Maximizing Multiprocessor Performance with the SUIF Compiler

Mary W. Hall† Jennifer M. Anderson Saman P. Amarasinghe
Brian R. Murphy Shih-Wei Liao Edouard Bugnion
Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

† *USC Information Sciences Institute*
Marina del Rey, CA 90292

Abstract

This paper presents an overview of the SUIF compiler, which automatically parallelizes and optimizes sequential programs for shared-memory multiprocessors. We describe new technology in this system for *locating coarse-grain parallelism* and for *optimizing multiprocessor memory behavior* essential to obtaining good multiprocessor performance. These techniques have a significant impact on the performance of half of the NAS and SPECfp95 benchmark suites. In particular, we achieve the highest SPECfp95 ratio to date of 63.9 on an eight-processor 440MHz Digital AlphaServer.

1 Introduction

Affordable shared-memory multiprocessors can potentially deliver supercomputer-like performance to the general public. Today, these machines are mainly used in a multiprogramming mode, increasing system throughput by running several independent applications in parallel. The multiple processors can also be used together to accelerate the execution of single applications. Automatic parallelization is a promising technique to allow ordinary sequential programs to take advantage of multiprocessors [4, 6, 9, 11].

Multiprocessors present more difficult challenges to parallelizing compilers than vector machines which were their initial target. Effective use of a vector architecture involves parallelizing repeated arithmetic operations on large data streams (*e.g.,* innermost loops in array-oriented programs). On a multiprocessor, however, parallelizing innermost loops typically does not provide sufficient *granularity of parallelism*—not enough work is performed in parallel to overcome the overhead of synchronization and communication between processors. To utilize a multiprocessor effectively, the compiler must exploit *coarse-grain parallelism*, locating large computations that can execute independently in parallel. Multiprocessor systems also have more complex memory hierarchies than typical vector machines. Modern multiprocessors contain multiple levels of caches in addition to the shared memory. Locating parallelism is but the first step in producing efficient multiprocessor code; it is critical to make effective use of the memory hierarchy to achieve high performance.

These additional challenges often prevented early parallelizing compilers from being effective for multiprocessors. Consequently, parallel programming today predominantly requires that the programmer explicitly manage both the parallelism and the memory use of the application. Developing an efficient parallel program in this way requires a highly knowledgeable programmer. Moreover, explicit parallel programming is tedious and error prone, and the resulting programs are only optimized for a specific machine.

This research was supported in part by the Air Force Material Command and DARPA contracts F30602-95-C-0098, DABT63-95-C-0118 and DABT63-94-C-0054, NASA's HPCC program, an NSF Young Investigator Award, an NSF CISE postdoctoral fellowship, and fellowships from AT&T Bell Laboratories, DEC Western Research Laboratory, Intel Corporation and NSF. Jennifer Anderson is currently a researcher with Digital Equipment's Western Research Lab.

This paper presents an overview of the automatic parallelization techniques in the Stanford SUIF compiler. We describe two techniques essential to obtaining good multiprocessor performance for array-based numerical programs: locating coarse-grain parallelism and managing multiprocessor memory use. We present performance measurements for two complete benchmark suites, the NAS and the SPECfp95 benchmarks. Overall, the results for these scientific programs are promising. The compiler yields speedups on more than three-fourths of the programs, and has obtained the highest SPECfp95 ratio reported to date, indicating that the compiler is also able to achieve efficient absolute performance.

2 Finding Coarse-Grain Parallelism

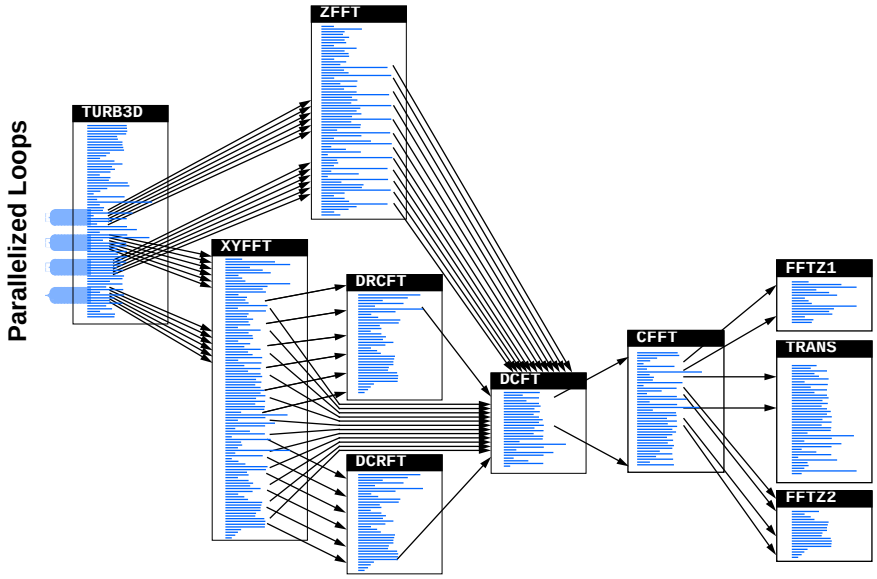
Multiprocessors work best when the individual processors have large units of independent computation. Finding such coarse-grain parallelism is challenging. First, it is necessary to find available parallelism across procedure boundaries. Furthermore, the original computations may not be parallelizable as given but require some transformations to parallelize. For example, experience in hand parallelization suggests that it is often necessary to replace global arrays by *private* versions on different processors. Sometimes the computation must be restructured—for example, a sequential accumulation replaced by a parallel *reduction* operation.

It takes a large suite of robust analysis techniques to successfully locate coarse-grain parallelism. To cope with the complexity involved in building such a system into SUIF, we rely on general and uniform frameworks. We have automated the analysis to privatize arrays and to recognize reductions to both scalar and array variables. All the analysis techniques in our compiler operate across procedure boundaries seamlessly. There are three major components in the analysis to detect coarse-grain parallelism:

Scalar analyses. An integrated pass analyzes scalar variables in the programs. It detects parallelism among operations with scalar variables using techniques such as data dependence analysis, scalar privatization analysis, and reduction recognization. It also derives symbolic information on these scalar variables that is useful to the array analysis pass, including constant propagation, induction variable recognition and elimination, recognition of loop-invariant computations, and symbolic relation propagation [8, 10].

Array Analyses. An integrated array analysis uses a unified mathematical framework based on linear algebra and integer linear programming [9]. The analysis applies the basic data dependence test to determine if accesses to an array can refer to the same *location*. To support array privatization, it also finds array data-flow information that determines if array elements used in an iteration refer to the *values* produced in a previous iteration. It also recognizes commutative operations on sections of an array and transforms them into parallel reductions. The reduction analysis is powerful enough to recognize commutative updates of even indirectly accessed array locations, allowing parallelization of sparse computations. All these analyses are formulated in terms of integer programming problems on systems of linear inequalities representing the data accessed. These inequalities are derived from loop bounds and array access functions. Optimizations to speed up common cases are implemented to reduce the compilation time.

Interprocedural Analysis Framework. To manage the software engineering complexity, all of the analyses are implemented using a uniform interprocedural framework. The framework uses a true interprocedural analysis [11], which is more efficient than the more common technique of *inline substitution* [4]. Inline substitution replaces each procedure call with a copy of the called procedure, then analyzes the expanded code in the usual intraprocedural manner. Inline substitution is not practical for large programs, as program size can increase to an unmanageable extent. Our technique analyzes only a single copy of each procedure, capturing its side effects in a function. This function is then applied at each call site to produce precise results. When necessary, the algorithm selectively clones a procedure so that code can be optimized or parallelized under different calling contexts (such as when different constant values are passed to the same formal parameter), thus achieving the full advantages of inlining without expanding the code indiscriminately.



Our analyses can successfully parallelize loops spanning hundreds of lines of code and numerous procedures, as shown by the above code segment from the SPECfp95 benchmark `turb3d`. The boxes in the figure represent procedure bodies and the lines connecting them represent procedure calls. The main computation is a series of four loops to compute three-dimensional fast Fourier transforms (FFTs). Using interprocedural scalar and array analyses, the SUIF compiler determines that these loops are parallelizable. Each loop contains over 500 lines of code spanning eight or nine procedures with up to 42 procedure calls. Note that if this program had been fully inlined, the loops presented to the compiler for analysis would have each contained over 86,000 lines of code.

Figure 1: Illustration showing scope of coarse-grain parallelism analysis.

3 Memory Optimizations

Numerical applications on high-performance microprocessors are often memory bound. One or more levels of caches are commonly used to bridge the gap between processor and memory speeds, but it is still not uncommon to find a processor wasting half its time stalled on memory accesses because of frequent cache misses. This memory bottleneck is further exacerbated on multiprocessors, as they have increased need for memory traffic and more contention on the memory bus.

To optimize for memory subsystem behavior, the compiler must address the following four issues that affect the behavior of the caches:

Communication. Processors on a multiprocessor system communicate data through accesses to the same memory location. Coherent caches typically keep the data consistent by causing accesses to data written by another processor to miss in the cache. Such misses are called *true sharing misses*.

Limited Capacity. Numeric applications tend to have large working sets, which typically exceed the capacity of the cache. These applications often stream through large amounts of data before re-using any of the data, resulting in poor *temporal locality* and a large number of *capacity misses*.

Limited Associativity. Caches typically have a small *set associativity*, that is, each memory location can only map to one or just a few locations in the cache. *Conflict misses* can occur even when the application's working set is smaller than the cache if the data are mapped to the same cache locations.

Large Line Size. Data in a cache are transferred in fixed-size units called *cache lines*. Applications that do not use all the data in a cache line, e.g. when accessing data with a non-unit stride, incur more misses and are said to have poor *spatial locality*. On a multiprocessor, large cache lines can also lead to cache misses that occur when different processors use different parts of the same cache line. Such misses are called *false sharing misses*.

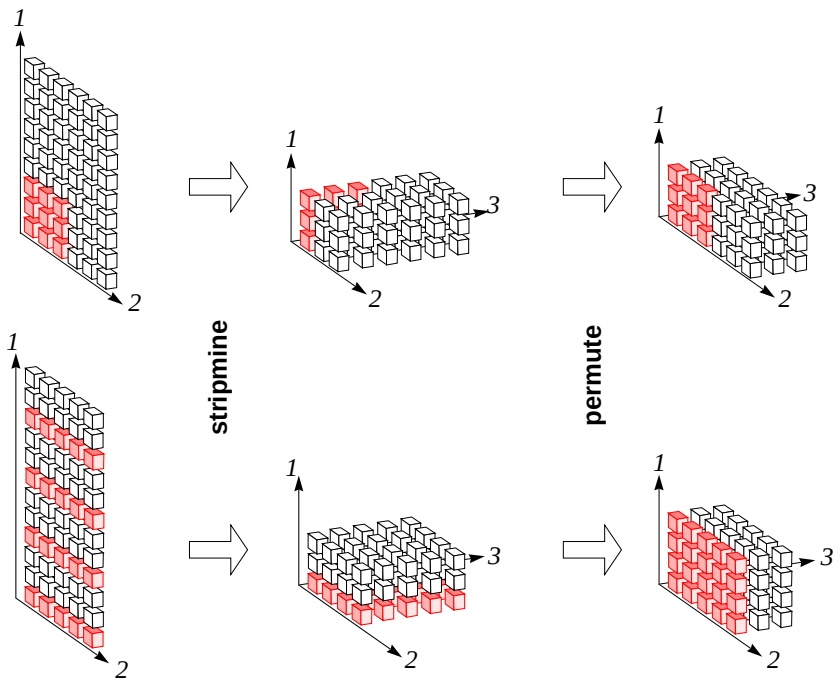
The compiler tries first to eliminate as many cache misses as possible, and then to minimize the impact of any that remain. We reduce the problem of eliminating cache misses due to the four factors above to two subproblems: (1) ensuring that processors re-use the same data as much as possible and (2) making the data accessed by each processor contiguous in the shared address space. The techniques to address each subproblem are discussed below. Finally, to tolerate the latency of remaining cache misses, the compiler uses *compiler-inserted prefetching* to move data into the cache before it is needed.

Improving Processor Re-use of Data. The compiler reorganizes the computation so that each processor re-uses the same data as much as possible[2, 3, 12]. This reduces the working set on each processor thus minimizing capacity misses; it also reduces communication between processors thus minimizing true sharing misses. To achieve this goal, the compiler uses the technique of *affine partitioning*. The technique analyzes the reference patterns in the program to derive an affine mapping (linear transformation plus an offset) of the computation and data onto the processors. The affine mappings are chosen so as to maximize the re-use of data by the same processor while maintaining sufficient parallelism to keep the processors busy. The compiler also uses *loop blocking* to reorder the computation executed on a single processor so that data is re-used in the cache.

Making Processor Data Contiguous. The compiler tries to arrange the data so that the accesses of a processor are contiguous whenever possible. This improves spatial locality, reduces conflict misses and reduces false sharing. The SUIF compiler is able to manage data placement within a single array, as well as across multiple arrays. The data-to-processor mappings computed by the affine partitioning analysis are used to determine the data being accessed by each processor. To make the data within a single array that is accessed by one processor contiguous, the compiler uses *data permutation* and *data stripmining*[1]. Data permutation interchanges the dimensions of the array—e.g. transposing a two-dimensional array. Data stripmining changes the dimensionality of an array so that all data accessed by the same processor is in the same plane of the array. To make the data across multiple arrays that is accessed by the same processor contiguous, we use a technique that involves the cooperation of the compiler and operating system called *compiler-directed page coloring*[5]. The compiler uses its knowledge of the access patterns to direct the operating system's page allocation policy to make each processor's data contiguous in the physical address space. The operating system uses these hints to determine the virtual-to-physical page mapping at page allocation time.

4 Experimental Results

We demonstrate the impact of SUIF's analyses and optimizations with a series of performance evaluations. The measurements were obtained on a Digital AlphaServer 8400 with eight 21164 processors, each with two levels of on-chip cache and a 4MB external cache. As it is harder to obtain speedups on machines with fast processors, the use of a state-of-the-art machine in our experiment makes our results more meaningful and applicable to future systems.



The diagram shows how data transformations are used to make the data accessed by each processor contiguous in the shared address space. The original arrays are two-dimensional; the axes are numbered so that elements along the first axis are contiguous. First the affine partitioning analysis determines which data elements are accessed by the same processor (in the figure, the shaded elements are those accessed by the first processor). Second, data stripmining turns the two-dimensional array into a three-dimensional array. Now all the shaded elements accessed by the same processor are in the same plane of the array. Finally, data permutation is applied to rotate the array so that data accessed by each processor become contiguous, as shown in the diagram.

Figure 2: Illustration of Memory Optimizations

We use two complete standard benchmark suites to evaluate our compiler. We present results on the 10 programs in the SPECfp95 benchmark suite, which is commonly used for benchmarking uniprocessors. The runs were performed using the full reference data sets. We also use the eight programs from the NAS parallel benchmarks designed to benchmark parallel systems. We used the official benchmark programs, with the exception of embar where we used a slightly modified version from Applied Parallel Research. In all cases we used the large sample data set size provided.

Figure 3 shows the speedups of the SPECfp95 and the NAS benchmarks, measured on up to eight processors on a 300MHz AlphaServer. The speedups were calculated over the best sequential execution time. Note that mgrid and applu appear in both benchmark suites (the program source and data set sizes differ slightly in the two suites). To measure the effects of the different compiler techniques, we break down the performance obtained on eight processors into three components, as shown in Figure 4. *Baseline* shows the speedup obtained with parallelization based on intraprocedural data dependence analysis and only scalar privatization and reduction transformations. *Coarse grain* includes the baseline techniques as well as techniques for locating coarse-grain parallel loops: array privatization and reduction transformations, and

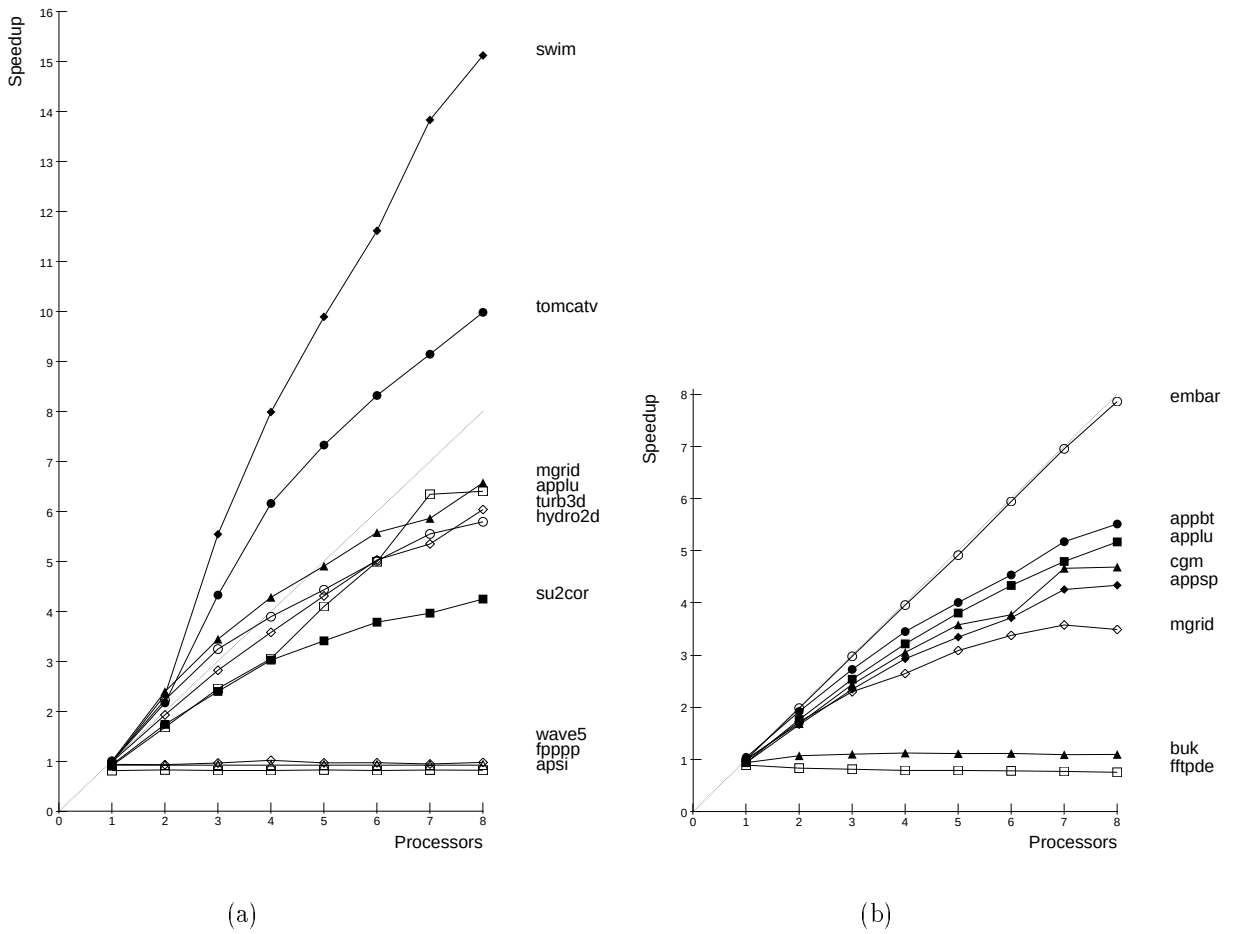


Figure 3: Speedups on the (a) SPECfp95 and (b) NAS Parallel Benchmarks

full interprocedural analysis of both scalar and array variables. *Memory* includes the coarse-grain techniques as well as all the multiprocessor memory optimizations described in Section 3.

The figure shows that of the 18 programs, three-fourths show good parallel speedup and are thus able to take advantage of additional processors. SUIF's coarse-grain techniques and memory optimizations have a significant impact on the performance of half of all the programs. The swim and tomcatv programs show superlinear speedups because the compiler eliminates almost all cache misses, and their 14MB working sets fit into the multiprocessor's aggregate cache. For most of the programs that do not speed up, the compiler finds a significant portion of their computation to be parallelizable, but the granularity is too fine to yield good multiprocessor performance on machines with fast processors. Only two applications, fpppp and buk, have no statically analyzable loop-level parallelism and are thus not amenable to our techniques.

To show that our compiler delivers high absolute performance, we report the times and SPEC ratios obtained on an eight-processor 440MHz Digital AlphaServer 8400 in Table 1. The SPEC ratios compare the machine performance to that of a reference machine. The total SPEC ratio is the geometric mean of the ratios obtained for individual programs. The geometric mean of the performance improves over the uniprocessor execution by 3.0 with four processors and by 4.3 with eight processors. The eight-processor ratio of 63.9 we obtain represents a 50% improvement over the highest number reported to date[7].

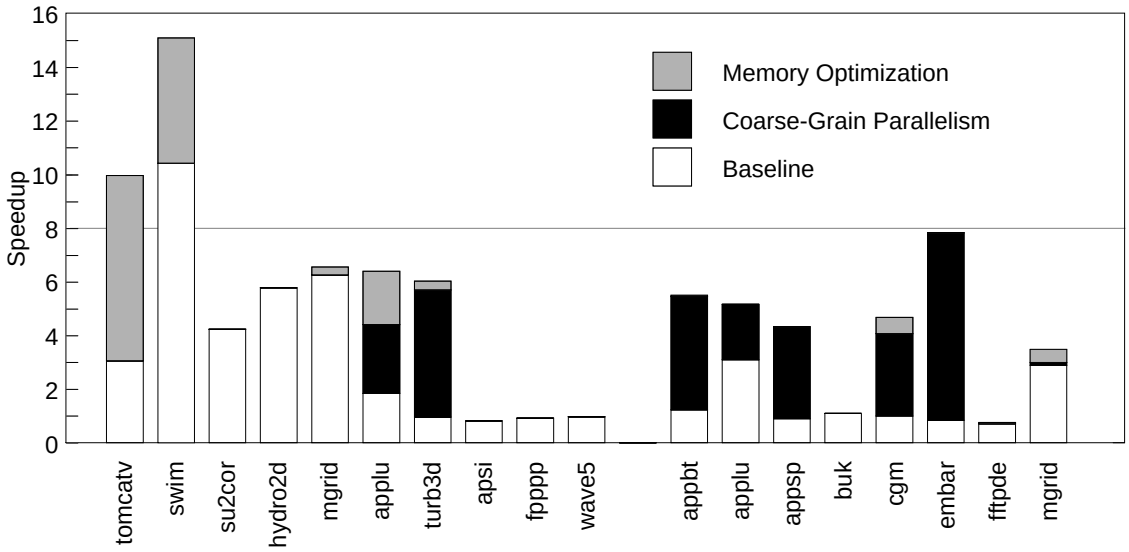


Figure 4: Impact of SUIF optimizations, shown as speedup on eight processors.

	Execution Time (sec)			SPEC ratio		
Benchmark	1P	4P	8P	1P	4P	8P
tomcatv	219.1	30.3	18.5	16.9	122.1	200.0
swim	297.9	33.5	17.2	28.9	256.7	500.0
su2cor	155.0	44.9	31.0	9.0	31.2	45.2
hydro2d	249.4	61.1	40.7	9.6	39.3	59.0
mgrid	185.3	42.0	27.0	13.5	59.5	92.6
applu	296.1	85.5	39.5	7.4	25.7	55.7
turb3d	267.7	73.6	43.5	15.3	55.7	94.3
apsi	137.5	141.2	143.2	15.3	14.9	14.7
fpppp	331.6	331.6	331.6	29.0	29.0	29.0
wave5	151.8	141.9	147.4	19.8	21.1	20.4
SPEC ratio				**15.0**	**44.4**	**63.9**

Table 1: Absolute performance for the SPECfp95 benchmarks measured on a 440MHz Digital AlphaServer.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[3] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10), October 1995.

[4] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Effective automatic parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

[5] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1996.

[6] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[7] Standard Performance Evaluation Corporation. Digital Equipment Corporation Alphaserver 8400 5/440 SPEC CFP95 results. *spec newsletter*, 8(3), October 1996.

[8] M. Haghighat and C. Polychronopolous. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4), July 1996.

[9] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[10] P. Havlak. *Interprocedural symbolic analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994.

[11] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[12] K. Kennedy and U. Kremer. Automatic data layout for high performance Fortran. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.