False Sharing and Spatial Locality in Multiprocessor Caches

Josep Torrellas, Monica S. Lam, and John L. Hennessy

Computer Systems Laboratory

Stanford University, CA 94305

Abstract

The performance of the data cache in shared-memory multiprocessors has been shown to be different from that in uniprocessors. In particular, cache miss rates in multiprocessors do not show the sharp drop typical of uniprocessors when the size of the cache block increases. The resulting high cache miss rate is a cause of concern, since it can significantly limit the performance of multiprocessors. Some researchers have speculated that this effect is due to false sharing, the coherence transactions that result when different processors update different words of the same cache block in an interleaved fashion. While the analysis of six applications in this paper confirms that false sharing has a significant impact on the miss rate, the measurements also show that poor spatial locality among accesses to shared data has an even larger impact. To mitigate false sharing and to enhance spatial locality, we optimize the layout of shared data in cache blocks in a programmer-transparent manner. We show that this approach can reduce the number of misses on shared data by about 10% on average.

1 Introduction

Scalable machines that support a shared-memory paradigm are a promising way of attaining the benefits of large-scale multiprocessing without surrendering programmability [1, 2, 3, 4, 5, 6]. An interesting subclass of these machines is the class that provides hardware cache coherence, which makes programming easier, while reducing storage access latencies by caching shared data. While these machines can do well on problems with low levels of data sharing, it is unclear how well caches will perform when accesses to shared data occur frequently.

The cache performance of shared data is the subject of intense ongoing research. Agarwal and Gupta [7] study locality issues in traces of memory references from a four-processor machine and report a high degree of processor interleaving in the accesses to a given shared-memory location. This suggests that shared data can be the source of frequent misses. Indeed, Eggers and Katz [8], in a simulation of 5 to 12 processors in a bus, show that shared data is responsible for the majority of cache misses and bus cycles. In addition, they show that the miss rate of the data cache in multiprocessors changes less predictably than in uniprocessors with increasing cache block size. While the miss rate in uniprocessors tends to go down with increasing cache block size, the miss rate in multiprocessors can go down or up with larger block sizes. A further understanding of the patterns of data sharing is provided by Weber and Gupta [9], who show that write-shared variables are usually invalidated from caches before being replicated in more than a few different caches. Finally, in another example of unusual behavior, Lee et al. [10] find that the optimal cache block size for data is one or two words long, in contrast to the larger sizes used in uniprocessors [11]. Clearly, given the performance impact of the cache behavior of shared data, a deeper understanding of it is necessary.

In this paper, we focus on one parameter that has a major effect on the cache performance of shared data, namely the size of the cache blocks. A second issue that motivates the interest in this topic is that the measurements obtained so far on the impact of the block size on the miss rate of shared data show such wide variation [8] that they are difficult to generalize. In this paper, we explain the effect of the cache block size on the miss rate as a combination of two well-behaved components: false sharing and spatial locality. False sharing, in its simplest form, occurs when two processors repeatedly write to two different words of the same cache block in an interleaved fashion. This causes the cache block to bounce back and forth between the two caches as if the contents of the block were truly being shared. False sharing usually increases with the block size and tends to drive miss rates up with increasing block size. The second component, spatial locality in the data [12], is the property that indicates that the probability of an access to a given memory word is high if neighboring words have been recently accessed. This well-known property produces the opposite effect from false sharing — a reduction in the miss rate as the block size increases.

We assess the contribution of each component by using a model of sharing where individual misses are classified as false sharing misses or as true sharing misses. The latter are due to the interprocessor communication intrinsic to the application. False sharing misses measure false sharing. The effectiveness of increasing the cache block size in eliminating true sharing misses measures the degree of spatial locality present. Experimental measurements show that poor spatial locality in shared data has a larger effect than false sharing in determining the overall miss rate.

To reduce the number of cache misses due to poor spatial locality and false sharing, we propose optimizations that require no programmer help and can therefore be implemented by the compiler. Further, we do not consider techniques that require changes to the assignment of computation to processors, as in loop interchange or loop tiling [13, 14], since they are only feasible in highly regular codes. Instead, we propose simple, local techniques that optimize the layout of shared data at the cache block level. These techniques are effective enough to eliminate, on average, about 10% of the misses on shared data in the applications.

This paper is organized as follows. Section 2 discusses the methodology and characteristics of the application set used throughout the study. Section 3 presents a model of data sharing. This model is used in Section 4 to analyze experimental data on cache miss rates and processor-memory traffic. Based on this analysis, we propose and evaluate optimizations to improve data caching in Section 5. Finally, in Section 6, detailed simulations of an existing architecture examine the performance impact of the issues raised in the previous sections.

2 Methodology and Application Set Characteristics

The results reported in this paper are based on simulations driven by traces of parallel applications. The applications are compiled with a conventional optimizing compiler. This section describes the characteristics of the applications used, presents the simulator models, and evaluates the effect of conventional code optimizations on the frequency of data sharing.

2.1 Application Set and Trace Characteristics

The parallel applications studied represent a variety of engineering algorithms [15, 16, 17, 18, 19, 20] (Table 1). Csim, Mp3d, and LocusRoute are research tools with between 1000 and 6000 lines of code. The other three applications, namely DWF, Maxflow, and Mincut implement several commonly used parallel algorithms and are less than 1000 lines of code each. Each application uses the synchronization and sharing primitives provided by the Argonne National Laboratory macro package [21]. The synchronization primitives are locks, barriers, and distributed loop control variables. The applications are in C and written so that they can run on any number of processors. We use code compiled with standard code optimizations.

Table 1: Application set characteristics. The third column lists the size of the data structures declared shared.

Application	Description	Shared Data Space (Mbytes)
Csim	Chandy-Misra logic gate simulator.	2.83
DWF	Performs string pattern matching.	2.10
Mp3d	3-D particle simulator used in aeronautics.	1.85
LocusRoute	Global router for VLSI standard cells.	1.84
Maxflow	Determines the maximum flow in a directed graph.	0.26
Mincut	Partitions a graph using simulated annealing.	0.01

We trace the applications using Tango [22], a tracing program that simulates a multiprocessor. The traces correspond to 16 and 32 processor runs of the applications. They contain only application virtual address references and range in size from 8 to over 32 million *data* references. Synchronization variables do not use spin-locking and, to minimize the possibility of hot spots, each synchronization variable is allocated to its own cache block.

2.2 Simulated Architectures

Two simulated architectures are used in this paper, the ideal and the detailed architecture. In the ideal architecture, caches are infinite; all memory references, reads or writes, hits or misses, take a single cycle; and every instruction executes in one cycle. We use the ideal architecture to remove dependencies on specific architecture characteristics from our study of shared data.

The detailed architecture, used to determine the practical implications of the ideal study, resembles the Silicon Graphics POWER Station 4D/240 [23] in memory system bandwidth and latency. Unlike the 4D/240 system, however, the detailed architecture has 16 processors, each of which has one 256 Kbyte direct-mapped data cache. In addition, synchronization accesses use the same bus as regular transactions. The memory access times without contention for 4- and 16-word blocks are 22 and 31 cycles respectively, during which the bus is locked for 6 and 15 cycles respectively. To simulate a steady state, the applications are executed twice; the first run warms up the cache, and the measurements are taken in the second run. Because bus contention would be too high with 32 processors, the detailed architecture is used for 16 processor runs only.

Both architectures use the invalidation-based Illinois cache coherence protocol [24]. Because in the 4D/240 a request for ownership on a shared block has the same timing and traffic requirements as a cache miss, we do not distinguish between the two in this paper.

2.3 Effect of an Optimizing Compiler on the Frequency of Sharing

While code optimizations are known to speed up uniprocessor applications [25], they have an important second effect in multiprocessor code: they increase the frequency of shared data references. This results from the different ways in which optimizations affect data. While some private references are eliminated by register allocation and other optimizations, shared data consistency prevents existing compilers from optimizing data declared shared, even if not used as such. Consequently, since some cycles are saved while the number of shared references remains the same, data sharing has a larger impact on the speed of the application.

To study the effect of an optimizing compiler, we measure, before and after compiler optimization, the fraction of references to data declared shared. The target architecture is the MIPS R2000 processor [26], which has 32 integer registers and 16 double-precision floating point registers. The optimizations applied include global register allocation and other conventional global optimizations. All data in the shared space is declared volatile, and therefore are not register-allocated or optimized. Because optimizations affect the different types of private data differently, we consider local and global private data separately. *Local* data are the variables declared within procedures. *Global* data is mostly static data set up by the master process for the slave processes.

Figure 1 shows the decomposition of the data reference streams for the optimized and unoptimized applications running with 16 and 32 processes. Due to limited disk space, the unoptimized versions of some traces were not run to completion (bars with a star). In those cases, the total number of references is calculated assuming the same relative ratios of private local, global, and shared references that existed when the trace was interrupted and the same number of shared references as the optimized trace. From the figure, we see that, for all applications, a large number of private references are eliminated, particularly among those directed to local variables. References to private global variables show a smaller change, almost solely due to the register allocation of the global pointer to the shared data space. We also see that the number of processes has little effect on the results. Appendix A shows tables with the actual numbers obtained in the experiments. The large difference in the ratio of shared to total references between optimized and unoptimized code suggests that performance studies of multiprocessor programs must be based on optimized code.

3 Analyzing Sharing

Data miss rates in large uniprocessor caches tend to vary predictably as cache blocks increase in size [11, 27, 28]. Initially, the miss rate drops quickly as the block size increases; for large blocks, around 32 words, the curve flattens out; eventually, there is a slight reversal of the curve because of misses resulting from conflicts. In contrast, how miss rates on shared data change with block size

Figure 1: Decomposition of the optimized and unoptimized data reference streams for 16 and 32 processes.

is much less predictable; experimental data shows a significant variation across programs (Figure 2). In this section, we first present a model of sharing that decomposes the widely varying miss rates on shared data in an invalidation-based cache coherence protocol into two well-behaved and intuitive components. Then, we describe an experiment to quantify each of these components. For simplicity, all the analysis in Section 3 assumes an infinite cache.

3.1 A Model of Sharing

Figure 3 shows the factors that determine the number of data misses in an infinite cache. For private data in single-word cache blocks, misses are solely caused by first-time references to the data. This effect we call *cold start* in Figure 3. If the cache has multi-word blocks, the prefetching provided by the multiple words of the block reduces the number of misses, as one miss is enough to bring all the words of a block into the cache. There are several more factors involved with the misses on shared data. If single-word blocks are used, *true sharing* as well as cold start dictate the misses. True sharing is the sharing of the same memory word by different processors. True sharing is intrinsic to a particular memory reference stream of a program and is not dependent on the block size. The presence of multi-word blocks further adds *false sharing* to true sharing, cold start, and prefetching effects. False sharing occurs when different processors access different words of the same block and the

Figure 2: Cache miss rates on shared data as a function of the block size for the ideal architecture. For a given application, the same problem size is used in the 16- and 32-processor executions.

coherence protocol forces the block to bounce among caches as if its words were truly being shared. A result of the collocation of different data in the same cache block, false sharing depends on the block size and the particular placement of data in memory. In the following paragraphs, we show how each individual cache miss can be traced back to these factors.

Figure 3: Factors that determine the data misses in an infinite cache.

True and false sharing are illustrated in Figure 4-(a), where words a and b are in the same memory block and an asterisk marks a cache miss. In Examples I, II, and III, processor P owns the block at the beginning of the reference stream, since P previously wrote words a and b — as denoted by Pa and Pb under 'Initial State'. In Example I, processor Q writes to word b and processor P writes to word a. In this classical case of false sharing, this pattern of access produces a miss for every access. Except for the first Qb reference however, no true data sharing is involved. In Examples II and III, processors P and Q need, and therefore truly share, word a. Word b is used only by P in both cases. However, because of the prefetch provided by the cache block, this common sharing pattern produces misses on different words in the two examples. A more complex sharing pattern can interact with the cache block in a variety of ways, resulting in different numbers of misses. The model we present now analyzes how data sharing and prefetching interact to result in the observed number of misses.

We assume a multiprocessor with infinite caches and an invalidation-based cache coherence protocol where a cached memory word may be owned by one cache or read shared among several. We define the *state* of the word as the pair *(mode, processors)*, where *mode* may be *owned* or *shared*, and *processors* is the set of processors that cache the word. An uncached word is a degenerate case where *processors* is $\emptyset$. A read miss loads the word in a shared mode. If the word is in a shared mode, a processor that caches it must issue a request for ownership before it can write the word. We count this request as a miss. A change in the state of the word is called a *state transition.* In the following, we focus on conditions after the cold start for the word, when no processor will access the word for the processor's first time.

To quantify the degree of intrinsic sharing of a memory word, we define the concept of true sharing transition.

Figure 4: Example of memory reference streams. For simplicity, the streams contain only writes. An asterisk marks a cache miss. The streams in part (a) are expanded in part (b) showing true (T) and false (F) sharing transitions, and misses saved by successful prefetches (X).

Definition 1: True Sharing Transition

Consider the stream S of references to a given memory word only and ignore any effects caused by references (not in S) to the other words in the same cache block, as if the block were single-worded. We call true sharing transition any state transition that occurs between two references that are contiguous in S, after cold start. Further, we say that the second reference causes a true sharing transition.

Example II in Figure 4-(a) shows two true sharing transitions for word a. One occurs between the initial state and reference Qa; the second between Qa and the last reference.

True sharing transitions and cache misses are strongly related: in caches with single-word blocks, every true sharing transition causes a cache miss, and every miss after cold start is due to a true sharing transition. In caches with multi-word blocks, however, a true sharing transition does not necessarily lead to a miss. This is shown in the second true sharing transition for word a in the same example. Between the two references involved in the transition, namely Qa and Pa, a third reference Pb to another word of the same block prefetches the original word to the desired state, owned by processor P. As a result, the second reference Pa hits. On the other hand, the first true sharing transition for word a in the same example, which occurs between the initial state and Qa, produces a miss. We can now define the concept of true sharing miss.

Definition 2: True Sharing Miss

A miss that occurs in a true sharing transition.

The previous discussion shows that prefetching can eliminate a miss in the second reference of a true sharing transition. Prefetching can also generate a miss in a reference that does not cause a true sharing transition. To formalize this situation, we first define the concept of false sharing transition.

Definition 3: False Sharing Transition

Consider two consecutive references to the same word where the second reference does not cause a true sharing transition. If, between the two references, there is at least one intervening reference to a different word of the same block that induces a transition on the second reference, we say that the second reference causes a false sharing transition.

As an example, the second Pb reference in Example II causes the only false sharing transition for word b in the stream: between the two Pb references, the intervening Qa reference changes the state of word b to be owned by Q, thereby inducing a transition on the second Pb reference.

Like a true sharing transition, a false sharing transition may or may not incur a miss. An example where a miss occurs is the false sharing transition for word *b* in Example II: between the two *Pb* references, reference *Qa* leaves the block in state owned by *Q*, causing the second *Pb* reference to miss. An example where the miss is avoided is shown in Example III, which is equal to Example II with the last two references flipped. In Example III, the second *Pb* reference causes a false sharing transition because reference *Qa* between the two *Pb* references induces a transition on the second *Pb*. Between *Qa* and *Pb*, however, reference *Pa* brings the block back to *P*'s ownership, thus successfully eliminating a cache miss in the *Pb* reference. We can now define the concept of false sharing miss.

Definition 4: False Sharing Miss

A miss that occurs in a false sharing transition.

Finally, based on the above definitions, the total number of cache misses, not counting the cold start effect, is the total number of true and false sharing transitions minus the number of successful prefetches. This equality is illustrated in Figure 4-(b), which expands the streams in Figure 4-(a). We analyze Example II carefully here; the reader is encouraged to go over the other examples. We consider the first reference after the initial state *Qa* and ask whether it causes a false sharing transition (FST), a true sharing transition (TST), or no transition at all. To answer this question, we look at the previous reference to the same word, namely *Pa*. We note that the two references are involved in a TST. We then check whether the accesses between the two references leave word *a* in the state that *Qa* requires, namely owned by *Q*. If that were the case, a successful prefetch would be recorded. Otherwise, the actual situation in the example, a true sharing miss (TSM) occurs. We now consider the next reference *Pb*. As before, we look for the previous reference to the same word, namely the *Pb* under Initial State. This pair of references are not involved in a TST. To check whether a FST occurs, we search the intervening references for at least one that induces a transition on the second *Pb*. Since *Qa* induces such a transition, *Pb* causes a FST. To determine whether a false sharing miss (FSM) or a successful prefetch occurs, we check whether the stream between *Qa* and *Pb* leaves *b* in the state required by *Pb*. Since this is not the case, a FSM occurs. The final reference causes a TST but a successful prefetch eliminates the miss: reference *Pb* sets the block to the desired state, namely owned by *P*. To summarize, the net result of three transitions and one successful prefetch is two misses, as postulated by our equality that relates misses, transitions, and successful prefetches.

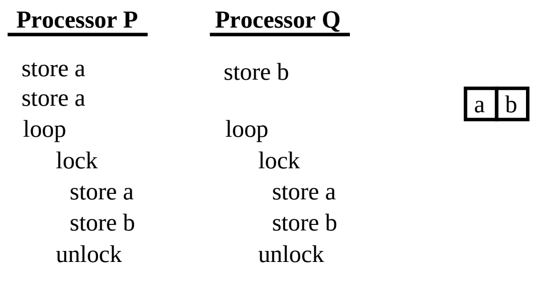
3.2 Effect of Data Prefetching through Increased Block Size

The previous analysis showed that the prefetching provided by multi-word blocks can eliminate or create misses. Unlike in uniprocessors, where prefetching always has a positive effect in infinite caches, prefetching in multiprocessors can have both a positive and a negative effect. Prefetching exploits spatial locality in data as in uniprocessors. It also, however, creates false sharing transitions, which may change what used to be cache hits without prefetching into false sharing misses.

We expect the positive effect, namely exploitation of spatial locality, to be lower in multiprocessors than in uniprocessors for three reasons. First, a processor may never reference the prefetched data: since computation is partitioned in a multiprocessor, this is more likely than in a uniprocessor. Second, even if the processor will eventually access the prefetched data, another processor may access it first and remove the data from the first processor's cache. Third, prefetched data may be removed by another processor accessing a different word in the same block. Because of the last reason, the benefits of spatial locality do not necessarily increase monotonically with the cache block size. Larger blocks may introduce transitions that reduce the spatial locality benefits present in a smaller block size.

False sharing transitions, the second effect of prefetching, increase monotonically with block size. As false sharing transitions increase, false sharing misses are likely to increase. However, since not every false sharing transition will cause a false sharing miss, the number of false sharing misses may not increase monotonically with increasing block size either.

Unfortunately, both the positive and negative effects of prefetching are determined by the particular placement of data in memory and cannot, in general, be changed independently. Figure 5 illustrates the interdependence of the two effects. In the figure, words a and b share the same block. In the beginning of the program, a potential instance of false sharing occurs because processor Q may write b while processor P writes a. During the rest of the program, the two processors access words a and b in sequence within a critical section. If we eliminated the false sharing by, for example, placing a in a different block, the benefits of prefetching within the loop would also disappear. We could be saving one false sharing miss at the cost of doubling the number of misses within the loop. This example suggests that it may not be desirable to eliminate false sharing misses at any cost.



```
Processor P        Processor Q

store a            store b
store a
loop               loop
    lock               lock
      store a            store a
      store b            store b
    unlock             unlock
```

Figure 5: The data prefetching provided by multi-word cache blocks can be beneficial, as in the loop shown, or may create false sharing misses, as in the statements before the loop.

3.3 Measurements

Although the positive and the negative effects of prefetching on the miss rate are closely related, we have been able to devise an experiment that allows us to measure each of the two effects. The experiment is based on our model of sharing. In the experiment, we use the ideal architecture, which assumes infinite caches and no cache miss penalties. We compare two simulations driven by the same interleaving of references and running in lockstep. One simulation uses caches with single-word blocks, while the other uses caches with multi-word blocks. In the simulations, we include the cold start period of the programs. Hence, in addition to false and true sharing misses, we capture misses on memory words referenced by a processor for the processor's first time. These misses we call *cold misses*. The relationships among cold, true sharing, and false sharing misses in the single-word and the multi-word simulations are as follows.

> If a cold miss is incurred for a reference in the multi-word block simulation, the same reference causes a cold miss in the single-word block simulation.
>
> True sharing transitions are intrinsic to a reference stream of a program and thus identical for both simulations. Since all true sharing transitions result in misses (true sharing misses) in the single-word case, if a true sharing transition in the multi-word block simulation causes a miss, it also causes a miss in the single-word block simulation.
>
> Therefore, the remaining misses, those that occur in the multi-word block simulation but not in the single-word case, must be all false sharing misses.

In summary, comparing the two simulations, a miss in the multi-word simulation is a false sharing miss if there is no equivalent miss in the single-word case; otherwise it is a cold or true sharing miss.

Figure 6 (not drawn to scale) depicts the relationships described. The number of cold references and true sharing transitions are the same in both simulations. Prefetching multiple words in a cache block has two effects: first, some of the cold references and true sharing transitions now result in hits; second, false sharing transitions appear, some of which result in hits and some in misses.

4 Analyzing the Cache Miss Rate and Traffic Behavior of Shared Data

In this section, we use the experiment just described to analyze the cache miss rates and traffic generated by shared data in real applications. To eliminate dependencies on specific architecture characteristics, we use the ideal architecture throughout the section. We start with an analysis of the miss rates; then we consider the traffic behavior.

Figure 6: Relation between the simulation of the ideal architecture using single and multi-word cache blocks. The figure is not drawn to scale. In the figure, the number of cold references and true sharing transitions are the same in both simulations.

4.1 Analysis of the Cache Miss Rates on Shared Data

Figure 2 shows the miss rates on shared data as a function of the block size for the applications studied. We observe a wide variation among applications, both in absolute values and in the way the block size affects them. For example, whereas miss rates for Csim, DWF, and LocusRoute start from relatively low values and decrease with increasing block size, Maxflow's miss rate starts with a higher value, decreases at first, and then increases. Mp3d's miss rate is high and not very sensitive to changes in the block size. Finally, Mincut shows an upward trend.

To understand the variation observed with changes in the block size, we plot the miss curves in relative values (Figure 7) and then decompose them into the miss components as described by our model. Figure 8 shows the misses for 16 processors decomposed into two groups: cold and true sharing misses, and false sharing misses. In addition, to show the degree of true sharing in each program, we mark with an arrow the number of true sharing misses on single-word blocks — which is also the number of true sharing transitions. The rest of the misses on single-word blocks are cold misses. In the following sections, we first analyze each component of the misses separately, relating the shape of the curves to the data structures in the source code that cause them. Then, we summarize the general observations.

4.1.1 Analyzing False Sharing Misses

Recall that, while false sharing transitions always increase monotonically with the block size, this is not necessarily so for false sharing misses. From Figure 8, however, we observe that false sharing

Figure 7: Cache misses on shared data as a function of the block size. Misses are shown as a fraction of the misses on single-word blocks for the same application and 16 processors.

Figure 8: Decomposition of the cache misses on shared data as a function of the block size for 16 processors. Misses are shown as a fraction of the misses on single-word blocks for the same application. The arrow shows the number of true sharing transitions in the program.

misses always increase with block size and that, except in two cases, this increase is slow. This slow increase is produced by several program characteristics. Distributing the computation such that each iteration of a loop is executed on a different processor produces false sharing misses when data from different iterations falls in the same cache block. Graph problems with irregular node interconnection where cache blocks frequently contain pieces of nodes belonging to different processors also exhibit false sharing misses (Maxflow and Mincut).

The two cases where false sharing misses increase quickly are when the blocks are small in Mincut and when the blocks are large in Maxflow. The sharp rise in these two cases is due to the presence of blocks containing multiple frequently-accessed scalar variables, where at least one of the scalars is written frequently. This effect can also happen with small arrays where each array entry is repeatedly updated by one processor.

As opposed to the previous applications, programs with little reuse of data by the same process (Mp3d), or where each processor is assigned a geographic domain where processor interaction is infrequent (LocusRoute, DWF, and Csim) are unlikely to exhibit a large amount of false sharing misses.

4.1.2 Analyzing Cold and True Sharing Misses

The slow decrease in cold and true sharing misses with increasing block size seen in Figure 8 shows that shared data has low spatial locality. A second observation is that, except for Maxflow, which shows a slight trend reversal for large blocks, the decrease in misses is monotonic.

Poor locality particularly affects programs with unstructured accesses, as is the case in fine-grained global task queues where processors continually process new tasks (Mp3d) or algorithms like simulated annealing that involve calls to random number generators to decide what memory area to access (Mincut). On the other side, programs with large data structures that are accessed sequentially and at different times by different processors (LocusRoute and Maxflow) show larger decreases in cold and true sharing misses.

4.1.3 Increasing the Number of Processors

The curves with the misses for 32 processors shown in Figure 7 are decomposed in Figure 9. The misses are shown as a fraction of the 16-processor, single-word block misses for the same application. From the figure, we see that the two components, false sharing and cold/true sharing misses, maintain the same trends for the larger number of processors.

Figure 9: Decomposition of the cache misses on shared data as a function of the block size for 32 processors. Misses are shown as a fraction of the misses on single-word blocks for the same application and 16 processors. The arrow shows the number of true sharing transitions in the program.

4.1.4 Overall Observations

Despite the widely varying shape of the overall curve, the two component curves behave consistently across all applications. First, cold and true sharing misses tend to decrease with increasing block size but, unlike in uniprocessors, the rate of decrease in misses is much less than the rate of increase in block size. Second, false sharing misses increase with block size and eventually neutralize or even overcome the small decreases in the cold and true sharing misses. The net effect is that the total number of misses either decreases slowly or does not decrease at all. As we will see, the result is a dramatic increase in processor-memory traffic with any increase in block size.

The plots show that cold and true sharing misses usually outnumber false sharing misses. Further, for the two applications with a significant number of false sharing misses, we show in Section 5 that simple data placement optimizations can eliminate an important fraction of these false sharing misses.

The two component curves in each plot may not be independent of each other. Figure 5 showed that a reduction in the number of false sharing misses may cause an increase in the number of cold and true sharing misses. The opposite case, namely a reduction in the number of false sharing misses causing a decrease in the amount of cold and true sharing misses, is also possible. Such scenario occurs if false sharing misses induce more misses by interfering with the successful prefetches for true sharing or cold accesses. In the worst case, a false sharing miss on a word by one processor could eliminate a successful prefetch in all the other processors that cache the word, thereby forcing cold or true sharing misses. Fortunately, the experiments performed while studying the optimizations of Section 5 show that such interaction is rare. The curves of false sharing misses, therefore, are a good approximation of the worst effects of false sharing.

The magnitude of the two component curves and the previous discussion suggest that the poor spatial locality of multiprocessor data — responsible for the slow decrease in cold and true sharing misses — contributes to the cache miss rates even more than false sharing does. For this reason, we believe that, to improve the performance of caches, trying to enhance the spatial locality of multiprocessor data is an approach at least as, or even more promising, than trying to remove false sharing.

4.2 Analysis of the Traffic Generated by Shared Data

Not only do misses increase the latency of memory accesses, they also generate traffic between processors and memory. As the block size increases, a miss produces a higher volume of traffic. If we estimate the traffic caused by shared data as *SharedMisses*BlockSize*, we produce the plots in Figure 10. The figure includes a curve for uniprocessor data with a *finite* cache (32 Kbytes) from [11] for comparison purposes. From the figure, we see that the block size that minimizes the traffic of shared data in this class of applications is one word, both for 16 and 32 processors. To determine the highest performance block size for a data cache, however, we need to take into account the start up overhead associated with a cache miss for the particular machine and know what fraction of the data misses are on shared data. Section 6 shows that this fraction is over 95% for a large cache.

Figure 10: Processor-memory traffic caused by shared data. The plot shows the ratio between the traffic at a given block size and the traffic for single-word blocks and 16 processors. We include a curve for uniprocessor data with a *finite* cache (32 Kbytes) for comparison purposes.

The traffic increase with larger blocks occurs because many of the words transferred are not used. Between two consecutive misses on a given block, a processor usually references a very small number of distinct words in that block, as shown in Table 2. Recall that misses include requests for ownership on a block. The low values in Table 2 show that, on average, the prefetching effect of cache blocks is not very effective. These numbers correlate with the trends in the miss rates shown in Figure 2. Mp3d has the lowest numbers in Table 2 because it has a high miss rate, which does not decrease with larger block sizes. LocusRoute shows the highest numbers because it has a low miss rate that decreases significantly with increases in block size. We also see that increasing the number of processors always decreases the number of words used in a block. The poor use of the cache blocks revealed by this data motivates the next section, where we try to optimize the use of the blocks based on our model of sharing.

Table 2: Average number of distinct words in a cache block referenced by one processor between two consecutive misses on that block by the same processor.

Application	16 Processors Block Size (Words)					32 Processors Block Size (Words)				
	2	4	8	16	32	2	4	8	16	32
Csim	1.4	1.9	2.4	3.0	3.5	1.4	1.8	2.3	2.7	3.2
DWF	1.5	1.9	2.4	2.3	2.5	1.4	1.7	1.9	2.0	2.1
Mp3d	1.1	1.1	1.1	1.1	1.1	1.0	1.1	1.1	1.1	1.1
LocusRoute	1.3	2.1	3.5	5.2	6.0	1.2	2.0	3.0	4.1	4.3
Maxflow	1.3	1.9	2.5	3.1	3.3	1.2	1.7	2.0	2.3	2.5
Mincut	1.2	1.2	1.4	1.7	2.0	1.1	1.2	1.3	1.4	1.7

5 Optimizing the Placement of Shared Data in Cache Blocks

This section addresses the problem of reducing the cache misses on shared data by enhancing the spatial locality of shared data and mitigating false sharing. We optimize the placement of data structures in cache blocks using local changes that are programmer-transparent and have general applicability. Our approach is partly motivated by the fact that cache misses on shared data are often concentrated in small sections of the shared data address space. Therefore, local actions involving relatively few bytes may yield most of the desired effects. An example of this skewed miss distribution is shown in Figure 11, which plots the average number of misses per byte in each shared data structure of Csim.

Figure 11: Distribution of the cache misses along the shared address space for the Csim application. For each data structure, we plot the average number of misses per byte. This plot corresponds to 16 processors, 4-word cache blocks, and the ideal architecture.

To guide the study of possible optimizations, we use address traces to generate the following profiling information for each shared-memory word: (1) degree of true sharing, measured as the number of misses beyond the cold start in the single-word block simulation, (2) false sharing misses, (3) cold and true sharing misses eliminated by prefetching, and (4) number of writes. The latter is needed since, in addition to words that have a high degree of true sharing, non-shared words that are frequently written can also be the cause of false sharing in a block. For example, false sharing may occur in a block with one word that is heavily read by only one processor and one word that is heavily written by only one other processor. We call a word *active* if its degree of true sharing or number of writes exceeds 0.1% of the program misses.

In the following, we first present the optimizations, then evaluate them using the ideal architecture. In the evaluation, we consider both the aggregate effect of all optimizations and the individual effect of each. Since it is rare to have this tracing information in practice, the final subsection examines the case where we have no dynamic information on the application at all.

5.1 Placement Optimizations

We propose five optimizations of the data layout. Because synchronization variables are a well-known source of contention in some programs, we use as a baseline a data layout where each of them is allocated to an empty cache block.

- *SplitScalar: Place scalar variables that cause false sharing in different blocks.* Given a cache block with scalar variables where the increase in misses due to prefetching exceeds 0.5% of the program misses, we remove the active variables and allocate each of them to an empty cache block.

- *HeapAllocate: Allocate shared space from different heap regions according to which processor requests the space.* It is common for a slave process to access the *shared* space that it requests itself. If no action is taken, the space allocated by different processes may share the same cache block and lead to false sharing. The policy we propose is more space-efficient than allocating only block-aligned space, particularly when very small chunks of space are repeatedly requested.

- *ExpandRecord: Expand records in an array (padding with dummy words) to reduce the sharing of a cache block by different records.* While successful prefetching may occur within a record or across records, false sharing usually occurs across records, when more than one of them share the same cache block. If the multi-word simulation indicates that there is much false sharing and little gain in prefetching, then consider expansion. If the reverse is true, do not apply the optimization. When both false sharing misses and prefetching savings are of the same order of magnitude, we assume that the prefetching succeeds within a record and we apply the optimization.

- *AlignRecord: Choose a layout for arrays of records that minimizes the number of blocks the average record spans.* This optimization maximizes prefetching of the rest of the record when one word of a record is accessed, and may also reduce false sharing. This optimization is possible when the number of words in the record and in the cache block have a greatest common divisor (GCD) larger than 1. The array is laid out at a distance from a block boundary equal to 0 or a multiple of the GCD, whichever wastes less space.

- *LockScalar: Place active scalars that are protected by a lock in the same block as the lock variable.* As a result, the scalar is prefetched when the lock is accessed.

All optimizations except *LockScalar* try to minimize false sharing. *LockScalar* and *AlignRecord* try to increase the spatial locality of the data. In our optimizations, we must avoid other effects that could offset the intended ones. First, false sharing and effective exploitation of spatial locality are not independent; changing one usually affects the other. In particular, strategies that increase the size of the data like *SplitScalar* and *ExpandRecord* may also reduce the effectiveness of prefetching in eliminating cold and true sharing misses. Second, large data expansions may increase the working set of a program and increase capacity misses in a finite cache. To guard against these effects, we restrict the optimizations to those that cause little data size increase.

5.2 Evaluation of the Optimizations: Aggregate Effect

To evaluate the effectiveness of these optimizations, we use as a metric the fraction of shared data misses that they eliminate. Table 3 shows this fraction together with the resulting increase in the size of the data structures for 16 and 32 processors with 4- and 16-word blocks. The table shows a large variation in the fraction of misses eliminated in the different applications: the results for individual programs range from 0% to over 40%, with an average close to 10%.

On average, our techniques tend to eliminate a higher percentage of misses for the larger block sizes. This effect is, however, the result of two opposing trends. On one hand, a larger cache block size increases the possibility of false sharing among scalars and small data structures, thus possibly increasing the effectiveness of the optimizations. On the other hand, a larger block also increases the cost of expanding records, making some data expansion optimizations infeasible. Further, a larger block may already benefit more from prefetching, rendering optimizations to increase spatial locality less effective.

The effect of the number of processors is also clear. When the number of processors increases, there are more cache misses. The data placement optimizations, however, also eliminate more misses.

Table 3: Effectiveness of the optimizations in the placement of shared data for 16 (top half of the table) and 32 (bottom half of the table) processors.

Application	Number of Processors	Reduction in Shared Data Misses				Increase in Shared Data Space			
		4-Word Blocks		16-Word Blocks		4-Word Blocks		16-Word Blocks	
		Relat. (%)	Absol. (Thous.)	Relat. (%)	Absol. (Thous.)	Relat. (%)	Absol. (Kbytes)	Relat. (%)	Absol. (Kbytes)
Csim	16	7.9	60.6	6.6	39.3	0.0	0.4	0.1	1.9
DWF	16	0.6	3.1	1.0	4.6	0.0	0.0	0.0	0.2
Mp3d	16	0.4	20.6	0.1	5.5	0.3	4.8	0.0	0.5
LocusRoute	16	10.2	45.3	28.7	57.5	0.0	0.5	0.1	1.6
Maxflow	16	8.9	198.9	14.2	235.3	0.6	1.6	0.6	1.6
Mincut	16	19.7	229.6	8.9	153.1	72.3	4.0	0.0	0.0
AVERAGE	16	8.0		9.9			1.9	0.1	1.0
Csim	32	15.6	196.2	11.5	110.3	0.0	0.9	0.1	3.9
DWF	32	0.6	5.5	1.1	9.0	0.0	0.0	0.0	0.2
Mp3d	32	0.4	24.3	0.2	13.3	0.3	4.8	0.0	0.5
LocusRoute	32	15.5	92.5	41.6	138.5	0.0	0.5	0.2	3.1
Maxflow	32	10.7	397.0	14.7	455.6	0.6	1.6	0.6	1.6
Mincut	32	22.0	394.1	8.8	190.9	72.3	4.0	0.0	0.0
AVERAGE	32	10.8		13.0			2.0	0.2	1.5

The result, for nearly all the applications studied, is that the relative miss reductions are higher for 32 processors than for 16 processors.

Finally, we see that the space requirements of the optimizations are small, usually in the 2 Kbyte neighborhood. This causes an insignificant relative increase in shared data space unless the size of the shared data space is very small originally. While it is possible to reduce the miss rate further by larger data expansions, their possibly detrimental effect on cache performance makes them undesirable.

Figure 12 shows how the optimizations affect the two types of misses: cold and true sharing, and false sharing misses. For each application, the figure considers the four processor and block size settings used in the previous table. For each setting, we show three bars. The leftmost bar shows the miss rate of shared data in the original program, where the compiler did not necessarily allocate each synchronization variable to a different cache block. The central bar shows the miss rate after each synchronization variable is allocated to a different cache block. This is the miss rate taken as a baseline. From the difference between the two bars, we can see the importance of the synchronization variable layout, especially considering that spin-locking is not used in the synchronization variables. Finally, the rightmost bar shows the miss rate after further applying the five placement optimizations.

We observe that the optimizations are more successful in eliminating false sharing misses than in eliminating cold and true sharing misses. For all applications, the maximum reduction in cold and true sharing misses is approximately 10%. In contrast, almost all false sharing is removed in LocusRoute and in Mincut for 4-word cache blocks, and 20 to 40% in Csim and Maxflow. The reduction of false sharing in Mincut is accompanied by an increase in cold and true sharing misses. This observation illustrates that, in general, the positive and negative effects of prefetching discussed in Section 3.2 cannot be totally separated.

5.3 Evaluation of the Optimizations: Individual Effect

Table 4 shows the contribution of each optimization to the reduction in shared data misses shown in Table 3. From Table 4, we see that *SplitScalar* is effective for all applications amenable to these optimizations. Most of the misses it eliminates can be attributed to two or three actively written scalars. As the block size increases, this optimization becomes more important. We also see that *ExpandRecord* is useful for the same set of applications. This optimization is applied either to small, active arrays used mainly for process communication or to the main data structures in the smaller programs. Finally, the other optimizations are relevant to only one or two of the applications.

A large fraction of the cache misses still remains after optimization. While some of the false sharing misses can be removed if the data caches are large enough to support more instances of the expansion optimization, the remaining misses are primarily cold and true sharing misses. This suggests that further optimizations should concentrate on increasing the spatial locality of the data.

Figure 12: Miss rates on shared data. For each set of three bars, the leftmost one shows the miss rate of the original program; the central one the miss rate after allocating synchronization variables to different cache blocks; and the rightmost one the miss rate after further applying the five placement optimizations.

Table 4: Fraction of shared data misses eliminated by each optimization. The notation 16p-4w means 16 processor execution and 4 words per cache block.

Application	*SplitScalar*	*HeapAllocate*	*ExpandRecord*	*AlignRecord*	*LockScalar*	Total
Csim 16p-4w	1.7		2.3	3.4	0.5	7.9
Csim 16p-16w	2.2		2.8	1.0	0.6	6.6
Csim 32p-4w	3.8		2.3	9.0	0.5	15.6
Csim 32p-16w	5.0		2.8	3.0	0.7	11.5
DWF 16p-4w	0.4			0.2		0.6
DWF 16p-16w	1.0					1.0
DWF 32p-4w	0.5			0.1		0.6
DWF 32p-16w	1.0			0.1		1.1
Mp3d 16p-4w	0.1		0.3			0.4
Mp3d 16p-16w	0.1					0.1
Mp3d 32p-4w	0.1		0.3			0.4
Mp3d 32p-16w	0.1			0.1		0.2
LocusRoute 16p-4w	1.5	6.7	0.7	0.5	0.8	10.2
LocusRoute 16p-16w	8.0	16.1	2.5	0.4	1.7	28.7
LocusRoute 32p-4w	1.4	11.3	1.4	0.8	0.6	15.5
LocusRoute 32p-16w	8.4	27.5	4.7	0.4	0.6	41.6
Maxflow 16p-4w	1.5		5.3		2.1	8.9
Maxflow 16p-16w	2.0		9.3		2.9	14.2
Maxflow 32p-4w	1.7		4.8		4.2	10.7
Maxflow 32p-16w	2.0		7.7		5.0	14.7
Mincut 16p-4w	4.7		9.6		5.4	19.7
Mincut 16p-16w	4.1				4.8	8.9
Mincut 32p-4w	6.7		11.3		4.0	22.0
Mincut 32p-16w	5.5				3.3	8.8

5.4 Effectiveness of the Optimizations without Program Profiling

The optimizations evaluated above were developed by using detailed information obtained by tracing the program. While some kind of profiling may be available in practice, it will probably not be as complete as the one used so far. In this section, we investigate the possibility of general and effective optimizations that do not rely on any profiling information. We consider how to apply each of the previous optimizations in the absence of this information:

- *SplitScalar*: If no information is provided, we place *each* shared scalar variable in a different cache block. This approach has almost the same effect as moving only active scalar variables since, in relatively large caches, the advantage of prefetching scalars is minor. Although most programs have a small number of shared scalars (the number in those studied ranged from 5 to 50), programs with many scalars and large cache blocks may waste much space. However, we expect little negative effect, since only a fraction of the scalars is accessed frequently.

- *ExpandRecord*: To expand all short arrays by placing, for example, one entry per cache block is impractical, since it wastes space and can have a positive or a negative net effect on cache misses. We leave it up to the programmer to pad the data structure if so desired.

- *HeapAllocate* and *AlignRecord*: The optimizations of allocating shared data from a process' own heap space and aligning arrays can be applied at all times, since the cost is low.

- *LockScalar*: If the machine allows lock variables and general data to reside in the same cache block, this optimization is feasible at a very low cost.

From the previous discussion, we conclude that the compiler and run time system can incorporate *HeapAllocate*, *AlignRecord*, *LockScalar*, and the modified *SplitScalar* without any profile information. The cumulative effect of these optimizations is shown in Table 5, together with a comparison to the fully optimized case. These numbers indicate that a significant part of the effect of the more costly optimizations can be obtained without any profile information. Moreover, the increase in data space, both absolute and relative, remains small.

6 Performance of a Real Architecture

After having studied data sharing in an ideal setting, we now use the detailed architecture to illustrate the performance impact of data sharing in practice. This section examines three issues. We first study the effect of the conventional code optimizations described in Section 2.3. Using optimized code, we then measure the overall cache performance of the applications. Finally, also using optimized code, we assess the effectiveness of the placement optimizations for shared data.

Table 5: Effectiveness of the optimizations without using a program profile. The numbers in parenthesis show the misses eliminated as a percentage of the misses eliminated with a full program trace.

Application	Shared Data Misses Eliminated (%)				Shared Data Space Increase (Kbytes)			
	16 Proc. 4-Word Block	16 Proc. 16-Word Block	32 Proc. 4-Word Block	32 Proc. 16-Word Block	16 Proc. 4-Word Block	16 Proc. 16-Word Block	32 Proc. 4-Word Block	32 Proc. 16-Word Block
Csim	5.6 (71)	3.8 (58)	13.3 (85)	8.7 (76)	0.6	2.9	0.6	2.9
DWF	0.6 (100)	1.0 (100)	0.6 (100)	1.1 (100)	0.1	0.6	0.1	0.6
Mp3d	0.0	0.1	0.0	0.1	0.4	1.9	0.4	1.9
LocusRoute	8.9 (87)	25.9 (90)	14.0 (90)	36.9 (89)	0.6	2.6	0.6	2.6
Maxflow	3.6 (40)	4.9 (35)	5.9 (55)	7.0 (48)	0.1	0.4	0.1	0.4
Mincut	10.1 (51)	8.9 (100)	10.7 (49)	8.8 (100)	0.0	0.3	0.0	0.3
AVERAGE	4.8 (60)	7.4 (75)	7.4 (69)	10.4 (80)	0.3	1.4	0.3	1.4

6.1 Impact of the Conventional Code Optimizations

To study the effect of the conventional code optimizations on overall performance, we compare the execution times of two applications, LocusRoute and Maxflow, using optimized and unoptimized code. LocusRoute is about twice as fast after optimization for both 4- and 16-word blocks. However, Maxflow yields an improvement of only about 5% for both 4- and 16-word blocks. This small improvement in Maxflow is due to increased bus contention, which offsets the advantages gained by the elimination of unnecessary private data fetches from the program. Thus, while there is a slight improvement in the speed of Maxflow, the utilization of the processors actually decreases by 25%. In conclusion, while some programs run substantially faster with compiler optimizations, those where shared data traffic saturates the interconnection cannot. In either case, since uniprocessor programs run faster while the amount of sharing remains unchanged, optimized code will give lower speedup figures. Since we are ultimately interested in overall performance, measurements on multiprocessor programs must be performed on optimized code.

6.2 Overall Cache Performance

In previous sections we studied the cache miss rates resulting from data sharing in isolation. In this section we examine the data cache performance of the detailed architecture, which has a finite cache and issues private data references too. As indicated in Section 2.2, the measurements are taken during the steady state execution of the programs. In this environment, the contribution of private and shared data to the misses of the finite caches is shown in Table 6. Because the caches are reasonably large and the programs are measured in their steady state, the miss rate on private data (columns 2 and 3) is minuscule compared to that on shared data (columns 4 and 5). In fact, most of the misses correspond to shared data (columns 6 and 7). Consequently, as shown in the last two columns of Table 6, the total miss rate is basically the shared data miss rate weighted by the frequency of shared data accesses.

Table 6: Data cache miss measurements for the detailed architecture with 16 processors and compiler-optimized code.

Application	Private Misses / Private References (%)		Shared Misses / Shared References (%)		Shared Misses / Total Misses (%)		Total Misses / Total References (%)	
	4-Word Block	16-Word Block	4-Word Block	16-Word Block	4-Word Block	16-Word Block	4-Word Block	16-Word Block
Csim	0.1	0.2	9.5	7.7	99	98	5.9	4.8
DWF	0.0	0.1	2.9	2.1	99	99	2.6	1.9
Mp3d	0.2	0.2	59.5	59.2	99	99	46.5	46.3
LocusRoute	0.5	0.3	10.6	5.9	95	94	5.5	3.0
Maxflow	0.3	0.3	13.4	10.0	98	97	8.3	6.2
Mincut	0.0	0.0	19.6	19.1	99	99	6.6	6.4

6.3 Impact of the Placement Optimizations

We have proposed two sets of data placement optimizations: one when full tracing information is available; the other when no profiling data is available. In practice, some information will probably be available. We therefore choose to evaluate the case that assumes full information and consider the results optimistic.

Table 7 shows the reduction in data miss rate achieved by the placement optimizations for 16 processors. The data in the table includes misses on both shared and private data. From the table, we see that the optimizations reduce the overall data miss rate of the applications by up to an absolute 1.5% (or a relative 40%). These miss rate reductions speed up the applications by about 10% on average. These speedups are partially the result of the bus contention generated by sixteen processors. However, while replacing the bus with another interconnection network may reduce contention, it may also increase overall memory access latencies.

Table 7: Effect of the shared data placement optimizations on the data miss rates of the detailed architecture. The numbers correspond to 16 processors and compiler-optimized code, and include both shared and private data.

Application	Overall Data Miss Rate					
	4-Word Block			16-Word Block		
	Unopt. (%)	Opt. (%)	Unopt. - Opt. (%)	Unopt. (%)	Opt. (%)	Unopt. - Opt. (%)
Csim	5.9	5.1	0.8	4.8	4.4	0.4
DWF	2.6	2.6	0.0	1.9	2.0	-0.1
Mp3d	46.5	46.4	0.1	46.3	46.2	0.1
LocusRoute	5.5	4.0	1.5	3.0	1.8	1.2
Maxflow	8.3	7.2	1.1	6.2	5.1	1.1
Mincut	6.6	5.4	1.2	6.4	5.5	0.9

7 Conclusions and Future Directions

There are two main contributions in this paper. First, we show how poor spatial locality in the data and false sharing explain the variation in the miss rate of shared data as the cache block changes in size. Second, we show that data layout optimizations that are programmer-transparent and not restricted to regular codes can be used to reduce the miss rate.

Based on the analysis of six applications, we find that, although false sharing sometimes plays a significant role, poor spatial locality has a larger effect in determining the high miss rates for moderate-sized cache blocks. In addition, data layout optimizations are more effective in eliminating false sharing than in improving spatial locality. Overall, these optimizations eliminate about 10% of the misses on shared data.

Our observations on where and how false sharing occurs lead us to hypothesize that false sharing is not the major source of the cache misses in compiler-parallelized code either. For such code, the compiler can easily avoid the obvious false sharing pitfalls. For example, in a *DOALL* loop, it is well known that interleaving individual iterations across different processors can cause false sharing. This effect can be avoided by increasing the granularity of the slices assigned to processors.

Optimizations that improve the performance of cache memories are likely to grow in importance as the latencies of cache misses increase. Of these optimizations, those that specifically optimize the performance of large cache blocks, like the ones presented here, are particularly interesting, since large blocks can be useful in amortizing the cost of a long-latency memory access. More effort should be devoted to optimizing the performance of large cache blocks. In this paper, we have shown data that suggests that researchers should focus on increasing the spatial locality of the data more than on reducing false sharing.

Acknowledgements

We thank the referees for their helpful comments. We also thank Steve Goldschmidt, Brian Bray, Helen Davis, Steve Tjiang, and the authors of the applications for their contributions. This research is funded in part by DARPA contract N00014-87-K-0828 and by financial support from *La Caixa d'Estalvis per a la Vellesa i de Pensions* and the *Ministerio de Educacion y Ciencia*, both of Spain.

A Decomposition of the Data Reference Stream

Tables 8 and 9 classify the data references for 16- and 32-process streams respectively.

Table 8: Decomposition of the data reference stream for 16 processes.

Application	Shared Refs. / Total Data Refs. (%)		Private Global Refs. / Total Data Refs. (%)		Private Local Refs. / Total Data Refs. (%)	
	Unopt.	Optim.	Unopt.	Optim.	Unopt.	Optim.
Csim	34.1	61.4	22.6	22.6	43.4	16.0
DWF	29.9	91.4	11.7	1.2	58.4	7.5
Mp3d	43.6	78.1	17.2	0.0	39.2	21.9
LocusRoute	13.7	50.2	15.3	22.5	71.0	27.3
Maxflow	36.1	61.1	6.2	6.9	57.7	32.0
Mincut	11.9	33.7	21.5	37.3	66.7	29.1
AVERAGE	28.2	62.6	15.7	15.1	56.1	22.3

Table 9: Decomposition of the data reference stream for 32 processes.

Application	Shared Refs. / Total Data Refs. (%)		Private Global Refs. / Total Data Refs. (%)		Private Local Refs. / Total Data Refs. (%)	
	Unopt.	Optim.	Unopt.	Optim.	Unopt.	Optim.
Csim	35.2	61.3	23.1	23.2	41.7	15.4
DWF	28.2	85.1	11.5	2.1	60.3	12.8
Mp3d	43.5	77.8	17.2	0.0	39.3	22.3
LocusRoute	13.9	50.4	15.4	23.5	70.7	26.1
Maxflow	37.0	61.7	6.5	6.6	56.6	31.7
Mincut	11.8	33.7	21.5	37.3	66.6	29.1
AVERAGE	28.3	61.7	15.9	15.4	55.8	22.9

References

[1] D. R. Cheriton, H. A. Goosen, and P. D. Boyle, "Multi-Level Shared Caching Techniques for Scalability in VMP-MC," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 16–24, June 1989.

[2] J. Elder, A. Gottlieb, C. K. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, "Issues Related to MIMD, Shared-Memory Computers: The NYU Ultracomputer Approach," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 126–135, June 1985.

[3] J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 422–431, June 1988.

[4] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 148–159, May 1990.

[5] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," in *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 764–771, 1985.

[6] A. W. Wilson, "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 244–

252, June 1987.

[7] A. Agarwal and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH," in *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 215–225, May 1988.

[8] S. J. Eggers and R. H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257–270, April 1989.

[9] W. D. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 243–256, April 1989.

[10] R. L. Lee, P. C. Yew, and D. H. Lawrie, "Multiprocessor Cache Design Considerations," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 253–262, June 1987.

[11] A. J. Smith, "Line (Block) Size Choice for CPU Caches," in *IEEE Trans. on Computers*, pp. 1063–1075, September 1987.

[12] A. J. Smith, "Cache Memories," in *Computing Surveys*, pp. 473–530, September 1982.

[13] F. Irigoin and R. Triolet, "Supernode Partitioning," in *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 319–329, January 1988.

[14] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," in *Proceedings ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pp. 30–44, June 1991.

[15] F. J. Carrasco, "A Parallel Maxflow Implementation." CS411 Project Report, Stanford University, March 1988.

[16] J. A. Dykstal and T. C. Mowry, "MINCUT: Graph Partitioning Using Parallel Simulated Annealing." CS411 Project Report, Stanford University, March 1989.

[17] A. Galper, "DWF." CS411 Project Report, Stanford University, March 1989.

[18] J. D. McDonald and D. Baganoff, "Vectorization of a Particle Simulation Method for Hypersonic Rarified Flow," in *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[19] J. Rose, "LocusRoute: A Parallel Global Router for Standard Cells," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 189–195, June 1988.

[20] L. Soule and A. Gupta, "Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*,

pp. 81–86, June 1989.

[21] E. Lusk, R. Overbeek, *et al.*, *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, Inc., New York, NY, 1987.

[22] H. Davis, S. Goldschmidt, and J. Hennessy, "Multiprocessing Simulation and Tracing Using Tango," in *Proceedings of the 1991 International Conference on Parallel Processing*, vol. II, pp. 99–107, August 1991.

[23] F. Baskett, T. Jermoluk, and D. Solomon, "The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second," in *Proceedings of the 33rd IEEE Computer Society International Conference - COMPCON 88*, pp. 468–471, February 1988.

[24] M. S. Papamarcos and J. H. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348–354, June 1984.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[26] G. Kane, *MIPS RISC Architecture.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

[27] M. D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance," Tech. Rep. UCB/CSD 87/381, University of California, Berkeley, November 1987.

[28] S. Przybylski, M. Horowitz, and J. Hennessy, "Performance Tradeoffs in Cache Design," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 290–298, May 1988.