data locality, we disabled these optimizations for this study so that the compiler-parallelized code more closely matches that produced by today's commercial systems.

For this experiment, we use a very simple approach to generating parallel code. The compiler finds the outermost loop in a loop nest for which it is safe to perform parallelization. Once SUIF identifies a parallel loop, its iterations are divided at compile time so that each processor performs a roughly equal number of consecutive iterations. Thus, for a loop with `N` iterations executed on `P` processors, processor 0 performs the first $\lceil$ `N / P` $\rceil$ iterations, processor 1 performs the next $\lceil$ `N / P` $\rceil$ iterations, and so on. This simple static scheduling heuristic maintains locality and minimizes run-time overhead; measurements show it does not lead to significant load imbalance for our application suite. SUIF programs rely on a run-time system built from ANL macros for thread creation, barriers, and locks. The run-time system has been tuned to eliminate false sharing and minimize true sharing; it has been ported to the Stanford DASH [16], SGI Challenge, and KSR-1 multiprocessors.

3 Methodology

For these experiments, we used an extended version of the MemSpy simulator [18, 19] and the TangoLite simulation and tracing system [4, 8]. TangoLite allows simulation of parallel programs by multiplexing their execution on a uniprocessor workstation. Because of this multiplexing, simulation statistics will vary depending on how often one switches between simulated threads. To fully capture potential sharing between processors, in this work we choose to interleave between threads at each memory reference.

MemSpy supports monitoring cold, replacement, and invalidation cache misses on a procedure and data item basis. For our study we have further broken down the category of invalidation misses into *true sharing* and *false sharing* misses using the scheme described by Dubois *et al.* [5]. In this definition, a true sharing miss occurs if: during a *lifetime* of the line in the cache, the processor accesses a word written by a different processor since the last true, cold or replacement miss by the same processor to the same cache line. This classification captures the prefetching effect of multiword lines in communicating newly defined values. In addition, we also study *upgrade* misses. When a line is cached non-exclusively, other processors may also be caching it. A write to that line forces an "upgrade" to exclusive mode, in which the coherence hardware sends out invalidations to the other processors caching the line. This transition from non-exclusive to exclusive mode caching is referred to as an upgrade miss.

3.1 Memory Systems Simulated

For our study we simulated two basic memory systems. First, we used an "ideal" PRAM memory system to characterize each application. In our PRAM model the memory system includes caches, but all memory access latencies are 1-cycle, so cache hits and cache misses take the same amount of time to complete. Speedups on the PRAM system are limited only by the amount of parallelism discovered by the compiler and load imbalance in the parallel code. We

have chosen to present the bulk of our application characterization results using this PRAM model because it allows us to measure inherent caching behavior without incurring the timing variations caused by more detailed architectural models.

We include additional results on an advanced memory system that more closely resembles an aggressive next-generation multiprocessor. It has a directory-based cache-coherent non-uniform memory access memory system with a high speed interconnect [14, 16]. Each processor has a single-level, least-recently-used cache whose size, associativity, and line size we vary. The penalty for a cache miss is dependent on the line size; Table 1 shows the cache miss penalties calculated for these machine parameters on a 16 processor system, assuming no contention. The penalties are calculated in terms of processor cycles. A *local miss* occurs when a cache miss is satisfied by the local memory at each processor. A *remote clean* miss occurs when the value is located at a remote node and has not been modified. A *dirty remote* miss is when the value is located remotely but has been modified – this implies an additional network hop to locate the correct value.

Line size	Local miss	Remote clean	Dirty remote
8 bytes	48	180	264
16 bytes	49	185	269
32 bytes	50	194	278
64 bytes	52	212	296
128 bytes	56	248	332
256 bytes	64	310	394
512 bytes	80	444	528

Table 1: Cache miss penalties (cycles) vs. line size.

In comparison to our earlier study of compiler-parallelized codes [24], the machine model in this work includes a realistic simulation of write buffering, and more detailed simulation of contention. In particular, we model contention for the local memory bus and memory ports. We do not model contention for the network, however, because our numbers indicate that network contention is not likely to be a limiting factor. A round-robin page allocation policy is employed to reduce contention. Each processor has an 8-entry write-buffer. Finally, we use a single-cycle barrier synchronization cost.

For our study, we measure memory system performance over a range of cache line lengths (8 to 512 bytes), set associativities (direct-mapped, 4-way, and fully-associative), and cache sizes (8KB to 1 MB). Due to space limitations, we choose to present results mostly for a *baseline* memory system with a 1 MB, 128 byte line 4-way set-associative cache. The cache parameters were selected to model a forward-looking multiprocessor memory hierarchy.

4 Applications: Background and Motivation

The applications used in our study consist of codes taken from standard scientific benchmark suites; their characteristics are listed in Table 2. Since we wish to evaluate their memory system behavior, and not the effectiveness of the SUIF compiler at finding parallelism, we selected programs where SUIF successfully parallelizes most of the code. Thus, our chosen benchmarks are not representative of all compiler-parallelizable codes, but rather of those whose performance does not suffer from non-memory-related problems. (Previous work has compared some of

Program	Suite	Description	Input Data Set	(MB)	Refs. Simd. (10^6)	Parallel Coverage (%)	Parallel Gran. (10^6 cycles)
COMPILER-PARALLELIZED BENCHMARKS							
appbt	NAS	block-tridiagonal PDEs	12^3 grid	(1.30)	48	100	5.73
appsp	NAS	scalar-pentadiagonal PDEs	12^3 grid	(0.32)	18	98	0.30
cgm	NAS	sparse conjugate gradient	1400 array elems.	(2.88)	37	98	1.02
erle64	MISC	ADI integration	64^3 array elems.	(4.70)	18	100	14.59
flo52	PERFECT	transonic inviscid flow	40x8 grid	(1.09)	92	99	0.23
hydro2d	SPEC	Navier-Stokes	102x40 grid	(0.62)	22	98	0.02
mgrid	NAS	multigrid solver	32^3 grid	(1.09)	74	94	3.05
ora	SPEC	ray tracing	650 array elems.	(0.04)	31	100	213.92
simple	RICEPS	Lagrangian hydrodynamics	203x183 grid	(3.17)	62	94	2.36
su2cor	SPEC	quantum physics	8^3x16 grid	(4.22)	240	98	0.05
swm256	SPEC	shallow water model	256^2 grid	(3.73)	44	100	5.34
tomcatv	SPEC	mesh generation	256^2 grid	(3.72)	44	100	4.88

Table 2: Characteristics of scientific applications in study.

these codes to a set of hand-parallelized applications [24].) We define *parallel coverage* as the percentage of sequential program execution time spent inside parallel regions. Using pixie to instrument each program, we found parallel coverage of our programs by the SUIF compiler was between 94 and 100%. Sufficient parallelism has thus been uncovered; we can concentrate on evaluating the impact of memory system behavior on performance.

Another measure of parallelism is *granularity*, the amount of computation enclosed in each parallel region. Programs with high granularity synchronize infrequently, thus performing little communication relative to time spent doing useful work in parallel. We derived our granularity measurements by first determining the number of instruction cycles per invocation of each parallel loop, and then computed a weighted average of the number of cycles based on the percentage of sequential execution time spent in each loop. The resulting granularities for each program are presented in Table 2. As we will show in Section 6.3, for many applications the compiler's ability to exploit larger granularities of parallelism is correlated to good memory performance.

For brevity, in the remainder of the paper we shall refer to these programs as the SUIF applications. Each program uses the standard data set provided; in the case of the NAS benchmarks, we use the smaller of the standard data sets. By using standard data set sizes, we avoid the red-herring memory performance problems that might be caused by unrealistically small problem sizes. Where necessary we have reduced the number of time steps in each application to limit simulation time. To prevent initialization behavior from skewing results, we reset statistics after initialization and cold start.

4.1 Motivation

To motivate our research, Figure 1 shows speedups for 16 processor simulations for the PRAM and advanced memory systems described earlier. Speedups are calculated relative to a uniprocessor run on the same memory model. For the PRAM system, SUIF applications achieve average speedups of 12.1, demonstrating that compilers can exploit reasonable levels of parallelism for these scientific applications.

When we look at speedups for the more realistic baseline memory system, the picture is quite different. We found speedups remain quite high for some applications, but most drop significantly compared to the ideal memory system. At 16 processors, speedups for SUIF applications

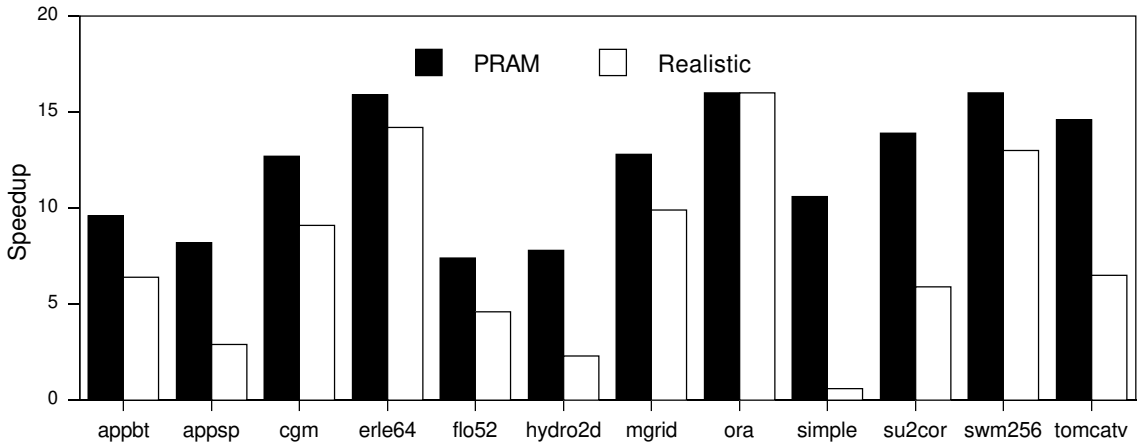

Figure 1: Application speedups for PRAM and advanced memory systems.

range from 0.6 to 16 with an average speedup of 7.6. These simulated speedups correspond well with actual speedups observed for these programs on the Stanford DASH and SGI Challenge multiprocessors [11, 25].

Clearly then, memory overhead is a primary factor in the less than linear speedups displayed by most of these applications. Ideally we would like to understand what aspects of these applications are not amenable to features of advanced memory systems when executing in parallel. Towards this goal, we present experimental data in two main categories. First, we use a PRAM model to explore basic application characteristics such as the working set size, the spatial locality, and the type and degree of sharing present. Following this, in Section 6, we shift to a more realistic memory system model to study the impact of these characteristics on program performance in machines with memory latencies representative of current parallel machines.

5 Application Characteristics on an Idealized Memory System

In this first section of results, we characterize the applications according to their behavior on the PRAM system previously described. This allows us to study the application sharing and memory referencing characteristics that are largely inherent, rather than tied strongly to a particular memory latency model. Our goal here is to establish a basic understanding about the inherent locality and sharing in these applications before moving on in Section 6 to study their performance with more realistic memory latencies. We measure both *temporal* and *spatial* locality by varying the cache size and cache line size respectively. Temporal locality refers to the property that if a value has been used, it is likely to be used again in the near future. Spatial locality refers to the property that if a value has been used, then it is likely that a value near to it in memory will be used in the near future.

5.1 Effect of Cache Size and Set Associativity

Figure 2 shows the cache miss rate for 16 processor runs of each compiler-parallelized application. Results are presented for direct-mapped, four-way, and fully-associative caches as the cache size varies from 8KB to 1 MB. The cache line size is held constant at 128 bytes. (Note that due to the widely varying miss rates, the y-axis scale differs for each application.)

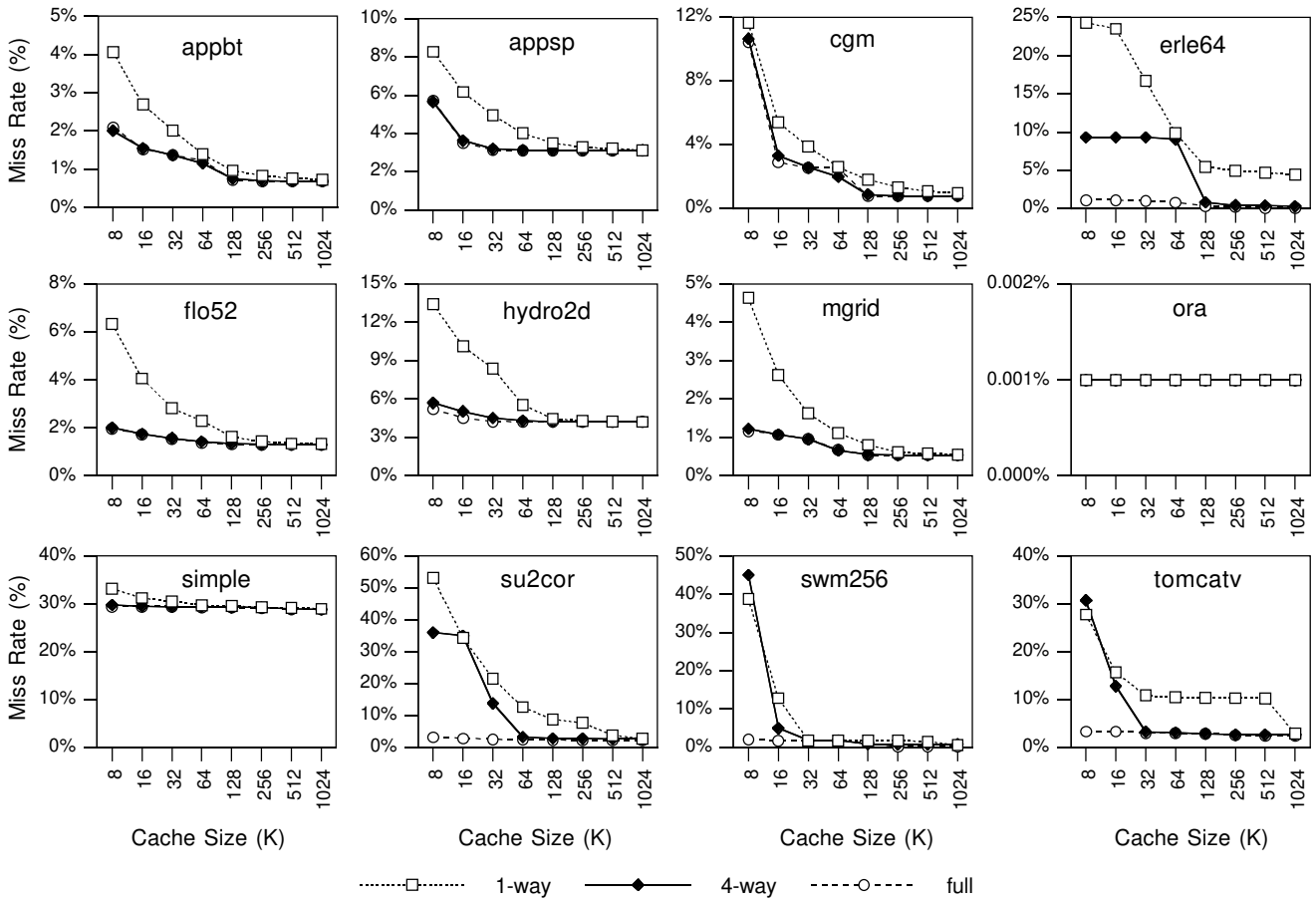

Figure 2: Cache misses vs. cache size and set associativity.

We see that applications generally show significant relative drops in their cache miss rates as the cache size is increased. We find that only two programs, appbt and cgm, possess sufficiently large working sets to benefit from caches larger than 64KB. By the time the cache size is increased to 1 MB, all the miss rates for all applications have plateaued; beyond this point, further increases in cache size do not reduce cache miss rate significantly.

Most applications had reduced miss rates when increasing associativity from direct-mapped to four-way caches. Smaller gains result going from four-way to fully-associative caches. Several applications show a large difference between four-way and fully associative caches for smaller cache sizes; for most these differences become negligible at 128KB cache sizes. For tomcatv, four-way and fully associative miss rates roughly merge at 32KB caches, but interference remains severe in the direct-mapped cache up to the 512KB cache size. For erle64, direct-mapped caches experienced conflict misses even at 1 MB. (Both of these applications possess multiple large data arrays that conflict in a 128KB cache.) Overall, we found 4-way and fully-associative caches generally yielded similar results.

These results demonstrate that our choice of a 4-way associative, 1 MB cache for our baseline advanced memory system is reasonable for avoiding excessive capacity and conflict cache misses. With this configuration, the essential spatial locality and interprocessor sharing is exposed.

5.2 Application Sensitivity to Cache Line Size (Spatial Locality)

As processor speeds continue to increase faster than memory speeds, there has been a corresponding trend towards increasing the cache line size. This increase attempts to take advantage of *spatial locality* in applications in order to amortize latency over more data. It is important to note that the decision to move to longer cache lines is being driven largely by *uniprocessor* system design. In uniprocessors, cache miss rates behave predictably with increasing line size, decreasing at first, eventually increasing as cache conflicts start to dominate.

Unfortunately, miss rates are not so predictable for multiprocessor caches [15, 23]. Longer cache lines may prove problematic for parallel codes for several reasons. First, *false sharing* may cause cache misses on logically separate data placed on the same cache line. Second, applications may exhibit less spatial locality when executing in parallel, depending on how computation is partitioned. Finally, longer cache lines may lead to increased data traffic, causing memory contention. Previous research has shown false sharing to be a problem for hand-parallelized applications [7]. Our study attempts to evaluate the effect of longer cache lines on applications amenable to compiler parallelization. By varying the cache line size, we are in effect measuring the spatial locality of compiler-parallelized applications.

In the subsections that follow, we look at the effect of longer cache lines on the parallelized applications, assuming a PRAM model. We examine their effect on (i) cache miss rates and causes, (ii) data and coherence traffic, and (iii) cache line utilization.

		Cache Miss Rate (vs. line size)				Change		
Run	Miss Type	8B	32B	128B	512B	8B →32B	32B →128B	128B →512B
1 proc	All	6.17%	1.56%	0.40%	0.11%	-74.7%	-74.3%	-72.7%
16 proc	All	8.83%	3.04%	3.83%	6.30%	-65.6%	+26.3%	+64.2%
	False	0.10%	0.34%	2.24%	5.02%	250%	553%	125%
	True	4.05%	1.31%	0.61%	0.41%	-68%	-54%	-33%
	Cold	0.62%	0.17%	0.06%	0.03%	-72%	-67%	-54%
	Repl	1.60%	0.42%	0.12%	0.07%	-74%	-71%	-45%
	Upgrade	2.45%	0.79%	0.81%	0.77%	-68%	2%	- 5%

Table 3: Cache miss rates for varying line sizes. Numbers represent averages calculated over all twelve SUIF applications.

5.2.1 Cache Misses

Table 3 presents average cache miss rates for a four-way set associative 128KB cache. The average is calculated by adding each application's miss rate together and dividing by the total number of applications. They show that for uniprocessors SUIF programs have sufficient spatial locality to reduce cache misses with lines up to 512 bytes. In the one-processor results, the miss rate drops steadily, and almost proportionally to the increase in cache line length.

With 16 processors, however, the picture changes. The transition from 8-byte lines to 32-byte lines improves the miss rate for all 12 of the applications. On average, the miss rates improve by 65.6%. In the transition from 32-byte lines to 128-byte lines, ten out of twelve applications get further benefits from the increased line size. The two applications which do not benefit from 128-byte lines are simple and su2cor. In particular, simple has a large jump in false sharing misses. At 32 byte lines, 1.9% of simple's references result in false sharing misses, while with

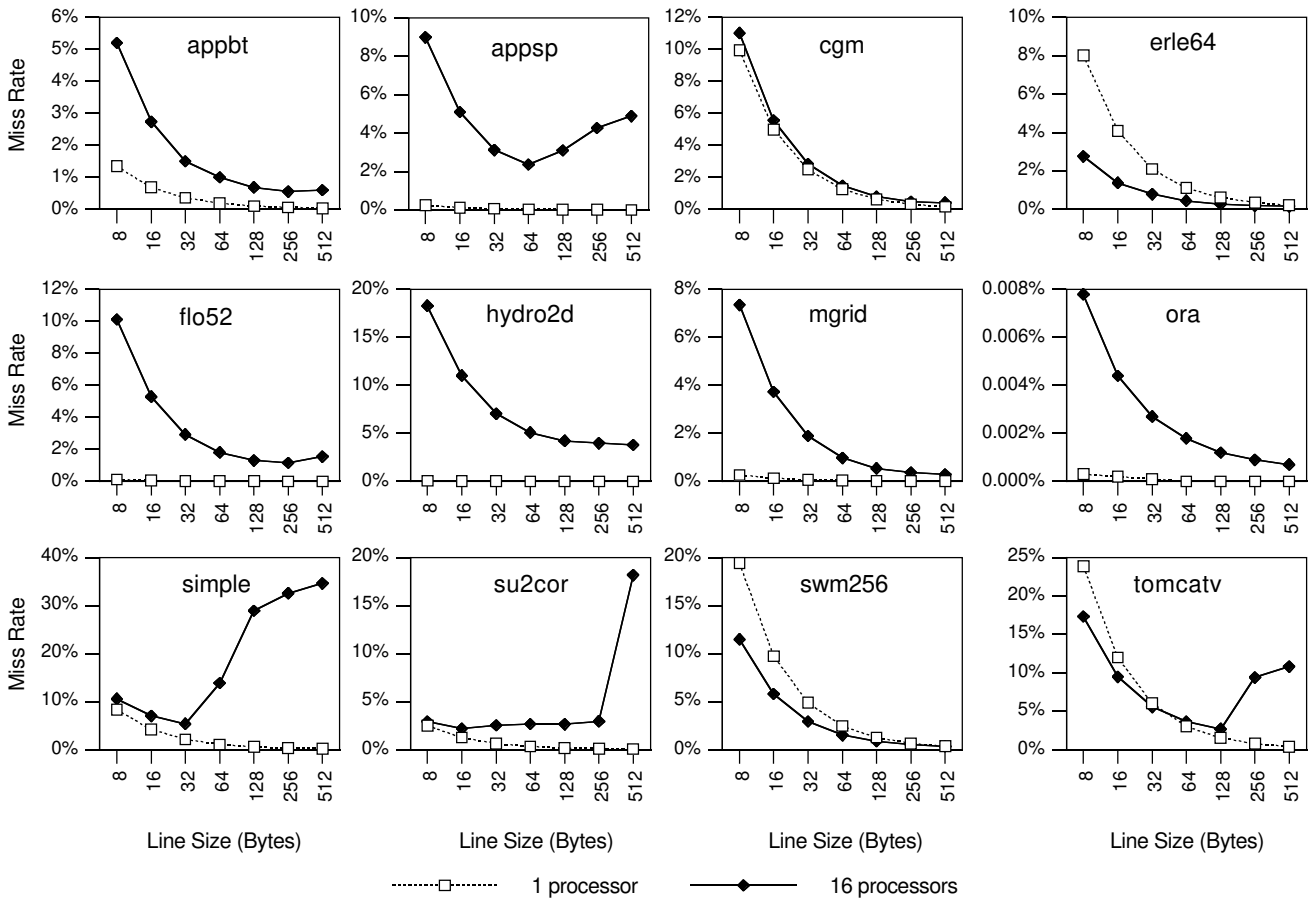

Figure 3: Cache misses vs. cache line size (1 and 16 processors).

128 byte lines, this percentage jumps to 22.1% (and skews the overall averages seen in Table 3.)

In moving from 128 byte lines to 512 byte lines, the applications get less benefit. Seven out of twelve applications have improved miss rates at these very long cache lines, but in Section 6 we will show that these improved *miss rates* do not necessarily translate into improved performance. For the other five out of twelve applications (appsp, flo52, simple, su2cor, and tomcatv) increasing the line size causes increases in miss rate. While appsp and flo52 have a relatively moderate increase, from 3.1% to 4.9% and from 1.3% to 1.6% respectively, the miss rates in the other three applications increase drastically, resulting in double-digit miss rates at the 512 byte line size. Thus, although most of the applications appear to have sufficient spatial locality to take advantage of 128 byte lines, cache lines longer than that lead to excessive sharing in several of the programs.

To look at this in more detail, Figure 3 shows the cache miss rate and causes of cache misses for each application over a range of cache line sizes. The figure shows that the applications divide themselves into two main categories. Seven of the twelve applications (appbt, cgm, erle64, hydro2d, mgrid, ora, and swm256) have miss rates that drop monotonically with increases in line size. These applications are dominated by the uniprocessor effects dictating that program spatial locality will allow increases in cache line size to reduce cold misses without significantly increasing false sharing, and thus improve overall caching performance.

Four of the remaining applications (appsp, simple, su2cor, tomcatv) have significant inter-

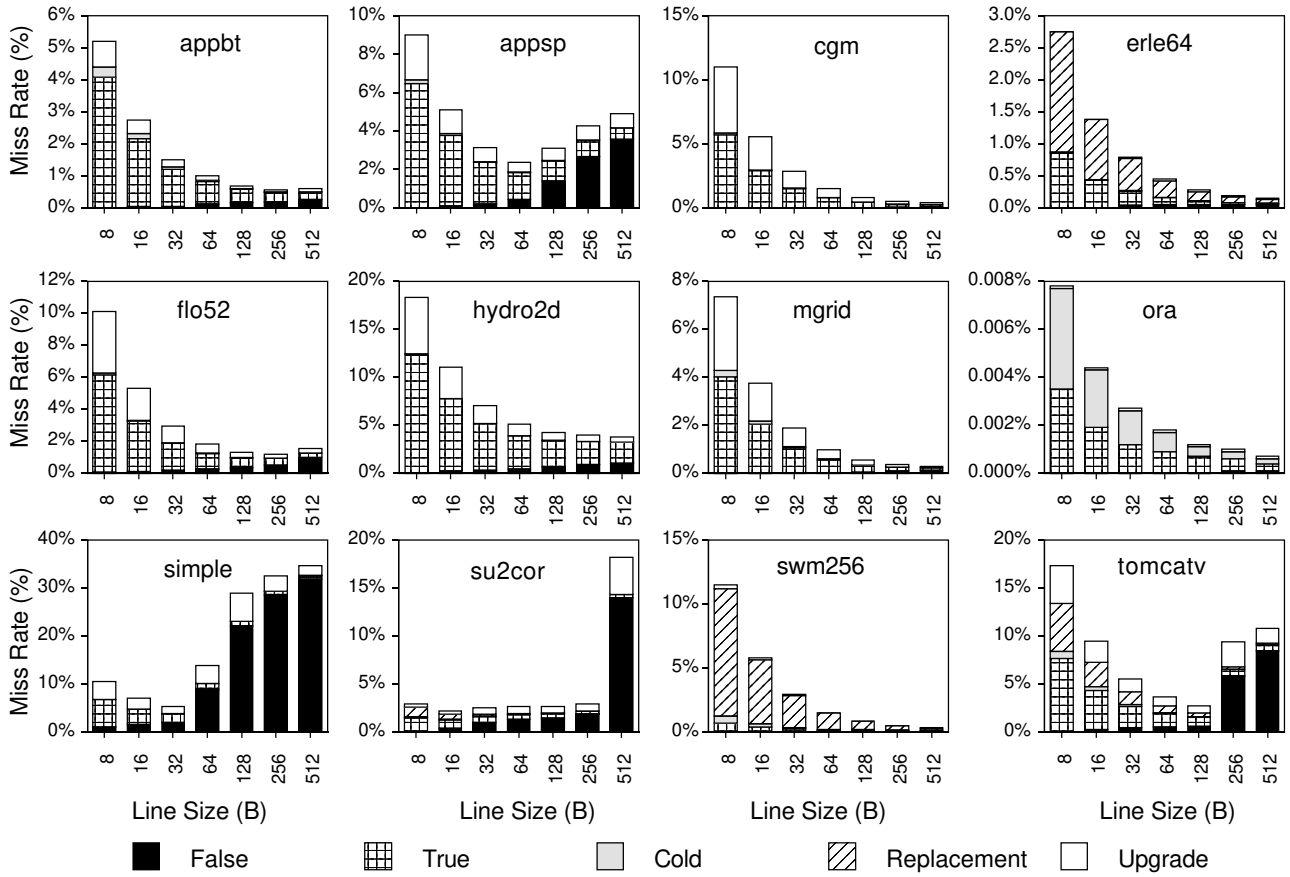

Figure 4: Cause of cache misses vs. cache line size (16 processors).

processor sharing which causes their miss rate graphs to take on a different shape. For these four programs, increases in line size from 8 bytes begin by causing improvements in cache miss rates; the benefits from spatial locality outweigh the costs of increased false sharing. Beyond a certain point however (64 byte lines for appsp, 32 byte lines for simple, 16 byte lines for su2cor, and 128 byte lines for tomcatv), false sharing effects begin to dominate. For these four applications, false sharing misses represent over half the program misses when the cache line size is 256 bytes. The final application (flo52) has a well-behaved monotonic decrease for all but the largest cache line size, where it increases slightly.

Figure 4 presents cache miss rates and their causes in more detail for our baseline system. The overall miss rate of each application is indicated by the height of the bar. Within each bar, shadings represent different causes of cache misses. The bars are broken down into five possible types of cache misses: (i) *false sharing* misses, (ii) *true sharing* misses, (iii) *cold* (or compulsory) misses, (iv) *replacement* misses, and (v) those occurring when a line is written to while in non-exclusive mode, resulting in an *upgrade* operation to transition to exclusive mode.

For programs that exhibit "perfect" spatial locality, increasing the line size by a factor of n will *decrease* the number of cold and replacement misses by the same factor n, ignoring cache conflicts. Increasing the line size four times can thus potentially reduce cache misses by 75%. Of the three SUIF programs (erle64, swm256, and to a lesser degree tomcatv) which are dominated by a large number of replacement misses, the replacement misses exhibit good but not ideal spatial locality. Only two SUIF applications, cgm and ora have almost ideal behavior for cold

and replacement misses as the line size is increased.

Even if an application has perfect spatial locality in a single processor reference stream, the interleaving of references from multiple processors introduces the possibility of false sharing misses. Figure 4 shows false sharing misses tend to increase as the line size is increased. In general this increase is slow, and is more than compensated for by the corresponding *decrease* in the other classes of misses. All SUIF applications except simple reduced miss rates going from 32 to 128 byte lines, and 7 of 12 continued to reduce misses going from 128 to 512 byte lines. It is important to note that when false sharing is encountered, its effect is frequently drastic (as in simple). Section 7.1 discusses distinguishing access patterns in compiler-parallelized programs that lead to excessive false sharing.

From Table 3, we see that cold misses come closest to ideal improvements. Averaged over all the applications, they decrease 72% in going from 8 byte to 32 byte lines, 66% between 32 bytes and 128 bytes, and an additional 52% in going from 128 bytes to 512 bytes. While true sharing miss rates decrease as line size increases, the decrease is not as great as for cold or replacement misses. For the same line size transitions as above, true sharing misses decrease by 68%, 55%, and 31% respectively. This result suggests that lines that are invalidated from the cache (*i.e.* actively shared lines) exhibit less spatial locality than other cache lines. In the following section, we investigate this observation further.

5.2.2 Data and Coherence Traffic

Cache miss rates only tell one half of the story. Figure 5 illustrates how a particular miss rate translates into an application's actual data traffic requirements. This figure displays data traffic for each program in bytes per instruction, which adjusts for varying run times of the applications, and gives intuition about the applications' computation to communication ratios.

Traffic is divided into six categories. *Shared* traffic corresponds to inherent communication data traffic and comprises both true and false sharing misses. *Cold* and *replacement* traffic represent cold and replacement miss data traffic. *Writeback* is data written back to memory because it has been modified in the local cache. *Overhead* is comprised of the 16-byte header information associated with each network transaction. Finally, *local* traffic is data written to and from the local memory for each processor – this traffic does not go over the network.

The applications fall into several groups. One application (ora) has essentially negligible traffic at any cache line size. Six of the applications (appbt, cgm, erle64, flo52, mgrid, swm256) have less than 1 byte of data traffic per instruction for all but the longest cache lines. The other five applications (appsp, hydro2d, simple, su2cor, and tomcatv) all have significant data traffic requirements. For 512 byte lines, these programs have data traffic of 5 or more bytes per application instruction executed.

5.2.3 Cache Line Utilization

To study traffic requirements in more detail, we use another metric related to data traffic that we term *cache line utilization*. For each line of data brought into the cache, cache line utilization indicates what percentage of the line is touched before the line is evicted or invalidated from the cache. This metric is a good indicator of spatial locality, because it shows how effectively the program is making use of longer cache lines. (This metric is similar to one that Torrellas et al. used in [23], where they studied the number of words touched in a cache line.)

Figure 6 gives a utilization graph per application for a 16 processor execution. The shaded

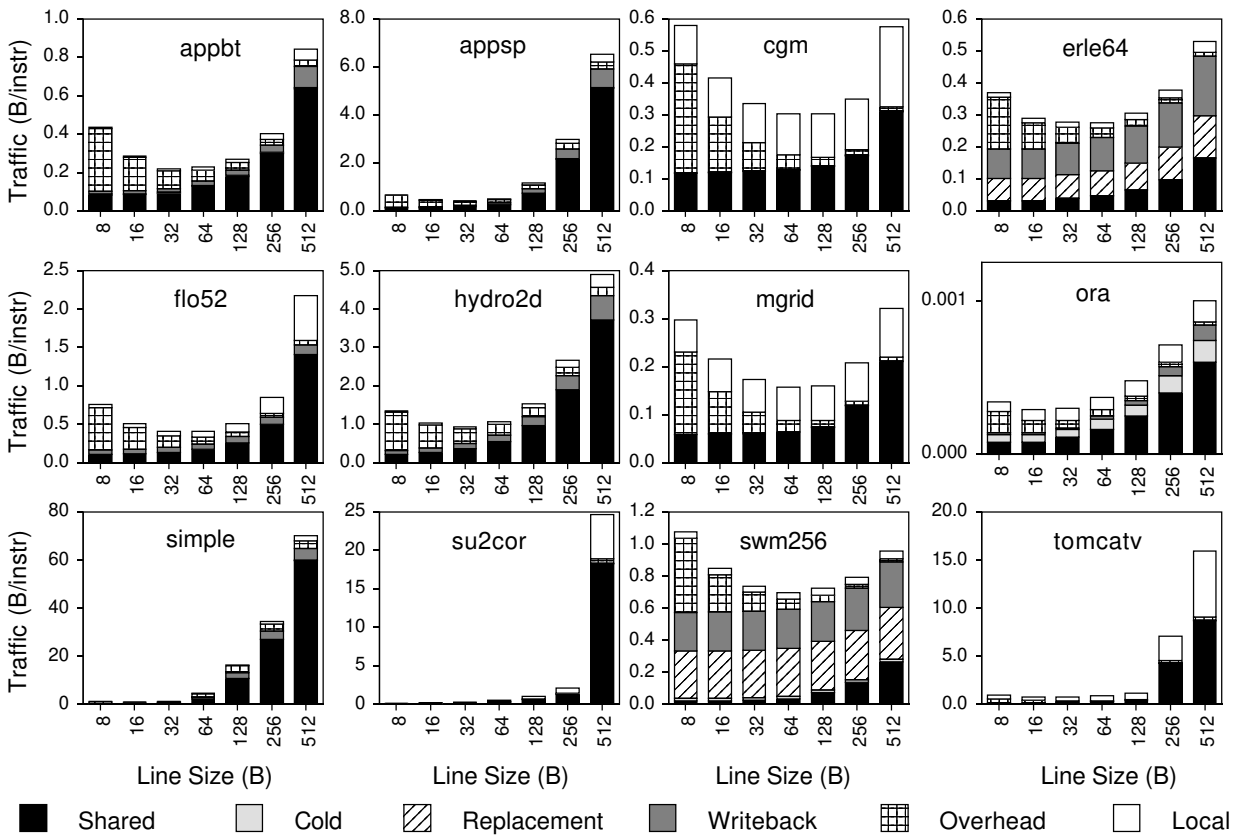

Figure 5: Data traffic vs. cache line size.

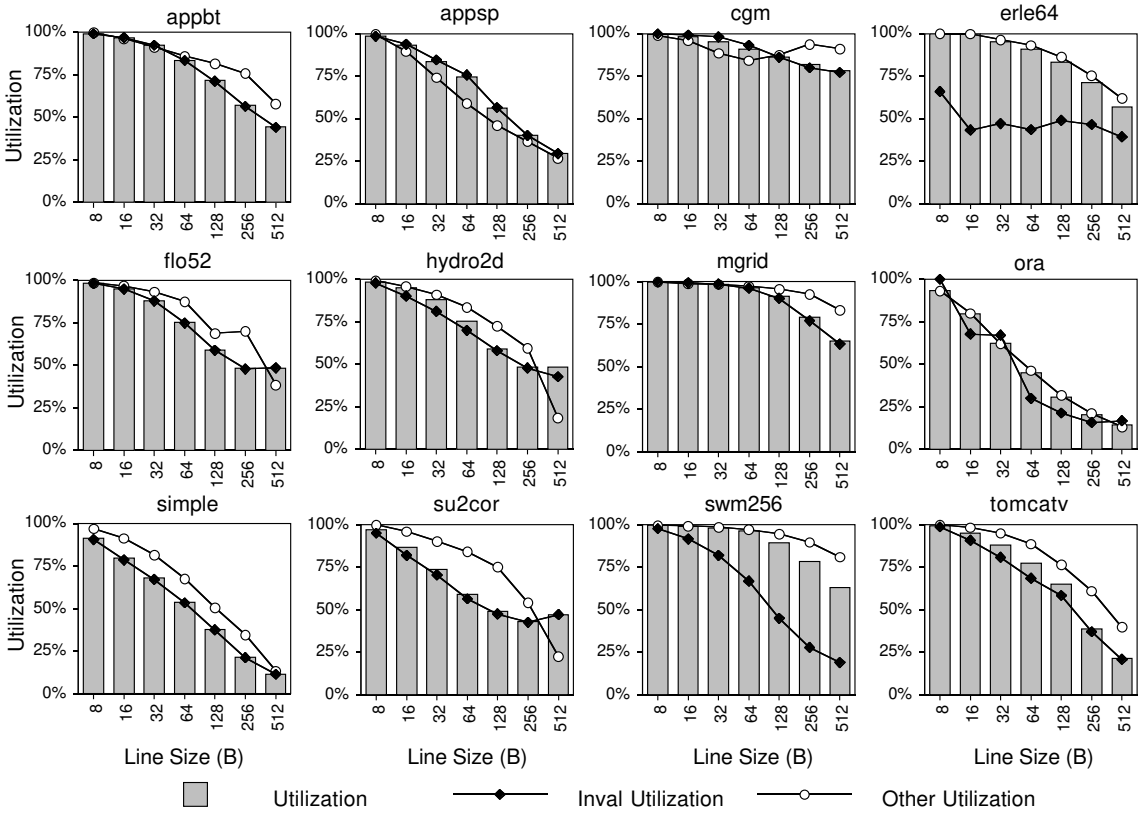

Figure 6: Cache line utilization vs. cache line size (16 processors).

bars indicate the overall cache line utilization for each application across a range of cache line sizes. The dark boxes show the utilization trends for the lines that were invalidated out of the cache, while the white boxes show the utilization trends for the remaining lines.

It is apparent that overall utilization rates decrease as cache line sizes increase, sometimes very quickly. The average utilization for SUIF programs drops from 85.5% at 32 byte lines to 64.9% at 128 byte lines. In going from 128 to 512 byte lines, it drops again to 43.6%. The low utilizations indicate that much of the data fetched into cache for longer cache lines is unused.

5.2.4 Classification of Utilization

As we have seen, low cache line utilization leads to large amounts of unused data traffic. In order to focus on the *causes* of poor cache line utilization for these applications, we ran experiments in which we further divided utilization statistics into two categories: lines that leave the cache due to invalidation and the remaining cache lines. The curves formed by the dark and light boxes in Figure 6 show the the utilization for invalidated cache lines, and all other cache lines, respectively. For most applications, with a 128 byte line size, invalidated cache lines exhibit lower utilization than the other cache lines. The intuitive explanation is that if lines are evicted by invalidation, either true or false sharing may have led to the line being prematurely evicted from the cache. As the line size increases, a larger fraction of the misses become invalidation misses, the "other" set can be quite small, and so the statistics of that set are sometimes (as in hydro2d and su2cor) based on too small a set of references to be representative.

Overall, distinguishing utilization behavior can be important, because the bimodal behavior

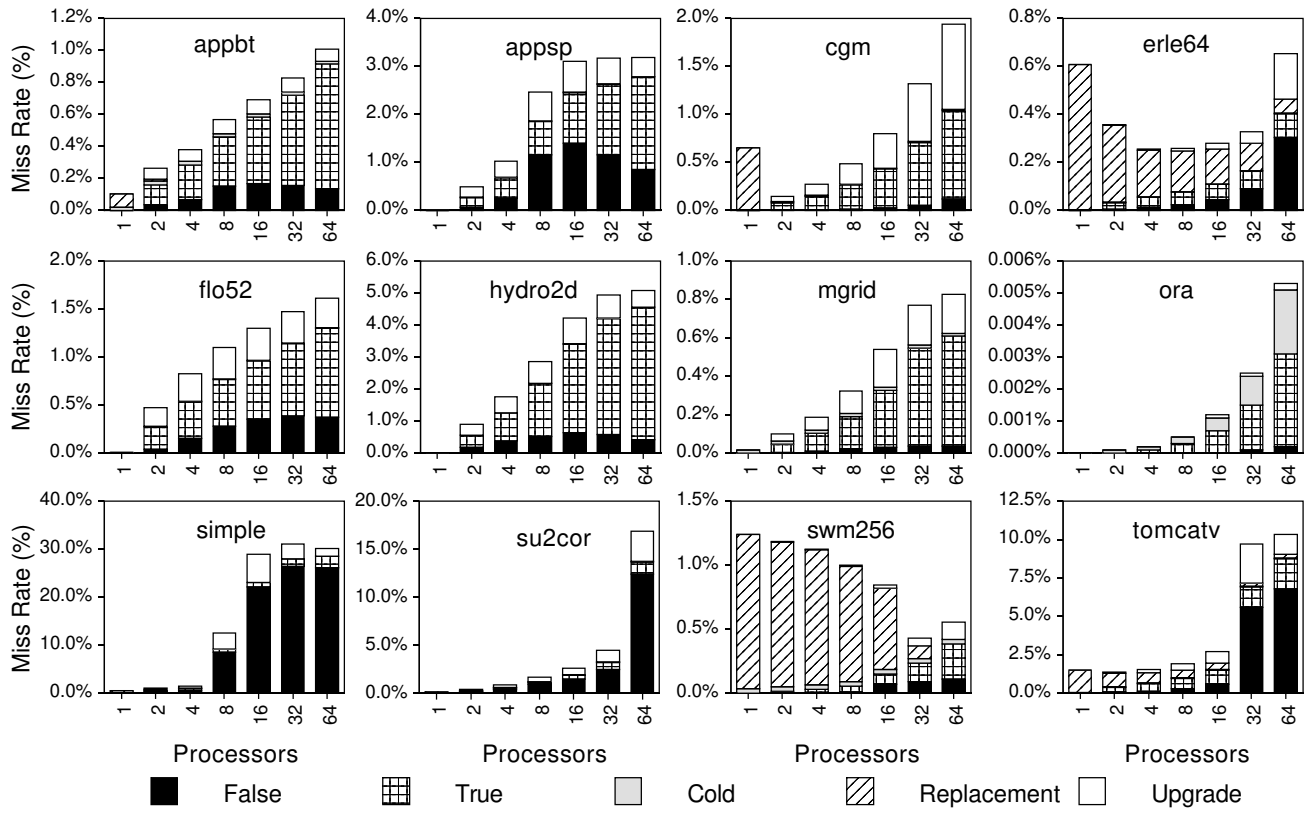

Figure 7: Cause of cache misses vs. processors.

suggests that special optimizations for each type of behavior may be possible. In systems allowing flexible protocols (such as Tempest [21]), one could specialize handling for each type of data. Essentially, the protocol could implement smaller coherence units for the previously-invalidated data, while maintaining coherence units equal to the cache line size for the previously replaced data. Alternatively, *prefetching* techniques could make use of this information to focus efforts on fetching non-invalidated cache lines in order to exploit their higher degree of spatial locality.

5.3 Effect of Number of Processors (Communication-to-Computation Ratio)

In addition to measuring the spatial locality of compiler-parallelized applications, we would also like to measure the scalability of these applications by measuring their inherent communication as the number of processors increases. By varying the number of processors, we are in effect measuring the computation-to-communication ratio of compiler-parallelized applications.

5.3.1 Cache Misses

Figure 7 shows the increase in cache miss rates as the number of processors is increased from one to 64 for our PRAM memory system. As with previous metrics, ora is extremely well-behaved, with low miss rates for all numbers of processors. Three applications, cgm, erle64 and swm256, have miss rates that actually decrease as one moves from a single processor run out to four and eight processor runs. The bulk of these misses are due to cache replacements, and as more processors are added, the miss rate drops. This is because the total cache of the machine has

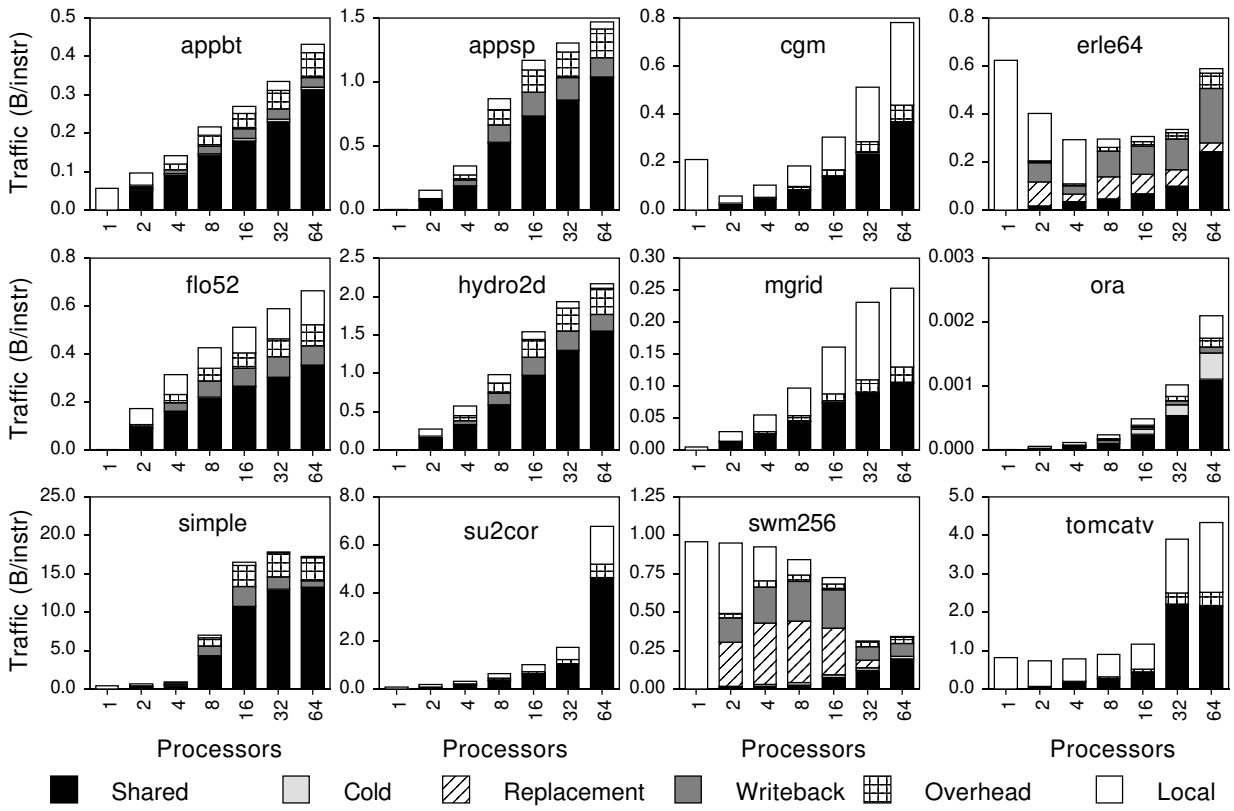

Figure 8: Data traffic vs. processors.

increased. For cgm, the drop is from a uniprocessor miss rate of 0.65% to its minimum miss rate at two processors of 0.15%. For erle64, the minimum miss rate is 0.26% and occurs at eight processors. For swm256 the minimum miss rate is 0.43% and occurs at 32 processors.

A third group of applications (appbt, appsp, flo52, hydro2d, and mgrid) start out with quite low uniprocessor miss rates. As the number of processors is increased, however, the miss rates climb steadily. In these applications, the primary cause of the increase is true sharing. That is, as the number of processors increases but the problem size remains constant, multiple processors are more likely to require pieces of data being generated by other processors. The program artifacts that lead to true sharing are described in more detail in Section 7. For hydro2d, its 64 processor miss rate is a very significant 5.1%. For the other 5 applications in this group, miss rates at 64 processors range from roughly 1% to 3%.

A fourth group of applications (simple, su2cor, tomcatv) is also characterized by relatively low uniprocessor miss rates. As the number of processors is increased here, however, the miss rate climbs due to *false sharing*. False sharing occurs with increasing processors, because as each processor's chunk of work becomes smaller, its data is more likely to share a cache line with some other processor's data. This excessive false sharing leads to some of the highest cache miss rates seen for the benchmarks. For example, simple has miss rates of roughly 30% for 16, 32, and 64 processor runs. The other two applications, su2cor and tomcatv, have miss rates of 16.9% and 10.3% for the 64 processor runs.

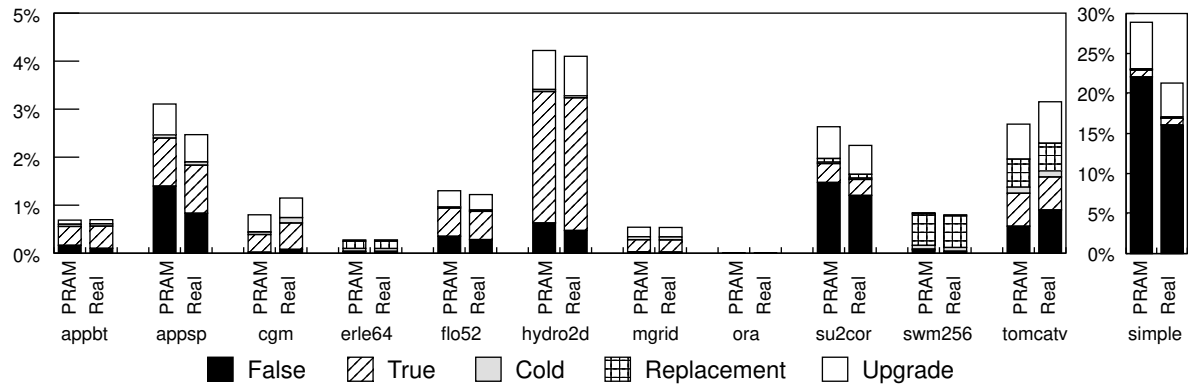

Figure 9: Cache misses vs. machine model.

5.3.2 Data and Coherence Traffic

Figure 8 displays the data traffic requirements for the different applications running with varying numbers of processors. As before, this is plotted in terms of the bytes of data required per instruction, which allows us to compare the computation to communication requirements of the different applications. The applications group themselves into categories roughly as they did in Figure 7.

Some applications such as ora, swm256, and (perhaps to a lesser extent) erle64 are extremely scalable. Two of these, swm256 and erle64, benefit from the increased total cache present as we increase the number of processors, while ora has almost negligible traffic to begin with and can thus scale more easily.

The remaining programs display increases in remote sharing traffic as the number of processors is increased. As we saw in Figure 7, this is due to true sharing in six applications (appbt, appsp, cgm, flo52, hydro2d, and mgrid). In the remaining three applications (simple, su2cor, tomcatv) the traffic increase is due to false sharing. The traffic is by far the most significant in simple, where a 16 processor run requires about 16 bytes of data to be transferred into the cache for each instruction executed. (Although a single instruction clearly cannot *reference* 16 bytes of data, it can cause an entire 128 byte cache line to be brought in. Thus, on average it is possible for the number of bytes of data transferred per instruction to be significantly larger than the amount of data addressable by a single load or store instruction.)

6 Application Characteristics on Advanced Memory Systems

The PRAM model is useful for measuring inherent sharing characteristics that are machine independent, but a realistic memory model can provide more accurate information on the impact of issues such as memory access costs and contention. In this section we evaluate the impact of a realistic advanced memory system.

One of the concerns with using a detailed memory model is that variations in the machine parameters can affect timing and skew results. Figure 9 demonstrates the difference in cache miss rates between the PRAM and baseline advanced memory model. We see that there are differences in the absolute miss rates, but qualitatively the relative application behavior is very similar.

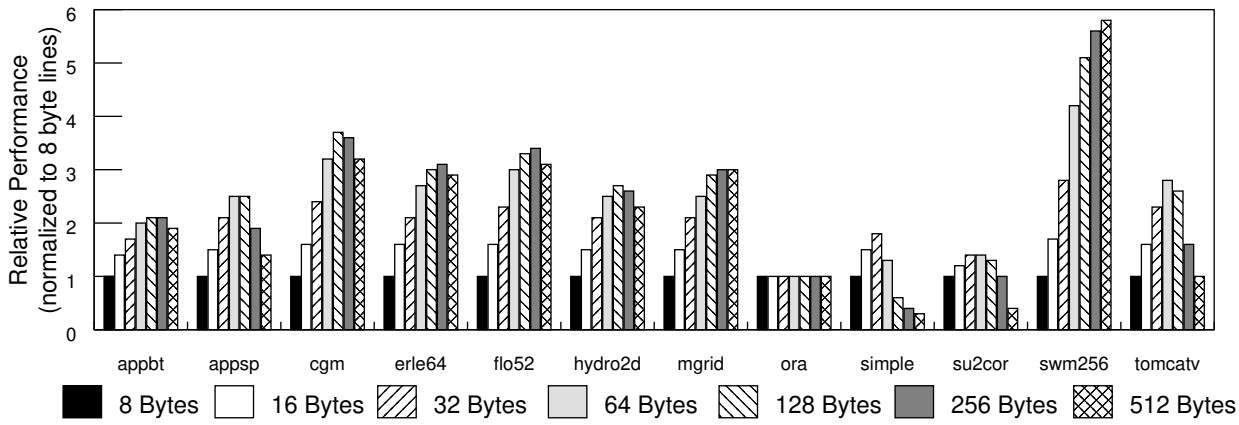

Figure 10: Relative application performance vs. cache line size.

6.1 Effect of Cache Line Size

Uniprocessor experiments show that the performance of SUIF applications improved an average of 36% going from 32 to 128 byte lines and 8% going from 128 to 512 byte lines. While these applications do display good uniprocessor spatial locality, we find that as parallel programs, their performance does not necessarily uniformly improve as cache lines grow longer. Figure 10 displays the change in performance for 16 processor executions as cache line size varies. (The y-axis shows performance relative to the simulated wall clock time for the 8-byte line run; that is, a bar of height 2 means that run was twice as fast as the base run for that application.)

Of the twelve applications, only one (swm256) receives any performance benefit from increasing the cache line size to 512 bytes. Four of the applications reach peak performance at 256 byte lines, while three applications reach peak performance at 128 byte lines. In the remaining applications, either minimal memory activity (ora) or excessive sharing (simple, su2cor, and tomcatv) limits the best cache line size to be quite small. Over all the applications, the average application improves performance by 27.8% by moving from 32 to 128 byte lines, but drops by 14.6% when going from 128 byte to 512 byte lines.

We discover a similar result when looking at the number of miss cycles per instruction (MCPI). Figure 11 displays the MCPI of 16 processor executions for different cache line sizes. The shading of each bar indicates three different types of stalls: (i) read stalls, (ii) stalls due to contention at the memory ports, and (iii) write stalls.

Increasing the cache line size from 32 to 128 bytes reduces MCPI for all SUIF programs except simple and su2cor. Removing simple (clearly an outlier) from the calculations, average MCPI *decreases* from 1.26 to 0.78. In comparison, going from 128 to 512 byte cache lines *increases* MCPI for 7 of 12 programs, with average MCPI (again excluding simple) jumping back up to 1.65.[1] It thus appears that for the given data sets and cache configurations, SUIF programs were able to exploit 128 but not 512 byte lines.

The shading indicating contention also highlights an interesting difference between some of the applications. The bulk of the applications (appbt, cgm, erle64, flo52, su2cor, and swm256) form a group of programs in which there is only moderate contention at small cache lines, and the contention decreases steadily as the line size increases. This is because the good spatial locality

[1] Including simple, the three average MCPI would be 1.50, 2.46, and 5.02 respectively.

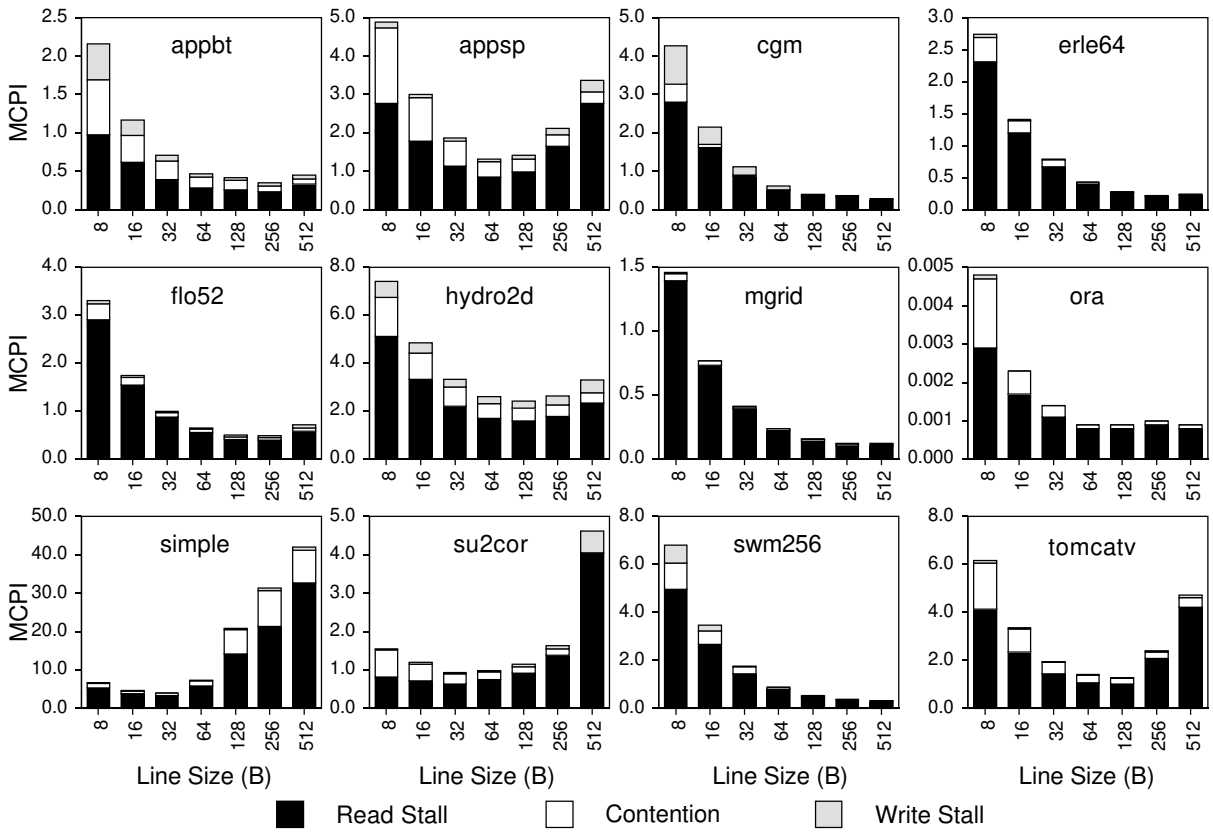

Figure 11: Miss cycles per instruction (MCPI) vs. cache line size. The runs execute with 16 processors.

in these applications causes the longer cache lines to be used efficiently. Other applications, like appsp, hydro2d and tomcatv, show similar trends although they display higher absolute contention stalls at short cache lines. On the other hand, simple is an example of an application which starts out with fairly low contention at short cache lines, and then displays increases in contention as the line size increases. This is further confirmation of simple's inability to make efficient use of long cache lines. With 512 byte lines, false sharing is so extreme that simple spends nearly 42 cycles per instruction in memory stalls, and 8.4 of these cycles per instruction are in contention at the memory port.

Note the vital role memory system behavior plays in determining the performance of the SUIF applications. For 128 byte cache lines, the average MCPI of 2.46 indicates over twice as many cycles are spent on memory accesses as on useful computation. For 512 byte cache lines SUIF applications spend 5 times as many cycles on memory accesses as on useful computation. For a few applications, MCPI is particularly high. In such cases poor memory system behavior severely degrades performance and may even cause slowdowns compared to sequential execution.

Figure 12 shows how the wait times of synchronization operations are affected by the memory system. This figure breaks down cycles spent within synchronization operators into three categories. First, there is time (due to load imbalance) at each barrier ending a parallel region. This time includes both the cost of passing through the barrier, as well as any wait time at the barrier. Second, there is time spent at each of the barriers that end sequential regions of code. At these barriers, N-1 processors are idle, waiting for the other processor to perform some task

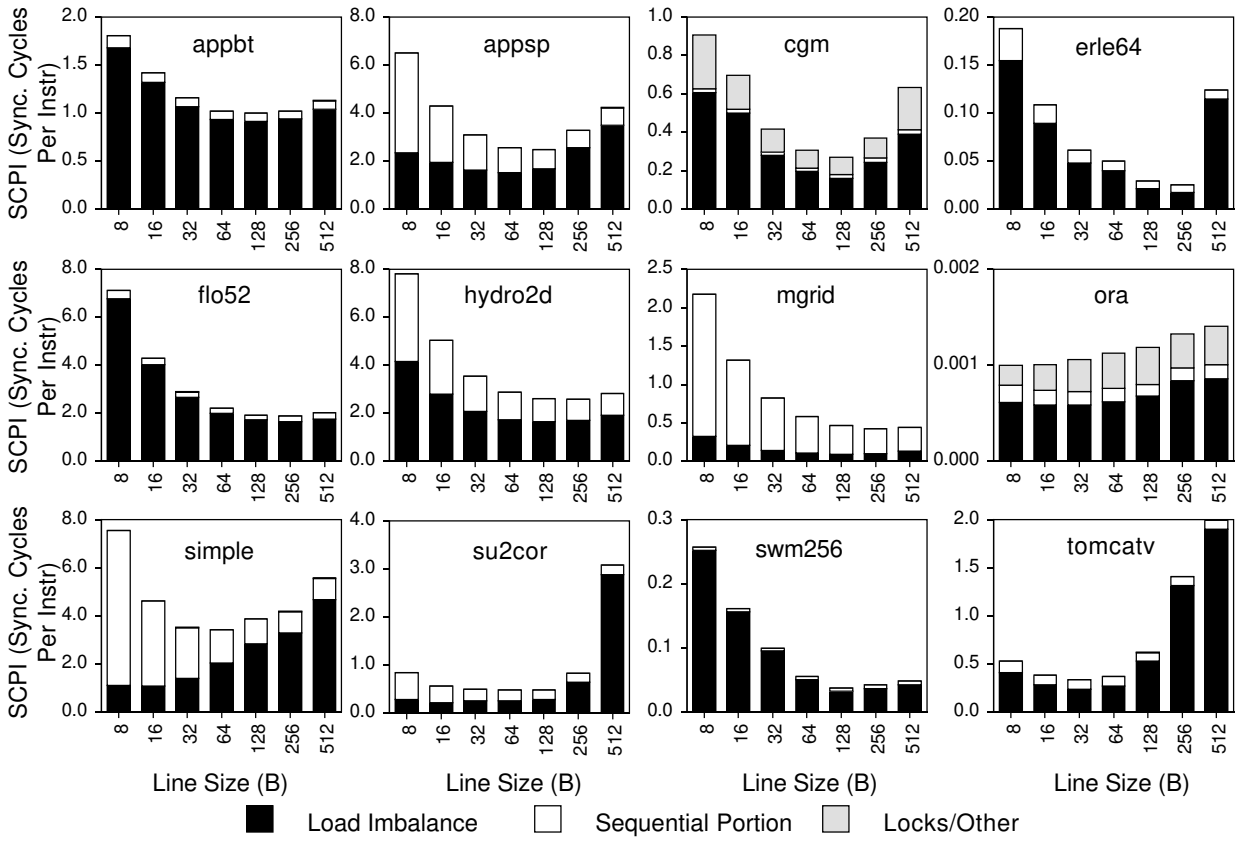

Figure 12: Synchronization cost vs. cache line size.

sequentially.[2] Finally, in two of the applications, a third category of time – time spent in locks and other synchronization – is also significant.

Overall, we find that for these applications, synchronization overhead, especially due to load imbalance, can be significant. Many synchronization operations such as passing parallel function arguments take place via shared memory, so cache line size affects parallelism overhead. In addition, memory system effects may increase the variance of task execution times, increasing load imbalance between processors. This latter effect is evidenced by the fact that for most of the applications, SCPI is minimized at a 128 byte cache line size. Recall that this is the most efficient line size in terms of program performance; by minimizing the task time, we tend to reduce the variance between tasks which leads to improved load balancing.

6.2 Effect of Number of Processors

In Section 5, we examined the effect of varying numbers of processors on the rate and causes of application cache misses. Here, we revisit this issue, this time studying the effect on performance in a more realistic memory model.

Figure 13 shows the average MCPI vs. the number of processors, for 1 to 64 processors. For almost all the applications, MCPI increases with increasing processors. This is because for these

[2] Although this wait time is primarily due to insufficient parallelism, it appears in our measurements as synchronization cost.

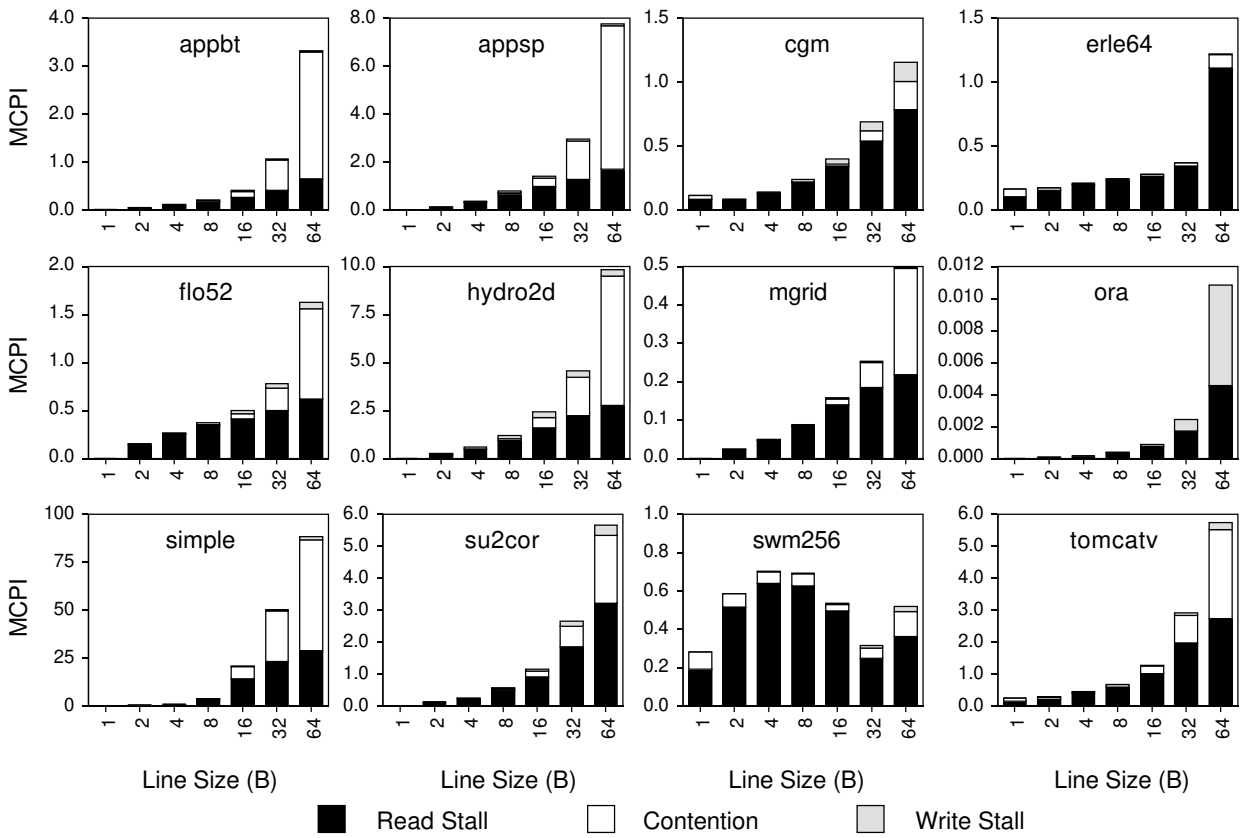

Figure 13: Miss cycles per instruction (MCPI) vs. number of processors.

applications and data sets, at 64 processors, memory performance degradation due to true and false sharing generally outweighs any possible benefits due to increased total cache size. (The only exception to this is swm256, which does show MCPI at 32 processors that is comparable to the uniprocessor MCPI.)

Paralleling MCPI, Figure 14 shows that SCPI typically increases strongly with the number of processors used. For most of the applications, synchronization overhead is comprised almost entirely of load imbalance overhead once the number of processors is increased to 64. This is in part due to the fact that we are scaling up the number of processors without changing the problem decomposition. It is also, however, due to memory system effects; as MCPI increases with the number of processors, so does the task variance leading to load imbalances.

6.3 Memory Behavior and Task Granularity

This section has provided a baseline characterization of the memory behavior of compiler-parallelized applications running on multiprocessors. It is important to note that we are using *full-sized* data sets for all the applications discussed. Our conclusions are thus being drawn not based on toy data sets, but on the default data sets provided by SPEC, NAS, and other benchmark suites.

Overall, we observe a high correlation between granularity of parallelism and good memory system behavior. To establish this trend, we have partitioned the SUIF programs in our study into three groups, based on whether their granularity was in the top, middle, or bottom third

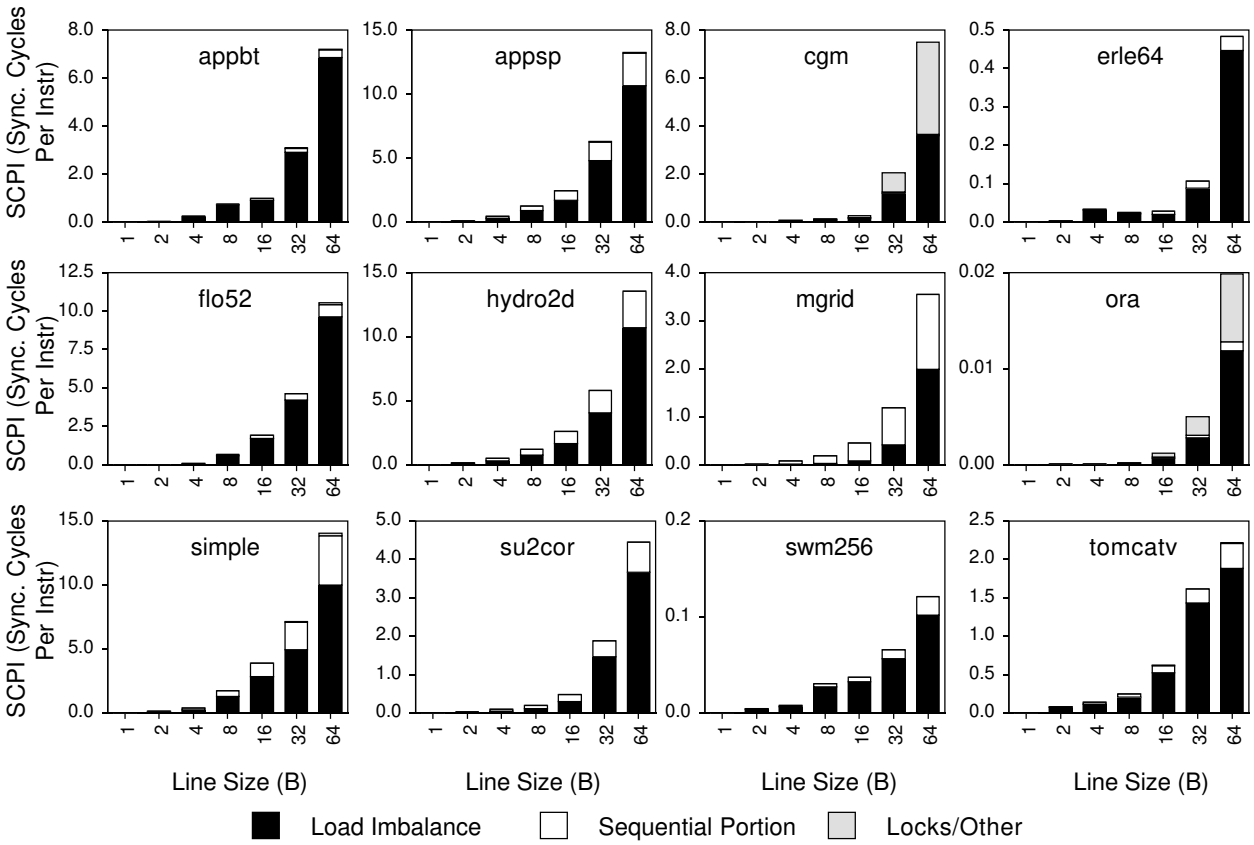

Figure 14: Synchronization cycles per instruction (SCPI) vs. number of processors.

of our suite. Their memory system behavior is summarized in Table 4. We find that SUIF applications with larger granularity have the best memory system performance, resulting in both greater speedups and higher percentage of ideal speedups achieved. Medium granularity SUIF applications have higher false sharing misses on average, skewed by the poor behavior of simple. SUIF applications with smaller granularity have generally poor performance, with especially high true sharing miss rates. Thus, while many compilers have so far striven for coarse-grain parallelism in order to reduce synchronization overhead, we provide compelling evidence of its importance for memory performance as well.

We can compare our results for SUIF applications to similar numbers collected previously for SPLASH [24]. Unfortunately, we also find that, overall, even coarse-grain SUIF applications for these problem sizes have relatively high memory system costs when compared to hand-tuned SPLASH applications. For the baseline memory system, SPLASH benchmarks tended to have lower MCPI and miss rates than the set of SUIF applications. To focus in greater detail on the causes of higher memory system cost in these codes, the following section examines particular application and compiler characteristics that have significant impact on memory system behavior.

Metric		Larger Grain	Med. Grain	Smaller Grain
Granularity (10^6 cycles)		59.9	2.8	0.2
Parallel Coverage		99.9%	96.5%	95.0%
Ideal Speedup (16 proc)		14.4	12.7	9.3
Sim'd. Speedup (16 proc)		12.4	6.5	3.9
% Ideal Speedup (16 proc)		86%	51%	42%
Performance vs. 32B lines	128B	1.37	1.10	1.21
	512B	1.39	0.84	0.85
Cache Miss Rate (% of refs)	32B	1.32	4.25	3.80
	128B	0.44	6.54	2.51
	512B	0.23	10.61	4.67
False Sh. Misses (% of refs)	32B	0.03	0.66	0.35
	128B	0.04	4.28	0.70
	512B	0.07	9.46	2.93
True Sh. Misses (% of refs)	32B	0.41	1.93	2.31
	128B	0.16	0.70	1.20
	512B	0.08	0.39	0.86
Cache Line Util.	32B	87%	88%	83%
	128B	70%	70%	58%
	512B	46%	44%	40%

Table 4: Summary of memory system behavior.

7 Characteristics of Compiler-Parallelized Applications

We have seen that SUIF applications with good speedup and memory behavior tend to have coarse granularity and large data sets. Here we describe in greater detail properties of the applications with poor memory behavior. Our simulator allows us to pinpoint parallel loops in the program that cause false and true sharing misses. We examine these loops, show how these problems are related to granularity and data set size, and discuss how new compiler technology can improve these programs' memory behavior.

7.1 Causes of False Sharing

Five of the programs, simple, su2cor, appsp, tomcatv and hydro2d have false sharing misses that account for more than 0.48% of all references for the 128 byte cache line configuration. Most notably, 16.1% of references in simple are false sharing misses. The primary cause of false sharing in these programs can be attributed to the following pattern of a parallel loop enclosing assignments to contiguous array elements:

```
DOALL I = 1, N
   A(I) = ...
```

Almost all such loops were innermost loops. The predominant false sharing problem arises when N/P, the number of iterations divided by the number of processors, is fewer than the number

of elements that fit on a cache line. Under this scenario, a processor may share a cache line with two or more processors. Though this example appears quite simple, it occurs frequently in practice; we found it to be the major cause of false sharing misses in simple, appsp and hydro2d. Note that this same pattern that has poor spatial locality on a multiprocessor would exhibit excellent spatial locality in a uniprocessor setting.

In tomcatv and su2cor, the same pattern occurs, but N/P is greater than or equal to the number of elements on a cache line. In this case, false sharing arises if the number of elements accessed by a processor is not a multiple of the number of elements on a cache line, or the array elements accessed are not aligned at a cache line boundary. When the loop strides through the data in this way, a processor may share the first and last cache lines it is using with at most one other processor; the other cache lines it uses are not shared. In all these programs, the loop bounds are fairly small (at most 512), and are a function of the data set size.

7.2 Causes of True Sharing

For SUIF compiler-parallelized applications, false sharing misses can occur within a single parallel loop. True sharing misses, in contrast, usually occur when a memory location written in one parallel loop is accessed in a subsequent parallel loop. The frequency of true sharing misses thus decreases for programs with coarse-grained parallelism, since the computation advances between parallel loops less frequently.

Note that some implementations of parallel reductions, operations that perform a sum, product, minimum or maximum over a series of data elements, could potentially lead to significant true sharing misses. These misses might result because one processor updates a memory location used by another processor, guarded by synchronization. However, reductions were not a significant cause of true sharing in our experiment. This result is largely due to SUIF's implementation of reductions, whereby the per-processor reduction computation is performed on private data followed by synchronized global accumulation of the private copies. Thus, the reduced memory locations are only globally updated at most once per processor rather than every time they are accessed.

For four of the programs, hydro2d, appsp, simple and tomcatv, true sharing misses account for more than 0.9% of all references. In particular, about 2.8% of the references in hydro2d are true sharing misses. We found that these misses are caused by parallelizing loops containing stencil patterns for small arrays. The pattern for such loops in hydro2d is the following:

```
DOALL J = 1, N
   DO I = 1, M
      A(I,J) = ...

DOALL J = 1, N
   DO I = 1, M
      B(I,J) = A(I,J-1) + A(I,J+1)
```

The first loop nest computes values of A while the second loop nest consumes those values. If the value of N is too small, each processor accesses large numbers of nonlocal elements of A in the second loop nest. In hydro2d the value of N can be as low as 20, causing significant true sharing for 16 processors.

Application	Data Set Size	Parallel Coverage	Parallel Gran.	16 Proc. Speedup	MCPI	Cache Miss Rate (% of refs) All Misses	False Sharing	True Sharing	Cache Line Util.
Effect of Array Privatization and Interprocedural Analysis									
appbt-naive	12^3	84%	0.12×10^6	0.3	6.0	5.2	1.4	3.1	76%
appbt	12^3	100%	5.73×10^6	6.4	0.4	0.7	0.1	0.5	73%
appsp-naive	12^3	91%	0.15×10^6	0.6	4.9	6.5	1.8	3.7	61%
appsp	12^3	98%	0.30×10^6	2.9	1.4	2.5	0.8	1.0	60%
Effect of Data Set Size									
su2cor-small	$6^3 \times 12$	97%	0.03×10^6	2.2	3.2	8.2	5.2	1.1	56%
su2cor	$8^3 \times 16$	99%	0.05×10^6	5.9	1.2	2.2	1.2	0.3	52%

Table 5: Effect of array privatization, interprocedural analysis, and data set size on memory behavior.

7.3 Effect of Granularity

It is well accepted that parallelizing compilers should find the largest parallel loops possible, since it increases the amount of parallel computation and reduces synchronization costs. What our study shows is that finding coarse-grain parallelism can also have a very beneficial effect on memory system behavior.

The SUIF compiler employs two techniques that enable detection of more outer parallel loops than current commercial compilers. First, *array privatization* locates arrays used as temporary storage within a loop. By creating private copies of the array for each parallel process, storage-related dependences associated with these arrays are eliminated. Private data, similar to the discussion of reductions above, are thus not involved in either true or false sharing misses. Second, all of the parallelization analyses in the SUIF compiler are performed *interprocedurally*, so that procedure boundaries do not affect the system's ability to locate parallel loops. The combination of these techniques enables the compiler to parallelize outer loops containing over a thousand lines of code in some cases. A more detailed discussion of the implementation of these techniques can be found in [11, 12].

To illustrate how these techniques can impact memory behavior, consider the performance of two SUIF applications, appbt and appsp. Table 5 displays their performance for the baseline memory architecture. appbt and appsp are the SUIF parallelized output that have been used throughout this paper, while appbt-naive and appsp-naive represent versions of the programs compiled without array privatization and interprocedural analysis. Parallel granularity increases with advanced analysis. Although parallel coverage is high for both versions of each program, memory behavior can be significantly different.

For the programs parallelized by SUIF with array privatization and interprocedural analysis, MCPI are lower, false and true sharing misses comprise a smaller fraction of all references, and cache line utilization increases. For appbt-naive, the impact of memory system effects overwhelms any improvements due to parallelization, causing the program to run one-third sequential speed on 16 processors. The impact on the memory system is almost as dramatic for appsp. These results show that advanced compilation technology aimed at detecting coarse-grain parallelism can be critical for improving memory system behavior.

7.4 Effect of Data Set Size

We also observe that for many SUIF programs, the data set size directly affects the parallelism granularity and hence memory behavior. As we showed earlier, false and true sharing misses are often caused by parallel loops with few iterations. For many SUIF applications, the number of iterations in these parallel loops depends on the data set size. For example, consider su2cor, one of the programs with moderate levels of false sharing. Table 5 presents results for our baseline memory system; it shows that when the data set size is reduced by a factor of three, memory behavior degrades drastically. The miss rate, and false and true sharing misses increase by a factor of two or more, while cache line utilization and speedups are halved. These results show that it is important to use realistic data set sizes when studying memory system behavior.

8 Related Work

Our work is unique in providing a detailed characterization of the memory behavior of compiler-parallelized codes. In addition, we have highlighted some of the constructs expected to be common in these codes and explained how they affect caching performance. By contrast, previous work has focused almost exclusively on characterizing memory system behavior of hand-parallelized applications on different styles of cache coherent multiprocessors. While our work draws on a significant body of related work in understanding multiprocessor memory behavior, we outline below the most directly relevant studies.

Eggers and Katz [7] did important early work characterizing application caching behavior of hand-parallelized programs in bus-based multiprocessors. For their applications, they show that the majority of cache misses in a bus-based multiprocessor are due to sharing misses. They also demonstrate that the overall miss rate in a multiprocessor can increase as the cache line size increases, whereas it tends to go down in uniprocessors. Bolosky and Scott [3] developed the cost component method to measure false sharing and applied it to four computation kernels. More recently, Dubois *et al.* [5] introduced a definition of false sharing and used it to measure four hand-parallelized applications. We use their definition for our study.

Torrellas *et al.* [23] measured false and true sharing and the number of bytes used per cache line. They find poor spatial locality has a greater impact than false sharing in determining the overall miss rate of their applications. In comparison, the SUIF applications in this study have excellent spatial locality and are limited mostly by false sharing. Both Torrellas *et al.* [23] and Eggers and Jeremiassen [6] suggest program transformations to eliminate false sharing in hand-parallelized programs. The latter have implemented their transformations in a compiler, and used them to eliminate false sharing in the SPLASH benchmarks by padding lock variables [13]. (In our SPLASH programs lock variables have also been padded to eliminate false sharing.)

Only a handful of researchers have looked at the behavior of compiler-parallelized applications. Blume and Eigenmann [2] analyzed the performance of commercial parallelizing compilers on the PERFECT benchmarks, concluding that they detected only limited amounts of parallelism. The SUIF compiler incorporates many of the analyses they deemed vital; as a result, it enjoys much better success in extracting parallelism.

More recently, Natarajan *et al.* [20] measured operating system, parallelism, and memory contention overhead for five PERFECT applications on the Cedar multiprocessor. They determined that parallelism overhead consumed 10–25% of program execution time and memory contention overhead was over 10%. Our study focused on a more advanced memory system and compiler; we also determine causes of poor memory behavior. Lilja [17] examines the impact

of prefetching in conjunction with loop scheduling strategies that schedule blocks of consecutive iterations to execute on each processor.

Finally, we note that there is much active research on compiler techniques to improve memory performance. Heuristics are being developed to reduce true sharing by improved co-location of data and computation [1, 3] and eliminate false sharing by better compiler management of large coherence units [9]. Our study helps to point out areas of poor program memory behavior deserving of additional research.

9 Conclusions

In this paper, we demonstrate that good memory system behavior is *vital* to achieving reasonable speedups on moderate-scale multiprocessors. We present the first detailed study of the impact of advanced memory systems on the performance of a large suite of compiler-parallelized codes running with their full-size data sets. Our results show applications amenable to compiler parallelization suffer from significantly higher memory costs than hand-parallelized codes, particularly for longer (e.g. 512 byte) cache lines. We discover that increases in granularity are frequently correlated with improvements in memory behavior and overall performance. We also identify compiler constructs that lead to frequent true and false sharing, and present case studies that quantify the positive impact of advanced compiler techniques such as interprocedural analysis and array privatization.

Overall, this study has several implications. For computer architects, our study shows a high degree of sharing is likely for compiler-parallelized applications running on advanced memory systems with long cache lines. For compiler writers, we discover small parallel loops to be the primary culprit in poor memory behavior; compilers need to be more careful in parallelizing small loops since sharing misses may outweigh any potential benefits from parallelism. For both architects and compiler writers, the potential impact of parallelism granularity on memory system behavior should be weighed carefully when making tradeoffs in system design.

10 Acknowledgments

We are grateful to Monica Lam and other members of the SUIF research group at Stanford for supplying the compiler-parallelized applications used in our study. We wish to thank Mori Ohara for providing some of our simulation technology. Finally, we are indebted to John Hennessy, Anoop Gupta, Monica Lam, and J.P. Singh for their helpful comments.

This research was supported in part by ARPA contract DABT63-94-C-0054, as well as two NSF CISE postdoctoral fellowships in Experimental Science. Margaret Martonosi is partly supported through an NSF Career Award (CCR-9502516). In addition, we acknowledge the Jet Propulsion Laboratory for a portion of Mary Hall's funding. Finally, many of the simulations were run at Princeton using machines purchased through an NSF CISE Research Infrastructure grant.

References

[1] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. SIGPLAN '93 Conf. on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[2] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[3] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, September 1993.

[4] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proc. International Conference on Parallel Processing*, August 1991.

[5] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, et al. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proc. 20th Intl. Symp. on Computer Architecture*, pages 88–97, May 1993.

[6] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proc. 1991 Int'l Conf. on Parallel Processing*, St. Charles, IL, August 1991.

[7] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Third Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 257–270, April 1989.

[8] Stephen R. Goldschmidt. *Simulation of Multiprocessors, Speed and Accuracy.* PhD thesis, Stanford University, June 1993.

[9] E. Granston and H. Wishoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proc. 1993 ACM Int'l. Conf. on Supercomputing*, Tokyo, Japan, July 1993.

[10] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessos. *IEEE Trans. on Computers*, 41(7):794–810, July 1992.

[11] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe. Interprocedural parallelization analysis: A case study. In *Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing*, San Francisco, February 1995.

[12] M.W. Hall, S.P. Amarasinghe, B.S. Murphy, S. Liao, and M. Lam. Interprocedural parallelization analysis. In *Proceedings of Supercomputing '95.* IEEE Press, December 1995.

[13] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[14] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Int'l Symp. on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[15] R. L. Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors.* PhD thesis, University of Illinois at Urbana-Champaign, May 1987.

[16] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Protocol for the DASH Multiprocessor. In *Proc. 17th Annual Int'l Symp. on Computer Architecture*, May 1990.

[17] D. J. Lilja. The Impact of Parallel Loop Scheduling Strategies on Prefetching in a Shared-Memory Multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 5(6):573–584, June 1994.

[18] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[19] Margaret R. Martonosi. *Analyzing and Tuning Memory Performance in Sequential and Parallel Programs.* PhD thesis, Stanford University, December 1993. Also Stanford CSL Technical Report CSL-TR-94-602.

[20] C. Natarajan, S. Sharma, and R. Iyer. Measurement-based characterization of global memory and network contention, operating system and parallelization overheads: Case study on a shared-memory multiprocessor. In *Proc. of the 21st Int'l Symp. on Computer Architecture*, Chicago, IL, May 1994.

[21] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. 21st Annual Int'l. Symp. on Computer Architecture*, pages 325–337, April 1994.

[22] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[23] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Trans. on Computers*, 43(6):651–63, June 1994.

[24] E. Torrie, C-W Tseng, M. Martonosi, and M. W. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 204–213, Limassol, Cyprus, June 1995.

[25] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[26] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proc. of the 22st Int'l Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.