Evaluating the Impact of Advanced Memory Systems on Compiler-Parallelized Codes

Evan Torrie* Chau-Wen Tseng* Margaret Martonosi† Mary W. Hall‡

torrie@cs.stanford.edu *tseng@cs.stanford.edu* *martonosi@ee.princeton.edu* *mhall@cs.caltech.edu*

**Computer Systems Laboratory, Stanford University, Stanford, CA 94305-4070*

† Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544-5263

‡ Dept. of Computer Science, California Institute of Technology, Pasadena, CA 91125

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 the SPEC, NAS, PERFECT, and RICEPS benchmark suites, we measure statistics such as speedups, memory costs, causes of cache misses, cache line utilization, and data traffic.

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 state-of-the-art 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 success opens parallel computing to a broad spectrum of users, since parallel programs can now 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, 15]. 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 [21, 24]. 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 the first detailed examination of the memory behavior of compiler-parallelized applications. 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 interprocedural parallelization and array privatization can increase parallelism granularity, improving memory system performance.

Overall, this study benefits 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.

This research was supported in part by ARPA contract DABT63-94-C-0054 and NSF CISE postdoctoral fellowships in Experimental Science.

To appear in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Limassol, Cyprus, June 1995.

Program	Suite	Description	Input Data Set	References Simulated (10^6)	Parallel Coverage (%)	Parallel Granularity (10^6 cycles)
COMPILER-PARALLELIZED BENCHMARKS						
appbt	NAS	block-tridiagonal PDEs	12^3 grid	48	100	5.73
appsp	NAS	scalar-pentadiagonal PDEs	12^3 grid	18	98	0.30
cgm	NAS	sparse conjugate gradient	1400 array elements	37	98	1.02
erle64	MISC	ADI integration	64^3 array elements	18	100	14.59
flo52	PERFECT	transonic inviscid flow	40x8 grid	92	99	0.23
hydro2d	SPEC	Navier-Stokes	102x40 grid	22	98	0.02
linpackd	RICEPS	Gaussian elimination	256^2 array elements	29	95	0.30
mgrid	NAS	multigrid solver	32^3 grid	74	94	3.05
ora	SPEC	ray tracing	650 array elements	31	100	213.92
simple	RICEPS	Lagrangian hydrodynamics	203x183 grid	62	94	2.36
su2cor	SPEC	quantum physics	8^3x16 grid	240	98	0.05
swm256	SPEC	shallow water model	256^2 grid	44	100	5.34
tomcatv	SPEC	mesh generation	256^2 grid	44	100	4.88
HAND-PARALLELIZED BENCHMARKS						
barnes	SPLASH	hierarchical n-body problem	8192 particles	598	100	
cholesky	SPLASH	sparse block cholesky	3968^2 (tk15.O)	281	100	
ocean	SPLASH	ocean model simulation	258^2 grid	117	100	
water	SPLASH	molecular dynamics	512 molecules	314	100	

Table 1: Characteristics of scientific applications in study.

In the following sections, we describe the compiler, applications, and simulation methodology 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 [23] 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. SUIF contains most of the features found in commercial parallelizing compilers such as KAP; additionally, it performs both array privatization and interprocedural analysis. Section 7.3 will demonstrate the importance of these features.

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. This simple static scheduling heuristic maintains spatial 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 [15], SGI Challenge, and KSR-1 multiprocessors.

3 Parallel Applications

The applications used in our study consist of standard scientific codes taken from the SPEC, NAS, PERFECT, and RICEPS benchmark suites; their characteristics are listed in Table 1. 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. 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. We measured the size of each parallel loop as cycles per invocation using pixie, 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 1. As we will show in Section 6.4, for many applications the compiler's ability to exploit larger granularities of parallelism is correlated to good memory performance.

In order to provide a basis for comparison, we also include in our study hand-parallelized programs from the SPLASH benchmarks [21, 24]. Their granularity was not computed because they use different synchronization mechanisms. For brevity, in the remainder of the paper we shall refer to these two groups of programs as the SUIF and SPLASH applications. Except for linpackd, each program uses the standard data set provided, avoiding poor memory behavior caused by unrealistically small problem sizes. Where necessary we have reduced the number of time steps in each application to limit simulation time. To compensate for this, we reset statistics after initialization and cold start to avoid skewing results.

4 Experimental Methodology

For these experiments, we used an extended version of the MemSpy simulator [17, 18] and the TangoLite simulation and tracing system [4, 8]. TangoLite allows simulation of parallel programs by multiplexing their execution on a uniprocessor workstation. To fully capture potential sharing between processors, we interleave threads after 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.

4.1 Memory Systems Simulated

For our study we simulated two basic memory systems. First, we used an "ideal" memory system with 1-cycle memory access latencies to demonstrate potential application performance. Speedups

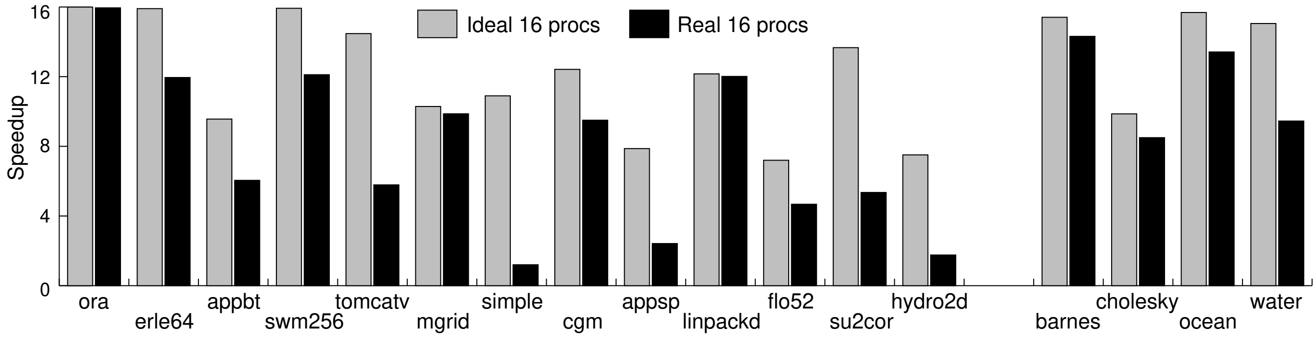

Figure 1: Application speedups for ideal and real memory systems.

on the ideal memory system are limited only by the amount of parallelism discovered by the compiler and load imbalance in the parallel code.

The bulk of our results are presented for 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 [13, 15]. We choose a 200 MHz processor, a 100 MHz 256-bit local memory bus, and a 200 MHz 16-bit wide mesh network interconnect. Each processor has a single level LRU cache whose size, associativity, and line size we vary. The penalty for a cache miss is dependent on the line size; Table 2 shows the average penalties assuming no contention on a 16 processor system.

Line size	Local miss	Remote clean	Dirty remote
32 bytes	82	288	406
64 bytes	84	306	444
128 bytes	88	342	484
256 bytes	96	414	564
512 bytes	112	558	724

Table 2: Cache miss penalties (cycles) vs. line size.

We model contention for the local memory bus and memory ports but not for the network, since we believe network traffic is not a limiting factor. A round-robin page allocation policy is employed to reduce contention. The memory system is write-buffered. Barrier synchronization requires a round-trip on the mesh network; we assume that barriers bypass memory and that there is only light contention for the barrier.

For our study, we measure memory system performance over a range of cache line lengths (32, 64, 128, 256, 512 bytes), set associativities (direct-mapped, 4-way, and fully-associative), and cache sizes (32K, 128K, and 512K bytes). Due to space limitations, we choose to present results mostly for a *baseline* memory system with a 128KB, 128 byte line 4-way set-associative cache. The cache parameters were selected to model a forward-looking multiprocessor memory hierarchy.

5 Overview of Application Behavior

We begin by presenting an overview of application performance (*i.e.,* simulated wall clock time) to motivate our paper. Figure 1 shows speedups for 16 processor simulations on two different memory models, with each speedup being calculated relative to a uniprocessor run on the same memory model. The SUIF applications are sorted with respect to their granularity, with the largest granularity (ora) leftmost. SPLASH programs are displayed on the right. (As will be shown in Section 6.4, there is a strong correlation between granularity and good memory behavior.)

The models approximate (i) an ideal memory system and (ii) a more realistic NUMA memory system, as described in Section 4. For the ideal memory system, SUIF applications achieve average speedups of 11.8 on 16 processors while SPLASH applications realized speedups of 14.0 on 16 processors. These results clearly demonstrate 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 range from 1.8 to 16.0 with an average speedup of 8.4. In comparison, SPLASH applications maintained average speedups of 11.4 on 16 processors. (These simulated speedups correspond well with actual speedups observed for these programs on the DASH and SGI Challenge multiprocessors [11, 23].)

These performance results suggest that memory system and synchronization costs are causing a significant drop in performance particularly as the number of processors increases. To quantify the effect of the memory system, we measured the *miss cycles per instruction* (MCPI) directly. For our baseline memory system, average MCPI for SUIF applications jumps from 0.72 to 2.65 going from 1 to 16 processors. Since most instructions execute in one cycle, this result means that SUIF applications spend over twice as much time on memory accesses as on useful computation! In comparison, average MCPI for SPLASH applications only increases from 0.52 to 0.63 going from 1 to 16 processors. Clearly memory system behavior significantly affects the performance of the SUIF applications and deserves closer examination.

6 Memory System Behavior

In this section, we examine the behavior of the cache and memory system as we vary three cache parameters, *line size*, *set associativity*, and *cache size*. As we shall see in Sections 6.2 and 6.3, for most of these applications, the important working set fits in realistic caches and thus set associativity and cache size do not affect performance as much as cache line size for these input data sets. Hence, we focus mainly on memory system sensitivity to cache line size.

6.1 Effect of Cache Line Size

As processor speeds continue to increase faster than memory speeds, there has been a corresponding trend towards increasing the cache line size. This increase takes 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.

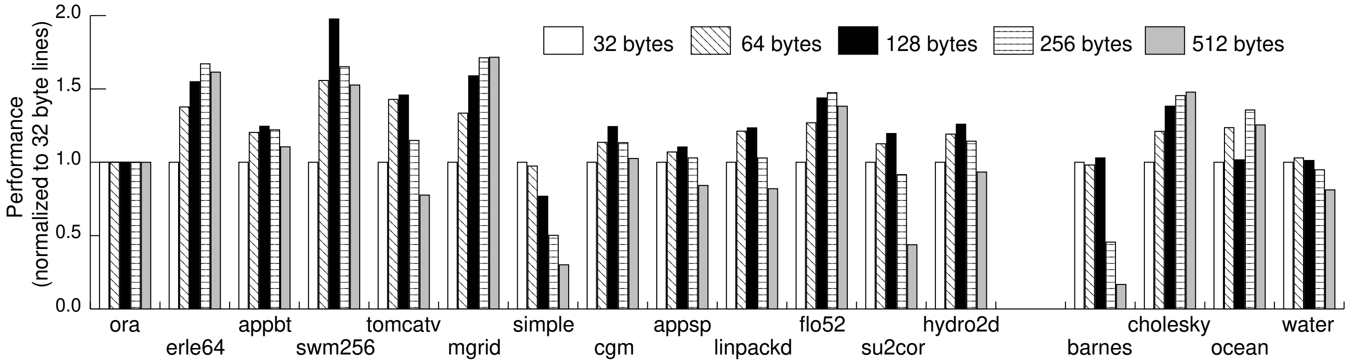

Figure 2: Relative application performance vs. cache line size.

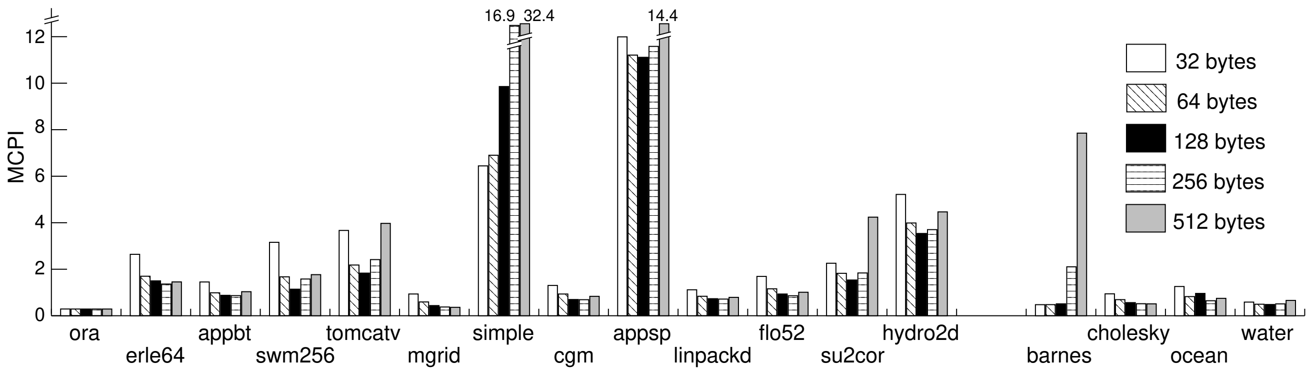

Figure 3: Miss cycles per instruction (MCPI) vs. cache line size.

Unfortunately, miss rates are not so predictable for multiprocessor caches [14, 22]. 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.

Throughout this section, we present results for cache line sizes of 32, 64, 128, 256 and 512 bytes, but because of space limitations, we concentrate on the 32, 128 and 512 byte results.

6.1.1 Performance

Our experiments show that for one processor, 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. Remember that we define performance as the simulated wall clock time. However, we find their multiprocessor performance does not uniformly improve as cache lines grow longer. Figure 2 displays the change in performance for 16 processor executions as cache line size varies. The height of each bar is determined by the ratio of wall clock time to 32 byte cache lines, with values above one indicating improvements. Results shows that 11 out of the 13 SUIF applications benefit when increasing the cache line size from 32 to 128 bytes, with an average 31% improvement. However, in sharp contrast to uniprocessor results, SUIF application performance on average degrades by 19% going from 128 to 512 byte cache lines. Three programs (simple, su2cor, tomcatv) worsen significantly.

We discover a similar result when looking at the number of miss cycles per instruction (MCPI). Figure 3 displays the MCPI of 16 processor executions for different cache line sizes. Increasing the cache line size from 32 to 128 bytes reduces MCPI for all SUIF programs except simple. Average MCPI *decreases* 18%, going from 3.24 to 2.65. In comparison, going from 128 to 512 byte cache lines *increases* MCPI for 10 of 13 programs, with average MCPI jumping back up to 5.16. SPLASH programs fared similarly, with average MCPI of 0.82, 0.63, and 2.44 for cache line lengths of 32, 128, and 512 bytes, respectively. It thus appears that for the given data sets and cache configurations, both SUIF and SPLASH programs were able to exploit 128 but not 512 byte lines.

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 2.65 indicates over twice as many cycles are spent on memory accesses as on useful computation. For 512 byte cache lines SUIF applications spend over five 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.

It is also important to point out that the compiler-parallelized SUIF applications experience worse memory behavior than the hand-parallelized SPLASH applications. For 128 byte cache lines, the average MCPI of SUIF applications is 4.2 times greater than the average of the SPLASH suite.

Though suggestive, these high-level memory performance numbers do not really tell the whole story. Ideally we would like to understand *why* these applications are not amenable to very long cache lines when executing in parallel. That will allow us to understand the application, compiler, and architectural characteristics that are needed to support efficient execution with long fetch and coherence units. We examine several factors, beginning with the rate of cache misses and their causes.

6.1.2 Cache Miss Rate

Table 3 presents average cache miss rates for a four-way set associative 128KB cache. They show that for uniprocessors SUIF programs have sufficient spatial locality to reduce cache misses with lines up to 512 bytes. However, with 16 processors these applications cannot take advantage of 512 byte lines (for these data sets and cache organizations). Hand-parallelized applications experience degradation on one processor with 512 byte lines due to a 120% jump in replacement misses for ocean, but otherwise exhibit behavior similar to SUIF applications.

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 four possible types of cache misses: (i) *replacement* misses, (ii) *cold* (or compulsory) misses, (iii) misses due to *true* sharing, and (iv) misses due to *false* sharing.

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%. For the four SUIF programs (erle64, swm256, appsp and to a lesser degree tomcatv) which are dominated by a large number of replacement misses, appsp has the poorest spatial locality with the replacement miss staying constant or increasing as the line size is increased. Only two SUIF applications, cgm and ora have close to 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 9 of 13 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. 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. While true sharing miss rates decrease as line size increases, the decrease is not as dramatic. 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.

6.1.3 Cache Line Utilization and Data Traffic

Torrellas *et al.* have looked at the number of words actually touched in a cache line as a measure of the spatial locality exploited by long cache lines [22]. We abstract this measure with a metric that we call *utilization*: the percentage of bytes in a cache line actually referenced by a processor from the initial cache miss on a line, until the time that line is evicted from the cache (either due to replacement, invalidation, or end of the program.)

Programs	Miss Type	Cache Miss Rate (vs. line size) 32B	128B	512B	Change 32B →128B	128B →512B
SUIF: 1 Proc	All	4.44%	1.88%	1.30%	–58%	–31%
SUIF: 16 Procs.	All	3.18%	2.16%	3.49%	–32%	+62%
	False	0.26%	0.55%	2.19%	+112%	+298%
	True	1.03%	0.52%	0.34%	–50%	–35%
	Cold	0.21%	0.07%	0.03%	–67%	–71%
	Repl.	1.68%	1.02%	0.93%	–39%	–9%
SPLASH: 1 Proc	All	1.46%	0.57%	0.93%	–61%	+63%
SPLASH: 16 Proc.	All	0.70%	0.32%	0.68%	–54%	+113%
	False	0.00%	0.01%	0.48%	+802%	+4200%
	True	0.07%	0.04%	0.03%	–41%	–20%
	Cold	0.08%	0.03%	0.01%	–68%	–68%
	Repl.	0.55%	0.24%	0.16%	–56%	–35%

Table 3: Average cache miss rates.

Figure 5 shows measured per-application utilization rates for different cache line sizes. It is apparent that utilization rates decrease as cache line sizes increase, sometimes very quickly. The average utilization for SUIF programs drops 36% going from 32 to 128 byte lines and falls an additional 67% going from 128 to 512 byte lines. Utilization for SPLASH applications drops similarly. The low utilization results indicate much of the data fetched into cache for longer cache lines is unused, raising concerns about how data traffic behaves with increasing line sizes. Overall, our measurements indicated that SUIF applications averaged 115% and 412% increases in data traffic per instruction for these two increases in line size.

Figure 6 displays data traffic for each program (in bytes per instruction, to adjust for varying run times of the applications). Building on the definition of utilization, we can define a notion of *useful data traffic*. If data traffic is considered to be the number of bytes transferred into the cache on cache misses, then useful data traffic is the number of these bytes that actually get referenced before the cache line is evicted. In general, one hopes that useful data traffic is a roughly invariant indicator of application data requirements and caching effectiveness. In practice, however, useful data traffic increases slightly with line size. If a word is referenced multiple times but the line is invalidated before all accesses are complete, these additional cache misses all count as useful traffic. Hence applications which have high false sharing miss rates also have corresponding larger increases in useful traffic (e.g. simple, su2cor and tomcatv).

We found *unused* data traffic to be a function of both false sharing rates and spatial locality. Applications with significant sharing misses and poor locality in those misses see large increases in unused data traffic with increases in line size. Over the SUIF suite, unused data traffic increased an average of 5.0 times when moving from 32 to 128 byte lines and 10.1 times when going from 128 to 512 byte line sizes.

6.1.4 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 results in Figure 7 show that invalidated cache lines typically exhibit much lower utilization (the exception, appsp, is due to pathological conflicts in the four-way associative cache). The intuitive explanation is that if lines are evicted by invalidation,

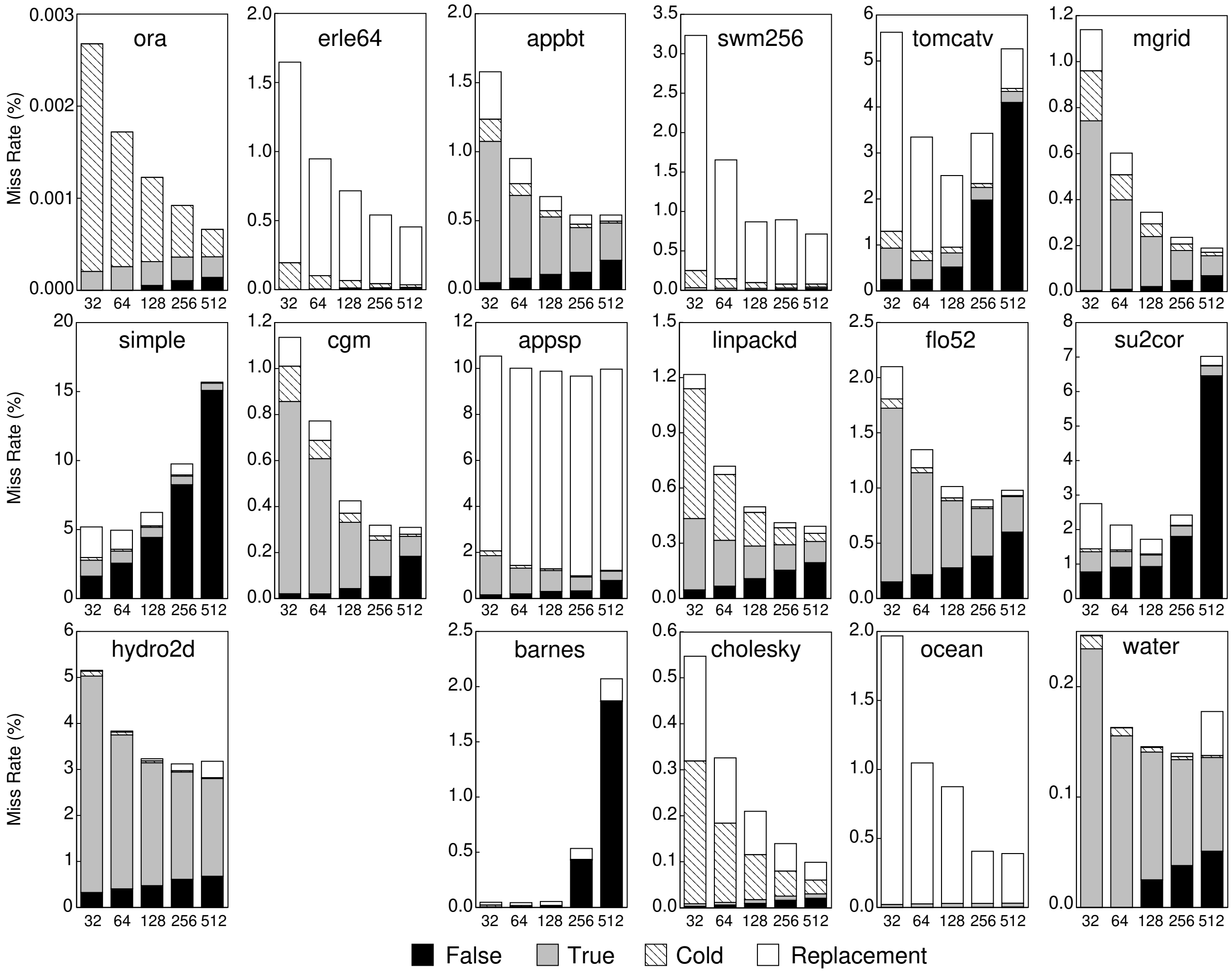

Figure 4: Cache misses vs. cache line size. Each box illustrates cache miss rate versus line size for a different application. The SPLASH applications are the last four applications in the lower right corner.

either true or false sharing may have led to the line being prematurely evicted from the cache.

This observation is important, because the bimodal behavior suggests that special optimizations for each type of behavior may be possible. In systems allowing flexible protocols (such as Tempest [20]), 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.

6.2 Effect of Cache Set Associativity

Up to this point we have considered a 4-way set-associative cache. To understand the effects of associativity, we also measured performance with direct-mapped and fully-associative caches. We found that performance improved between 10% and 50% for most programs when increasing associativity from direct-mapped to four-way caches. Smaller gains result going from four-way to fully-associative caches. Three applications, appsp, erle64, and su2cor demonstrated the greatest gains; they all possess multiple large data arrays that conflict in a 128KB cache. For swm256, performance is better for a four-way rather than fully-associative cache. The access pattern of swm256 is such that there are 30% more replacement misses in the fully-associative cache, resulting in poorer performance. Overall, we found 4-way and fully-associative caches generally yielded similar results.

6.3 Effect of Cache Size

As with associativity, until this point we have primarily focused on a single cache size of 128K bytes. When measuring 32KB and 512KB caches as well, we find that only two SUIF programs, erle64 and swm256, possess sufficiently large working sets to benefit from caches larger than 128KB. For tomcatv, overall cache miss rates decrease by 6% but performance also decreases as cache size increases from 128KB to 512KB. Examining its behavior, we found the ratio of remote to local misses increases by a factor of four and the number of false and true sharing misses increases by a factor of two. These longer latencies and higher contention increase average memory cost and reduce performance.

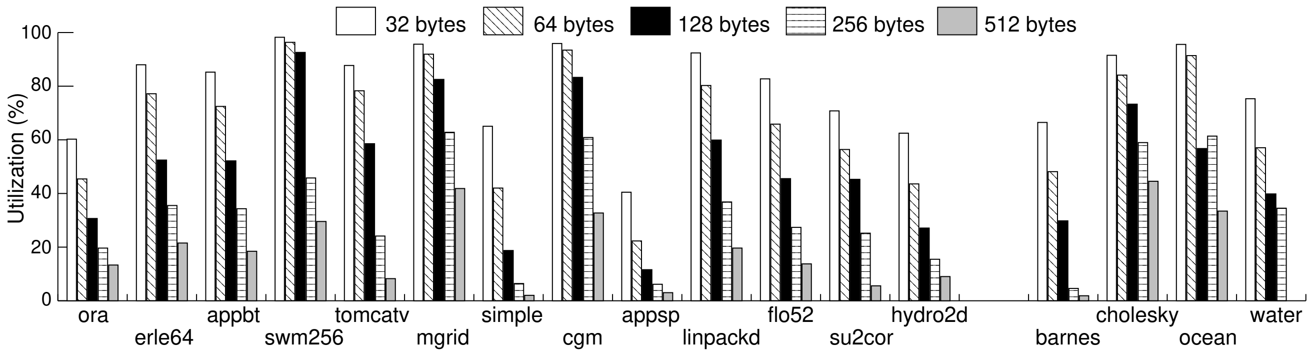

Figure 5: Fraction of cache line utilized vs. cache line size.

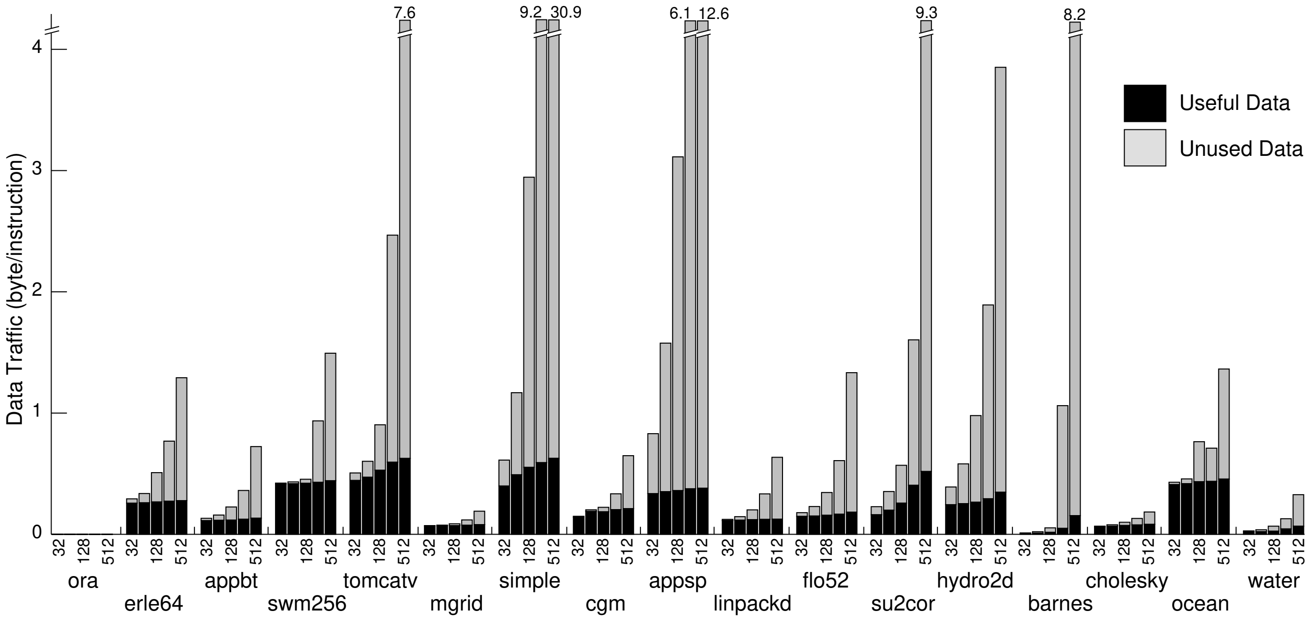

Figure 6: Data traffic vs. cache line size.

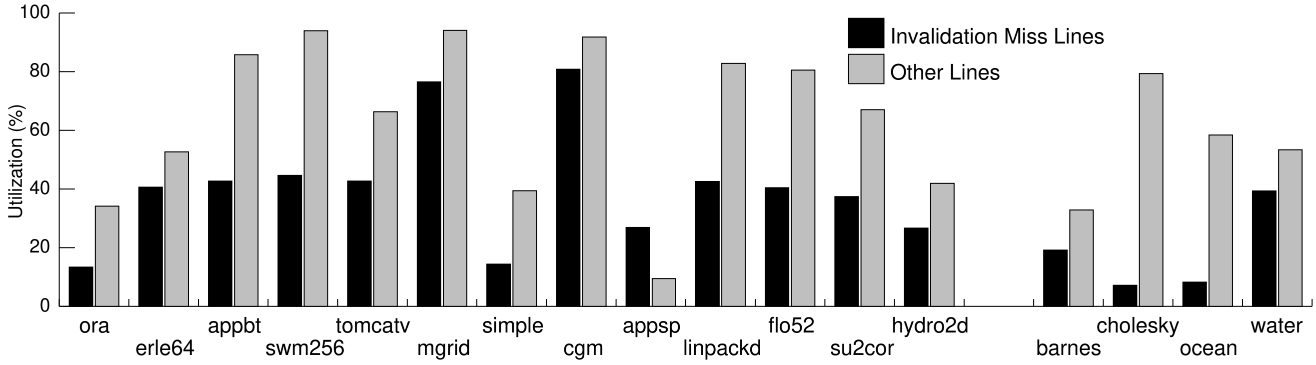

Figure 7: Utilization of invalidated and non-invalidated lines.

6.4 Summary

This section has provided a baseline characterization of the memory behavior of SUIF and SPLASH applications running on multiprocessors. It is important to note that except for linpackd, we are using *full-sized* data sets. 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 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.

Unfortunately, we also find that that even coarse-grain SUIF applications have relatively high memory system costs when compared to hand-tuned SPLASH applications. To focus on the causes of higher memory system cost in greater detail, the following section examines particular application and compiler characteristics that have a significant impact on memory system behavior.

Metric		SUIF Larger Grain	SUIF Med. Grain	SUIF Smaller Grain	SPLASH
Granularity (10^6 cycles)		59.9	2.8	0.2	
Parallel Coverage		99.9%	96.5%	95.0%	100%
Ideal Speedup (16 proc)		13.8	12.8	9.7	14.0
Sim'd. Speedup (16 proc)		11.3	9.8	5.2	11.4
% Ideal Speedup (16 proc)		82%	77%	54%	81%
Performance vs. 32B lines	128B	1.42	1.29	1.24	1.11
	512B	1.10	1.16	0.93	0.93
Cache Miss Rate (% of refs)	32B	2.61	2.27	4.35	0.70
	128B	1.01	1.93	3.27	0.32
	512B	1.63	4.33	4.31	0.68
False Sh. Misses (% of refs)	32B	0.07	0.41	0.29	0.00
	128B	0.16	1.12	0.42	0.01
	512B	1.08	3.83	1.74	0.49
True Sh. Misses (% of refs)	32B	0.44	0.69	1.79	0.07
	128B	0.19	0.31	0.94	0.04
	512B	0.13	0.18	0.65	0.03
Cache Line Util.	32B	83%	86%	70%	82%
	128B	57%	61%	38%	50%
	512B	21%	21%	10%	25%

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

Four of the programs, simple, su2cor, tomcatv and hydro2d have false sharing misses that account for more than 0.45% of all references for the 128 byte cache line configuration. Most notably, 4.4% 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 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.

For four of the programs, hydro2d, appsp, flo52 and simple, true sharing misses account for more than 0.5% of all references. In particular, about 2.7% 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 Granularity	16 Proc. Speedup	MCPI	Cache Miss Rate (as % of all references) All Misses	False Sharing	True Sharing	Cache Line Utilization
EFFECT OF ARRAY PRIVATIZATION AND INTERPROCEDURAL ANALYSIS									
appbt-naive	12^3	84%	0.12×10^6	0.33	8.62	4.76	1.39	2.83	25.6%
appbt	12^3	100%	5.73×10^6	6.04	0.88	0.67	0.11	0.41	52.2%
flo52-naive	40×8	88%	0.09×10^6	3.96	1.28	1.65	0.68	0.78	34.9%
flo52	40×8	99%	0.22×10^6	4.67	0.94	1.01	0.27	0.60	45.5%
EFFECT OF DATA SET SIZE									
su2cor-small	$6^3 \times 12$	97%	0.03×10^6	1.98	3.78	5.74	4.22	0.89	21.3%
su2cor	$8^3 \times 16$	99%	0.05×10^6	5.35	1.54	1.72	0.93	0.34	45.3%

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. 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].

To illustrate how these techniques can impact memory behavior, consider the performance of two SUIF applications, appbt and flo52. Table 5 displays their performance for the baseline memory architecture. appbt and flo52 are the SUIF parallelized output that have been used throughout this paper, while appbt-naive and flo52-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 as fast on 16 processors. The results are less dramatic for flo52, but the improvement in memory system performance is noticable. 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.* [22] 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.* [22] 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 [12]. (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.* [19] measured operating system, parallelism, and memory contention overhead for five PER-

FECT 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 [16] 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 Acknowledgements

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 Steve Woo and J.P. Singh for providing SPLASH applications and some of the simulation technology; Mark Heinrich also assisted us with the details of TangoLite. Finally, we are indebted to John Hennessy, Anoop Gupta, Monica Lam, and J.P. Singh for their helpful comments.

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, Nov. 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, Sept. 1993.

[4] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proc. International Conference on Parallel Processing*, Aug. 1991.

[5] M. Dubois, J. Skeppstedt, L. 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, Aug. 1991.

[7] S. J. Eggers and R. 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, Apr. 1989.

[8] S. 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] A. Gupta and W.-D. 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, Feb. 1995.

[12] 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.

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

[14] 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.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Protocol for the DASH Multiprocessor. In *Proc. 17th Annual Int'l Symp. on Computer Architecture*, May 1990.

[16] 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.

[17] M. Martonosi, A. Gupta, and T. 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.

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

[19] 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.

[20] 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, Apr. 1994.

[21] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

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

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

[24] S. Woo, M. Ohara, E. Torrie, J. 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.