Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates

Josep Torrellas, Monica S. Lam, and John L. Hennessy
Computer Systems Laboratory
Stanford University, CA 94305

1 Introduction and Approach to the Problem

Shared-memory multiprocessors with hardware coherent caches [1,4,6,7,10] are attractive in that they can be programmed relatively easily and that they allow the program to take advantage of the caching of shared data. Recent studies have shown that *shared data* may exhibit a high cache miss rate, especially if the parallelism is fine-grained [3,9]. In large multiprocessors with considerable memory access latencies, a high cache miss rate may lead to poor machine performance.

While optimizations that change the parallel algorithm or the data structures can greatly modify the cache miss behavior, these optimizations often require application knowledge and user intervention. Here we focus only on the optimization of repositioning shared data at the cache block level. The changes are all transparent to the programmer. This approach is motivated by the fact that cache misses on shared data are often concentrated in small sections of the data space. Therefore, localized optimizations can potentially generate most of the desired effects. Figure 1 shows the cache misses on the shared data structures for one application. The average number of misses per byte for each data structure is computed by dividing the total misses on the data structure by the size of the data structure. The leftmost peak corresponds to an area of scalar variables and some small arrays.

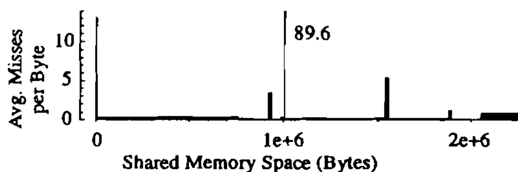

Figure 1: *Distribution of the cache misses along the shared data space for the Csim application. Within each data structure, the misses per byte are average values. The execution corresponds to 16 processors, 4 word cache blocks and the architecture independent environment of Section 4.*

The effect of the cache blocks in multiprocessors is different from that in a uniprocessor. While the data cache miss rate of uniprocessors tends to drop substantially with the cache block size, trace simulations of parallel applications indicate that the data miss rate curve may drop slowly, remain relatively flat, or even increase with block size increases [3,9]. The factor in determining the new behavior is data sharing. While prefetching provided by the blocks may bring in useful data, it may also remove useful data from other caches and create false sharing. As a result, miss rates can increase with the block size. Data placement optimizations for multiprocessors are therefore different from uniprocessor versions. They can be effective by increasing the success of prefetching data or reducing misses due of false sharing.

Reference traces of programs compiled with a compiler that performs register allocation and other conventional optimizations on private data, show that sharing is more important than suggested in earlier studies. Before studying the data placement problem, we present data on the effect of an optimizing compiler. In the rest of the paper, we use only compiler optimized code to evaluate the data placement improvements.

The methodology of the research is as follows. We first develop a model that makes it possible to quantitatively measure the prefetching and false sharing of individual data. With this model, the characteristics of some real-life applications are studied using compiler optimized RISC code traces. The traces are generated by Tango [5], a program that simulates a multiprocessor environment. The traces correspond to the C applications in Table 1 for 16 and 32 processors, and include their complete parallel section, from when all processes are spawned after initialization, until the end of the program. The traces contain only application references, they assume no process migration, and range in size from 7.5 to over 25 million *data* references.

Table 1: *The application set.*

Application	Description (Shared Data Space in Mbytes)
Csim	Chandy-Misra logic gate simulator (2.83).
DWF	Performs string pattern matching (2.10).
Mp3d	3-D particle simulator for rarefied flow (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).

The applications represent a variety of engineering algorithms. Csim, Mp3d, and LocusRoute are between 1000 and 6000 lines of code. The other three programs, DWF, Maxflow, and Mincut implement several commonly used parallel algorithms and are less than 1000 lines of code. The applications are written in such a way that they can run on any number of processors. The Argonne National Laboratory macro package [2] is used to provide synchronization and sharing primitives. The traces contain *spin-locking free* locks, barriers, and distributed loop control variables.

The measurements on these applications identify data that exhibit high cache miss rates or high degrees of false sharing. By correlating these measurements with the programs, we identify several simple optimizations. Most of these optimizations can be applied to all programs without having to measure the data through traces.

To evaluate the optimizations, two sets of trace driven simulations are performed. The first one is an architecture independent simulation where we assume that caches are infinite, and that cache hits and misses all take the same amount of time. This experiment shows the effects of the optimizations on shared data in isolation. A second set of simulations is performed using the characteristics of a 16 processor machine. This experiment, run using large data caches (256 kbytes) with ordinary block sizes (4 and 16 words) shows that data miss rates are reduced by up to an absolute 1.5%. While small, this improvement is attractive because it does not require any programmer intervention.

2 Effects of the Optimizing Compiler on the Frequency of Sharing

This section shows that the frequency of shared data references in a processor reference stream is higher when the code is generated by an optimizing compiler. By allocating commonly used private data in registers, and performing other optimizations on private data, many of the memory references to private data are removed. While removing these references may have only a small effect on the number of misses in a large cache, the processor runs faster after optimization for the same number of shared references. The result is an intensification of the shared data access bottleneck. The remaining sections of this paper use the *optimized trace* to evaluate the placement optimizations.

Figure 2 compares the data reference streams of the applications with and without the optimizing compiler effects. The target architecture is the MIPS R2000, which has 32 integer and 16 double precision floating point registers. The optimizations applied include global register allocation and other traditional global optimizations. All data in the shared space is declared to be volatile, and therefore not register allocated or optimized. We divide private data references into *local* and *global*. Local data are the variables declared within procedures. Global data include static data set up by the master process for the slaves, and the pointer to the shared data space, which we allocate to a register.

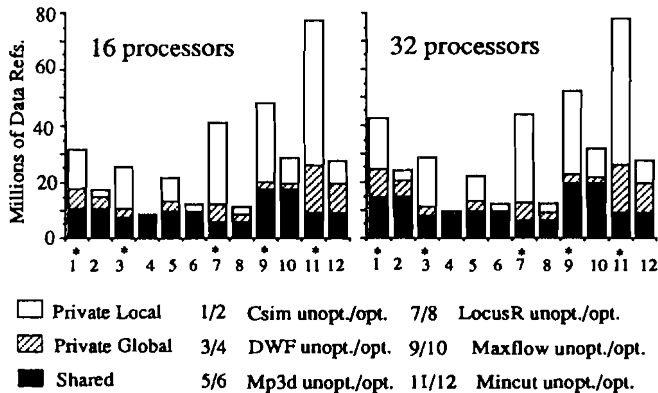

Figure 2: *Analysis of the overall data reference stream. For space reasons, some unoptimized traces were not completed (bars with a star). In these cases, the total number of references is estimated assuming the same data distributions as the ones up to the point where the trace was interrupted.*

Figure 2 shows that, for both 16 and 32 processors, all applications experience a remarkable decrease in the number of private references. Most of the reduction is due to register allocating local variables. The register allocation of the global pointer to the shared data space is responsible for most of the references saved among the global variables. This result suggests that all future studies of multiprocessor programs must be based on optimized code.

3 Optimizing the Placement of Shared Data in Cache Blocks

3.1 A Model of Sharing

Sharing can be classified into *true sharing* and *false sharing*. The sharing of the same memory word by different processors is called true sharing. This is the only type of sharing present in caches with single-word blocks. In caches with multi-word blocks, two processors may share a block because they need to access two different words that happen to be in the same block. This kind of sharing is called false sharing. While true sharing is intrinsic to a particular memory reference stream of a program, false sharing is a result of collocation of data in the same cache block. It depends on the cache block size and the particular placement of data in memory.

For infinite caches, misses beyond the cold start can be classified into *true sharing misses* and *false sharing misses* [9]. A miss is a true sharing miss when a processor misses because the word was previously used by another processor. In single-word cache blocks, every incidence of true sharing causes a true sharing miss, and all misses (not counting the cold misses) are true sharing misses. In multi-word blocks, some of the true sharing misses present in the cache with single-word blocks may be removed because of prefetching. However, prefetching may also induce false sharing by removing some data in caches that may need it before any other cache. The induced cache miss is a false sharing miss. Therefore, the effect of prefetching can be measured by the number of false sharing misses present in the multi-word block cache, and the number of true sharing misses saved in the multi-word block cache [9].

	References	Misses
EXAMPLE I	Qa Pb Qa	3
EXAMPLE II	Qa Pa Pb	2
EXAMPLE III	Qa Pb Pa	2

Figure 3: *Data sharing interacts with prefetching provided by the cache block. Pa means that processor P writes to word a.*

Let us illustrate the concepts of true and false sharing misses by some examples. Figure 3 shows simple reference streams that contain only writes. Memory words a and b belong to the same cache block and are initially owned by processor P. In all these examples, the first access by processor Q causes a true sharing miss. In example I, processors P and Q alternate writing into different words, creating two false sharing misses. In both examples II and III, processor P needs to access both words a and b. In example II, the miss experienced when accessing a is a result of true sharing between processors P and Q. Although word b is removed from processor P's cache inadvertently by processor Q's access, no false sharing miss results because the data is successfully prefetched by P's access to variable a. In general, when servicing either a true or a false sharing miss, prefetching can eliminate a potential sharing miss, true or false. Example III shows how a potential true sharing miss is eliminated by a false sharing miss and subsequent prefetching.

3.2 Measurements

The discussion above suggests that shared data misses can be quantified as follows. Cold misses are simply misses to memory words referenced by a processor for the first time. True and false sharing misses can be measured by comparing two simulations running in lock-step and driven by the same interleaving of memory references on infinite caches [9]. One simulates caches with single-word blocks and the second one simulates caches with multi-word blocks. Shared data misses in the multi-word simulation that do not have an equivalent in the single-word simulation are all false sharing misses. Shared data misses that are present in the single-word simulation but not in the multi-word case are cold and true sharing misses eliminated by prefetching.

To aid in our study of data placement optimizations, each shared memory word is characterized in the following way: (1) Degree of true sharing, measured by the misses on the word beyond the cold start in the single-word block simulation. (2) False sharing misses on the word. (3) Cold and true sharing misses on the word eliminated by prefetching. This quantity minus the false sharing misses is the net benefit the word receives from being in the block with other words. (4) Number of writes to the word. False sharing in a block may be produced by words with a high degree of true sharing or by words that are frequently written to. If any of these two metrics for a word is higher than 0.1% of the program misses, the word is considered *active*.

3.3 Data Placement Optimizations

It is well known that synchronization variables create hot spots and should be treated specially. We assume that the compiler has already allocated each synchronization variable in a different and empty cache block. The optimizations considered in this paper are as follows:

Opt1: Place scalar variables that exhibit false sharing in different blocks. This optimization is applied to a block of scalar variables when the net increase in misses due to prefetching exceeds 0.5% of the program misses.

Each active variable is removed from the block and allocated in a cache block by itself.

Opt2: Place active scalars that are protected by a lock in the same block as the lock. In this way, the data is prefetched when the lock is accessed.

Opt3: Allocate shared space requested by different processes from different heap regions. The observation is that a slave process is likely to access the shared space that it requests itself. Especially when small amounts of space are requested, the allocated space may share the same cache block with that allocated by another process, and thus lead to false sharing. An alternative is to allocate only block-aligned space, but this may be wasteful if a small amount of space is requested at a time.

Opt4: Position an array of records so that the number of different blocks that the average record spans is minimized. This optimization is attempted when the number of words in the record and the number of words in the cache block have a greatest common divisor (GCD). The optimization tries to exploit prefetching of as much of the rest of a record as possible when one word of the record is accessed. False sharing may also be reduced. The array is positioned starting at a block boundary or at a distance from a block boundary equal to the mentioned GCD or a multiple of it, whichever wastes less space.

Opt5: Expand records in an array to minimize the sharing of a cache block by different records if cache misses can be reduced and the space increase is tolerable. False sharing usually arises from having two records share the same cache block; successful prefetching, however, may occur within a record or across records. Therefore, if multi-word simulation measurements indicate that there is much false sharing and little gain in prefetching, then consider expansion; or if there is a large prefetching advantage and a small degree of false sharing, then do not apply the optimization. When both false sharing misses and prefetching savings are of the same order of magnitude, we may assume that the prefetching succeeds within a record rather than across records, and possibly apply the optimization.

The *Opt2* and *Opt4* optimizations attempt to reduce cold and true sharing misses by increasing the effectiveness of prefetching. *Opt1*, *Opt3*, *Opt5*, and also *Opt4* try to minimize false sharing. Since both false sharing and prefetching are the result of a particular data placement in memory, changing one usually affects the other. In particular, *Opt1* and *Opt5* may also reduce the effectiveness of prefetching in eliminating cold and true sharing misses, because data is expanded in size. Furthermore, large data expansions may increase the working set of a program and increase the conflict misses in a small cache. Therefore, we should restrict the optimizations to those that cause little data size increase.

Table 2: *Results of the placement optimizations on the 16 processor architecture independent model.*

Application (Block Size in Words)	Reduction in Shared Data Misses		Increase in Shared Data Space	
	Relative (% of Misses)	Absolute (Thousands)	Relative (% of Space)	Absolute (Kbytes)
Csim(4)	7.9	60.6	0.0	0.4
Csim(16)	6.6	39.3	0.1	1.9
DWF(4)	0.6	3.1	0.0	0.0
DWF(16)	1.0	4.6	0.0	0.2
Mp3d(4)	0.4	20.6	0.3	4.8
Mp3d(16)	0.1	5.5	0.0	0.5
LocusRoute(4)	10.2	45.3	0.0	0.5
LocusRoute(16)	28.7	57.5	0.1	1.6
Maxflow(4)	8.9	198.9	0.6	1.6
Maxflow(16)	14.2	235.3	0.6	1.6
Mincut(4)	19.7	229.6	72.3	4.0
Mincut(16)	8.9	153.1	0.0	0.0

4 Architecture Independent Evaluation

This section evaluates the effectiveness of the optimizations in an architecture independent fashion: caches are infinite; memory references all take the same time, reads or writes, hits or misses; every instruction executes in one cycle. This approach is taken to eliminate any architecture dependence. The Illinois cache coherence protocol [8][1] is used. Synchronization variables are assumed to be in different cache blocks already, and are not repositioned.

The fraction of shared data misses eliminated by the optimizations for machines with 16 and 32 processors is shown in Tables 2 and 3, respectively. The numbers show a large variation across applications, and are significant for many of the applications. The fraction of misses eliminated is usually below 20% but can reach up to 40% in the case of 32 processors. The initial misses include cold start misses, which are less amenable to optimization.

Table 3: *Results of the placement optimizations on the 32 processor architecture independent model.*

Application (Block Size in Words)	Reduction in Shared Data Misses		Increase in Shared Data Space	
	Relative (% of Misses)	Absolute (Thousands)	Relative (% of Space)	Absolute (Kbytes)
Csim(4)	15.6	196.2	0.0	0.9
Csim(16)	11.5	110.3	0.1	3.9
DWF(4)	0.6	5.5	0.0	0.0
DWF(16)	1.1	9.0	0.0	0.2
Mp3d(4)	0.4	24.3	0.3	4.8
Mp3d(16)	0.2	13.3	0.0	0.5
LocusRoute(4)	15.5	92.5	0.0	0.5
LocusRoute(16)	41.6	138.5	0.2	3.1
Maxflow(4)	10.7	397.0	0.6	1.6
Maxflow(16)	14.7	455.6	0.6	1.6
Mincut(4)	22.0	394.1	72.3	4.0
Mincut(16)	8.8	190.9	0.0	0.0

The relative reductions in cache misses are not correlated with the cache block size. A larger cache block size means that there is more opportunity for false sharing to arise among scalars and small data structures, thus the relative miss reductions may increase with block size. On the other hand, the larger cache block size means that it is more costly to expand records, thus making some data expansion optimizations infeasible. Also, the longer cache block may already deliver part of the benefits obtained from prefetching, rendering optimizations to enhance prefetching less effective.

When the number of processors is increased, there are more cache misses, and the data placement optimizations can also eliminate more misses. The relative miss reductions are higher for 32 processors than for 16 processors.

The space requirements of the optimizations are small, usually in the 2 kbyte range. The relative increase in shared space is almost always insignificant, unless the shared space is small originally. More miss elimination would result from more data expansion.

Figure 4 shows the effect of the optimizations on the shared data miss rates. The shared miss rates of the programs before and after optimizing are shown as the central and rightmost bars in the figures respectively. The effects on false sharing misses, and cold and true sharing misses are also shown. The miss rates of the original programs, without necessarily allocating synchronization variables to different cache blocks, are shown as the leftmost bars. The importance of such an optimization is obvious, considering that the simulations already assume spin-locking free synchronization.

The fraction of each type of miss in the programs shows a large variation across applications. For example, false sharing misses constitute an important percentage of the total misses in Maxflow, but they are less significant in Mp3d.

The optimizations are more successful in reducing false sharing misses than in reducing cold and true sharing misses, although the effects differ across applications. Almost all false sharing is removed in LocusRoute, as well as in the 4-word cache block case of Mincut. Some limited form of success is observed in Csim and Maxflow, where the remaining false sharing ranges from 60 to 80% of the old false sharing. The reduction of false sharing in Mincut is accompanied by a small increase in cold and true sharing misses, illustrating the interaction between the two. The number of remaining cold and true sharing misses is about 90 to 100% of the old number of these misses, with some exceptions that perform worse. More false sharing misses

[1] A request for ownership on a shared block is considered a cache miss since, in the architecture studied in Section 5, it has the same traffic and timing behavior as a regular miss.

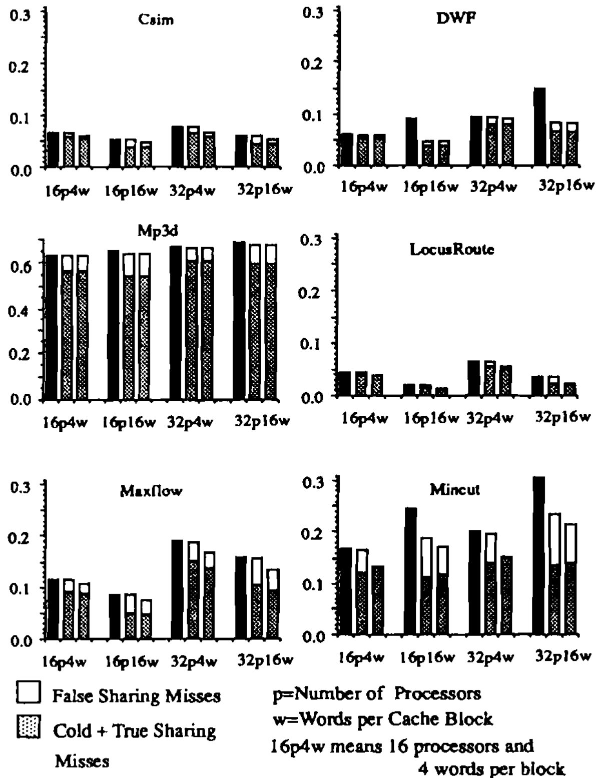

Figure 4: *Shared data miss rates. For each set of three bars, the central and the rightmost ones correspond to before and after the optimizations respectively. The leftmost bar shows the miss rates of the original program without necessarily allocating synchronization variables to different cache blocks.*

tend to be eliminated with larger blocks, since more false sharing conflicts are likely to occur. The optimizations that eliminate cold and true sharing misses tend to be less effective with 16-word cache blocks because some of the benefits of prefetching are already enjoyed. Both types of misses tend to experience higher reductions as the number of processors increases. The effectiveness of each optimization is shown in Table 4. Most of the misses eliminated by *Opt1* can be attributed to repositioning 2 to 3 active scalars.

A large fraction of the cache misses still remains after optimization. Some of the false sharing misses can be removed if the data caches are large enough to support the expansion optimization. The remaining misses are dominated by cold and true sharing misses, suggesting that further optimizations that increase the effectiveness of prefetching may be useful. Data placement optimizations for a cache can only reduce the number of true sharing misses by a fraction equivalent to the size of a cache block. Further reductions of the degree of true sharing can be achieved only by changing the parallel algorithm or changing the assignment of computation to processors.

5 Architecture Dependent Evaluation

To observe the effect of these optimizations on one architecture, the experiment was rerun substituting the idealized machine model with that of a bus-based multiprocessor. The simulated architecture has a memory system bandwidth and latency, and consistency support loosely based on an SGI 4D-MP multiprocessor [1]. Unlike the SGI machine, the simulated one has sixteen processors, each of which has one 256 kbyte direct mapped data cache, and an execution rate of one instruction per cycle. Synchronization is supported in the main bus as explained in Section 1 (no spin-locking). The zero-contention latency of a memory access is $19 + 0.75 * B$ cycles, where B is the cache block size in words. This is a large number, as is typical in machines with high performance processors. Bus transactions hold the bus for $3 + 0.75 * B$ cycles. The programs are executed twice: the first run warms up the caches, so that when the measurements are taken in the second run, the results obtained are similar to those of a much longer run.

Table 4: *Effectiveness of each data placement optimization. An X is shown for the optimizations that eliminate more than 2% of the shared misses in any of the four experiments run per application.*

Application	*Opt1*: Scalar Expan.	*Opt2*: Protection by Lock	*Opt3*: Heap Alloc.	*Opt4*: Array Alignm.	*Opt5*: Record Expan.
Csim	X			X	X
DWF					
Mp3d					
LocusRoute	X		X		X
Maxflow	X	X			X
Mincut	X	X			X

Table 5: *Results of the placement optimizations on the 16 processor architecture dependent model.*

Application	Block Size (Words)	Execution Speed Increase (%)	Overall Data Miss Rate Unopt. - Opt. (%)	Opt. (%)
Csim	4	10	0.8	5.1
	16	6	0.4	4.4
DWF	4	0	0.0	2.6
	16	-1	-0.1	2.0
Mp3d	4	0	0.1	46.4
	16	0	0.1	46.2
LocusRoute	4	11	1.5	4.0
	16	26	1.2	1.8
Maxflow	4	16	1.1	7.2
	16	24	1.1	5.1
Mincut	4	18	1.2	5.4
	16	16	0.9	5.5

Table 5 shows the overall data miss rates and the reduction in data miss rates achieved by the optimizations. The miss rate includes both shared and private data for sixteen processor runs. Private data has a small miss rate, since the cold start of the caches has been avoided. The optimizations reduce the overall miss rate by up to 1.5%.

The execution speed-up resulting from these optimizations is also shown in the table. The performance improvements reach 25%. This result is highly dependent on the architecture considered, and has to be interpreted with care. Sixteen processors accessing a common bus may lead to considerable contention in some applications. On the other side, replacing the bus with another interconnection network may reduce the contention, but may also increase the overall latencies.

Reducing cache misses is more important with larger cache blocks, and with more processors in the system. Larger cache blocks mean that each miss would incur a higher volume of traffic. More processors mean that the same problem will execute faster, thus exacerbating network contention.

6 Effectiveness of Optimizations without Program Information

In choosing the optimizations in the experiments above, we take advantage of the detailed information obtained by profiling the program. In this section, we investigate if it is possible to have general and effective optimizations that do not rely on such costly information.

Opt1': Always place each shared scalar variable in a different cache block. This approach has practically the same effect as moving only active scalar variables since, in relatively large caches, the advantage of prefetching scalars

is minor. Most programs have a small number of shared scalars (the number of shared scalars in the programs studied ranged from 5 to 50). For programs with many shared scalars and large cache blocks, much space may be wasted. However, since only a fraction of the scalars is accessed frequently, little negative effect is expected.

Opt2: In practice, the optimization of allocating data with their locks may not be implementable because large multiprocessors are likely to manage synchronization variables differently from regular data.

Opt3 and *Opt4*: The optimizations of allocating shared data from a process' own heap space, and positioning the records of an array to enhance prefetching can be applied at all times since their cost is low.

Opt5: It is not a good heuristic to expand all short arrays. It wastes space, and the net effect on the cache misses can be either positive or negative. We leave it up to the programmer to pad the data structure if so desired.

The compiler and run time system can therefore incorporate *Opt3*, *Opt4*, and *Opt1'*. The cumulative effect of these optimizations on the idealized architecture is shown in Table 6. These numbers indicate that, for many of the programs, much of the effect of the previous optimizations can be obtained by these general optimizations. Moreover, the increase in data space, both absolute and relative, remains small.

Table 6: *Effect of simple program independent optimizations. The numbers in parenthesis show the percentage of the misses eliminated in Tables 2 and 3 that can be removed by the program independent optimizations.*

Application	Shared Misses Eliminated (%)			
	16 Proc. 4 Words / Block	16 Proc. 16 Words / Block	32 Proc. 4 Words / Block	32 Proc. 16 Words / Block
Csim	5.1 (65)	3.2 (48)	12.8 (82)	8.0 (70)
DWF	0.6 (100)	1.0 (100)	0.6 (100)	1.1 (100)
Mp3d	0.0	0.1	0.0	0.1
LocusRoute	8.1 (79)	24.2 (84)	13.4 (86)	36.3 (87)
Maxflow	1.5 (17)	2.0 (14)	1.7 (16)	2.0 (14)
Mincut	4.7 (24)	4.1 (46)	6.7 (30)	5.5 (63)
Application	**Shared Space Increase (Kbytes)**			
	16 Proc. 4 Words / Block	16 Proc. 16 Words / Block	32 Proc. 4 Words / Block	32 Proc. 16 Words / Block
Csim	0.6	2.9	0.6	2.9
DWF	0.1	0.6	0.1	0.6
Mp3d	0.4	1.9	0.4	1.9
LocusRoute	0.6	2.6	0.6	2.6
Maxflow	0.1	0.4	0.1	0.4
Mincut	0.0	0.3	0.0	0.3

7 Conclusions

After pointing out the need to use address traces generated using an optimizing compiler, this paper proposes and evaluates simple optimizations of the placement of shared data to reduce cache miss rates in multiprocessors. The optimizations try to increase the efficiency of prefetching provided by the cache blocks, and to reduce false sharing. The optimizations proposed are shown to have a small but still significant impact on the miss rates of large data caches for some of the applications studied. We expect that these optimizations will be more important in future large-scale multiprocessors. First, communication latencies are likely to be higher, increasing the penalty of each cache miss. Second, with larger numbers of processors, the ratio of misses eliminated over program execution time will possibly be higher, increasing the impact of the optimizations. We recommend that the generalized optimizations, optimizations requiring no profiling analysis, be incorporated into compilers and run-time systems. The optimizations are cheap to perform and have been shown to offer much of the benefits of the knowledge-intensive optimizations.

8 Acknowledgements

We thank S. Goldschmidt, B. Bray, H. Davis, and the referees for useful input to this work. We also thank the authors of the applications: F. Carrasco, J. Dykstal, A. Galper, J. McDonald, T. Mowry, J. Rose, and L. Soule. This work 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.

References

[1] 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*, pages 468–471, February 1988.

[2] J. Boyle, E. Lusk, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.

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

[4] Encore Computer Corporation. *Multimax Technical Summary*. 1986.

[5] S. R. Goldschmidt and H. Davis. Tango Introduction and Tutorial. Technical Report CSL-TR-90-410, Stanford University, January 1990.

[6] 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*, pages 422–431, June 1988.

[7] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. In *Proceedings of the 35th IEEE Computer Society International Conference - COMPCON 90*, February 1990.

[8] 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*, pages 348–354, June 1984.

[9] J. Torrellas, M. S. Lam, and J. L. Hennessy. Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors. Technical Report CSL-TR-90-412, Stanford University, February 1990.

[10] A. W. Wilson. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.