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

Parallelizing compilers for multiprocessors face many hurdles. However, SUIF's robust analysis and memory optimization techniques enabled speedups on three fourths of the NAS and SPECfp95 benchmark programs.

The affordability of shared memory multiprocessors offers the potential of supercomputer-class performance to the general public. Typically used in a multiprogramming mode, these machines increase throughput by running several independent applications in parallel. But multiple processors can also work together to speed up single applications. This requires that ordinary sequential programs be rewritten to take advantage of the extra processors.[1–4] Automatic parallelization with a compiler offers a way to do this.

Parallelizing compilers face more difficult challenges from multiprocessors than from vector machines, which were their initial target. Using a vector architecture effectively involves parallelizing repeated arithmetic operations on large data streams—for example, the innermost loops in array-oriented programs. On a multiprocessor, however, this approach typically does not provide sufficient granularity of parallelism: Not enough work is performed in parallel to overcome processor synchronization and communication overhead. To use a multiprocessor effectively, the compiler must exploit coarse-grain parallelism, locating large computations that can execute independently in parallel.

Locating parallelism is just the first step in producing efficient multiprocessor code. Achieving high performance also requires effective use of the memory hierarchy, and multiprocessor systems have more complex memory hierarchies than typical vector machines: They contain not only shared memory but also multiple levels of cache memory.

These added challenges often limited the effectiveness of early parallelizing compilers for multiprocessors, so programmers developed their applications from scratch, without assistance from tools. But explicitly managing an application's parallelism and memory use requires a great deal of programming knowledge, and the work is tedious and error-prone. Moreover, the resulting programs are optimized for only a specific machine. Thus, the effort required to develop efficient parallel programs restricts the user base for multiprocessors.

This article describes automatic parallelization techniques in the SUIF (Stanford University Intermediate

Format) compiler that result in good multiprocessor performance for array-based numerical programs. We provide SUIF performance measurements for the complete NAS and SPECfp95 benchmark suites. Overall, the results for these scientific programs are promising. The compiler yields speedups on three fourths of the programs and has obtained the highest ever performance on the SPECfp95 benchmark, indicating that the compiler can also achieve efficient absolute performance.

Finding Coarse-grain Parallelism

Multiprocessors work best when the individual processors have large units of independent computation, but it is not easy to find such coarse-grain parallelism. First the compiler must find available parallelism across procedure boundaries. Furthermore, the original computations may not be parallelizable as given and may first require some transformations. For example, experience in parallelizing by hand suggests that we must often replace global arrays with private versions on different processors. In other cases, the computation may need to be restructured—for example, we may have to replace a sequential accumulation with a parallel reduction operation.

It takes a large suite of robust analysis techniques to successfully locate coarse-grain parallelism. General and uniform frameworks helped us manage the complexity involved in building such a system into SUIF. We automated the analysis to privatize arrays and to recognize reductions to both scalar and array variables. Our compiler's analysis techniques all operate seamlessly across procedure boundaries.

Scalar Analyses

An initial phase analyzes scalar variables in the programs. It uses techniques such as data dependence analysis, scalar privatization analysis, and reduction recognition to detect parallelism among operations with scalar variables. It also derives symbolic information on these scalar variables that is useful in the array analysis phase. Such information includes constant propagation, induction variable recognition and elimination, recognition of loop-invariant computations, and symbolic relation propagation.[5,6]

Array Analyses

An array analysis phase uses a unified mathematical framework based on linear algebra and integer linear programming.[3] 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 dataflow information that determines whether array elements used in an iteration refer to the values produced in a previous iteration. Moreover, it 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 that represent the data accessed. These inequalities are derived from loop bounds and array access functions. Implementing optimizations to speed up common cases reduces the compilation time.

Interprocedural Analysis Framework

All the analyses are implemented using a uniform interprocedural analysis framework, which helps manage the software engineering complexity. The framework uses interprocedural dataflow analysis,[4] which is more efficient than the more common technique of inline substitution.[1] 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, because it can make the program too large to analyze.

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 different calling contexts make it necessary, the algorithm selectively clones a procedure so that code can be analyzed and possibly parallelized under different calling contexts (as when different constant values are passed to the same formal parameter). In this way the full advantages of inlining are achieved without expanding the code indiscriminately.

In Figure 1 the boxes 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. Using interprocedural scalar and array analyses, the SUIF compiler determines that these loops are parallelizable. Each loop contains more than 500 lines of code spanning up to nine procedures with up to 42 procedure calls. If this program had been fully inlined, the loops presented to the compiler for analysis would have each contained more than 86,000 lines of code.

Memory Optimization

Numerical applications on high-performance microprocessors are often memory bound. Even with one or more levels of cache to bridge the gap between processor and memory speeds, a processor may still waste half its time stalled on memory accesses because it frequently references an item not in the cache (a cache miss). This

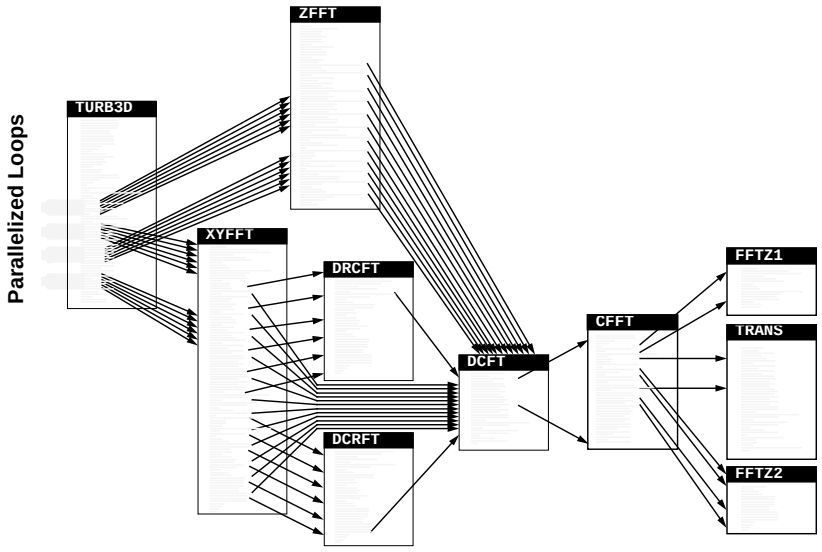

Figure 1
The compiler discovers parallelism through interprocedural array analysis. Each of the four parallelized loops at left consists of more than 500 lines of code spanning up to nine procedures (boxes) with up to 42 procedure calls (lines).

memory bottleneck is further exacerbated on multiprocessors by their greater need for memory traffic, resulting in more contention on the memory bus.

An effective compiler must address four issues that affect cache behavior:

- Communication: Processors in a multiprocessor system communicate 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 cache capacity. These applications often stream through large amounts of data before reusing any of it, resulting in poor temporal locality and numerous capacity misses.
- Limited associativity: Caches typically have a small *set associativity;* that is, each memory location can map to only one or just a few locations in the cache. Conflict misses—when an item is discarded and later retrieved—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 incur more misses and are said to have poor spatial locality. On a multiprocessor, large cache lines can also lead to cache misses when different processors use different parts of the same cache line. Such misses are called *false sharing misses.*

The compiler tries to eliminate as many cache misses as possible, then minimize the impact of any that remain by

- ensuring that processors reuse the same data as many times as possible and
- making the data accessed by each processor contiguous in the shared address space.

Techniques for addressing each of these subproblems 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 Data Reuse

The compiler reorganizes the computation so that each processor reuses data to the greatest possible extent.[7-9] This reduces the working set on each processor, thereby minimizing capacity misses. It also reduces interprocessor communication and thus minimizes true sharing misses. To achieve optimal reuse, the compiler uses *affine partitioning.* This technique analyzes reference patterns in the program to derive an affine mapping (linear transformation plus an offset) of the computation of the data to the processors. The affine mappings are chosen to maximize a processor's reuse of data while maintaining sufficient parallelism to keep all processors busy. The compiler also uses loop blocking to reorder the computation executed on a single processor so that data is reused in the cache.

Making Processor Data Contiguous

The compiler tries to arrange the data to make a processor's accesses contiguous in the shared address space. This improves spatial locality while reducing conflict misses and false sharing. SUIF can manage data placement within a single array and 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.

Figure 2 shows how the compiler's use of data permutation and data strip-mining[10] can make contiguous the data within a single array that is accessed by one processor. Data permutation interchanges the dimensions of the array—for example, transposing a two-dimensional array. Data strip-mining changes an array's dimensionality so that all data accessed by the same processor are in the same plane of the array.

To make data across multiple arrays accessed by the same processor contiguous, we use a technique called *compiler-directed page coloring*.[11] 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.

Experimental Results

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

We used two complete standard benchmark suites to evaluate our compiler. We present results for the 10

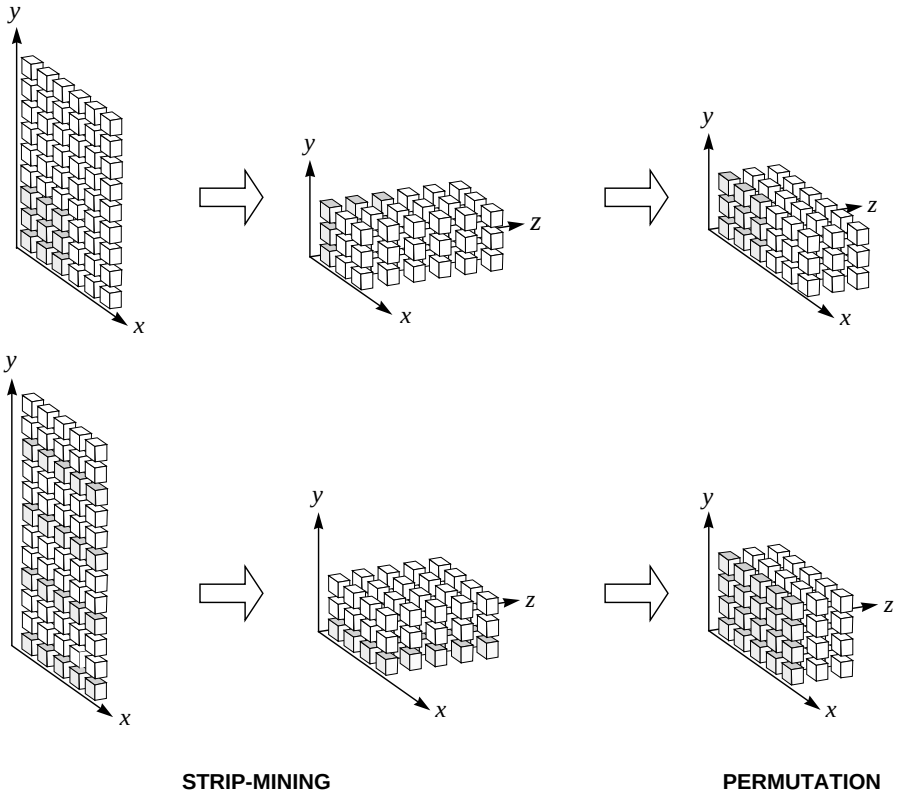

Figure 2
Data transformations can make the data accessed by each processor contiguous in the shared address space. In the two examples above, the original arrays are two-dimensional; the axes are identified to show that elements along the first axis are contiguous. First the affine partitioning analysis determines which data elements are accessed by the same processor (the shaded elements are accessed by the first processor.) Second, data strip-mining turns the 2D array into a 3D array, with the shaded elements in the same plane. Finally, applying data permutation rotates the array, making data accessed by each processor contiguous.

programs in the SPECfp95 benchmark suite, which is commonly used for benchmarking uniprocessors. We also used the eight official benchmark programs from the NAS parallel-system benchmark suite, except for embar; here we used a slightly modified version from Applied Parallel Research.

Figure 3 shows the SPECfp95 and NAS speedups, measured on up to eight processors on a 300-MHz AlphaServer. We calculated the speedups over the best sequential execution time from either officially reported results or our own measurements. Note that mgrid and applu appear in both benchmark suites (the program source and data set sizes differ slightly).

To measure the effects of the different compiler techniques, we broke down the performance obtained on eight processors into three components. In Figure 4, baseline shows the speedup obtained with parallelization using only intraprocedural data dependence analysis, scalar privatization, and scalar reduction transformations. Coarse grain includes the baseline techniques as well as techniques for locating coarse-grain parallel loops—for example, array privatization and reduction transformations, and full interprocedural analysis of both scalar and array variables. Memory includes the coarse-grain techniques as well as the multiprocessor memory optimizations we described earlier.

Figure 3 shows that of the 18 programs, 13 show good parallel speedup and can thus take advantage of additional processors. SUIF's coarse-grain techniques and memory optimizations significantly affect the performance of half the programs. The swim and tomcatv programs show superlinear speedups because the compiler eliminates almost all cache misses and their 14 Mbyte working sets fit into the multiprocessor's aggregate cache.

For most of the programs that did not speed up, the compiler found much 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

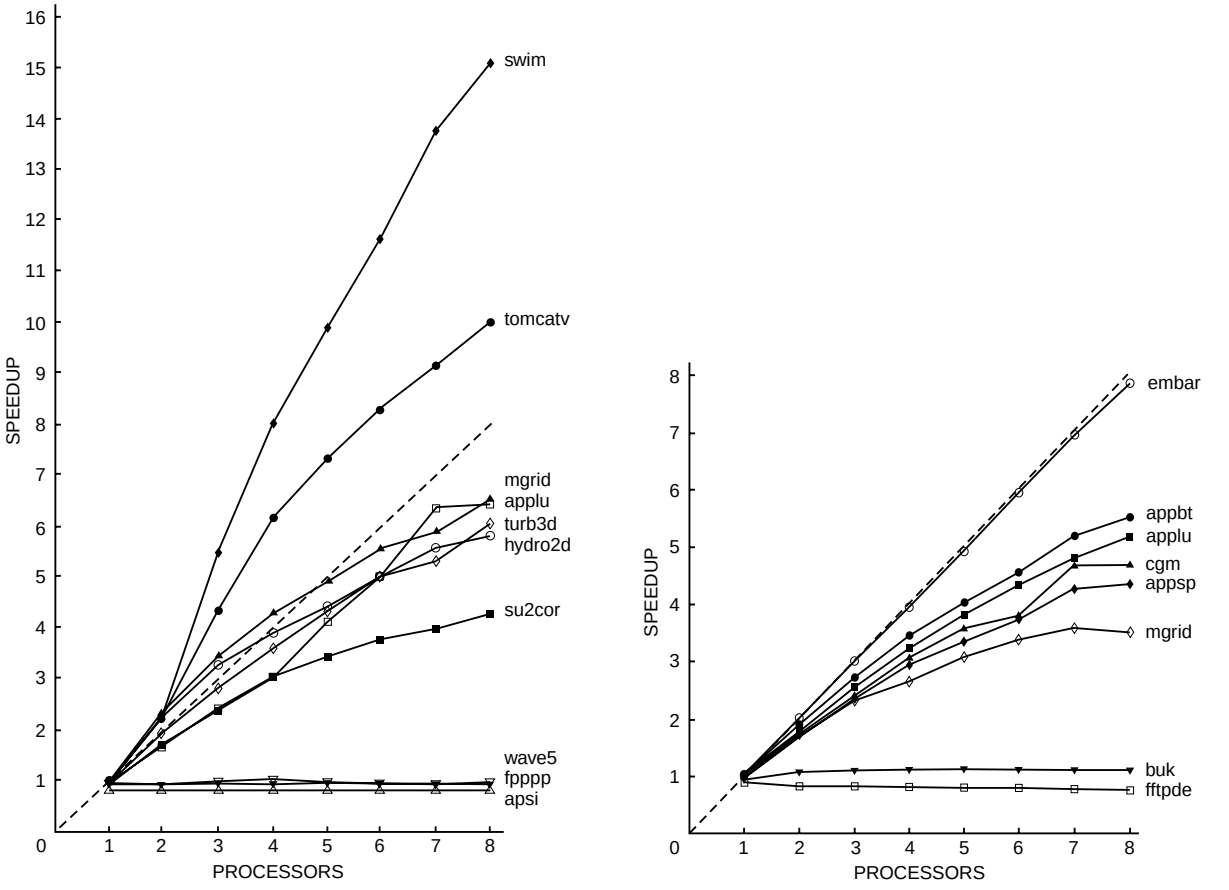

Figure 3
SUIF compiler speedups over the best sequential time achieved on the (a) SPECfp95 and (b) NAS parallel benchmarks.

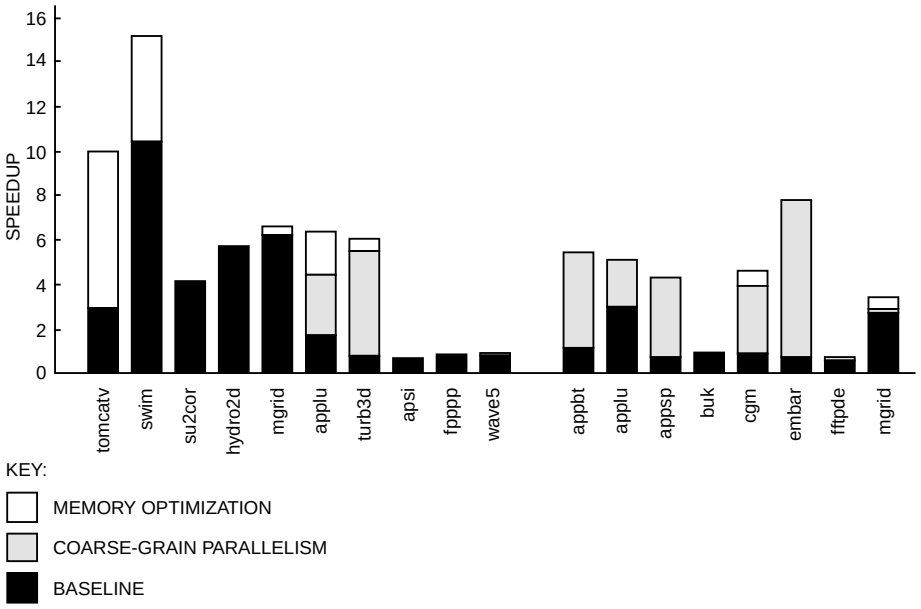

Figure 4
The speedup achieved on eight processors is broken down into three components to show how SUIF's memory optimization and discovery of coarse-grain parallelism affected performance.

no statically analyzable loop-level parallelism, so they are not amenable to our techniques.

Table 1 shows the times and SPEC ratios obtained on an eight-processor, 440-MHz Digital AlphaServer 8400, testifying to our compiler's high absolute performance. The SPEC ratios compare machine performance with that of a reference machine. (These are not official SPEC ratings, which among other things require that the software be generally available. The ratios we obtained are nevertheless valid in assessing our compiler's performance.) The geometric mean of the SPEC ratios improves over the uniprocessor execution by a factor of 3 with four processors and by a factor of 4.3 with eight processors. Our eight-processor ratio of 63.9 represents a 50 percent improvement over the highest number reported to date.[12]

Table 1
Absolute Performance for the SPECfp95 Benchmarks Measured on a 440-MHz Digital AlphaServer Using One Processor, Four Processors, and Eight Processors

	Execution Time (secs)			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
Geometric Mean				15.0	44.4	63.9

Acknowledgments

This research was supported in part by the Air Force Materiel Command and ARPA contracts F30602-95-C-0098, DABT63-95-C-0118, and DABT63-94-C-0054; a Digital Equipment Corporation grant; an NSF Young Investigator Award; an NSF CISE postdoctoral fellowship; and fellowships from AT&T Bell Laboratories, DEC Western Research Laboratory, Intel Corp., and the National Science Foundation.

References

1. J.M. Anderson, S.P. Amarasinghe, and M.S. Lam, "Data and Computation Transformations for Multiprocessors," *Proc. Fifth ACM SIGPlan Symp. Principles and Practice of Parallel Programming*, ACM Press, New York, 1995, pp. 166–178.
2. J. M. Anderson and M.S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proc. SIGPlan '93 Conf. Programming Language Design and Implementation,* ACM Press, New York, 1993, pp. 112–125.
3. P. Banerjee et al., "The Paradigm Compiler for Distributed-Memory Multicomputers," *Computer,* Oct. 1995, pp. 37–47.
4. W. Blume et al., "Effective Automatic Parallelization with Polaris," *Int'l. J. Parallel Programming,* May 1995.
5. E. Bugnion et al., "Compiler-Directed Page Coloring for Multiprocessors," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, 1996, pp. 244–257.
6. K. Cooper et al., "The ParaScope Parallel Programming Environment," *Proc. IEEE,* Feb. 1993, pp. 244–263.
7. Standard Performance Evaluation Corp., "Digital Equipment Corporation AlphaServer 8400 5/440 SPEC CFP95 Results," *SPEC Newsletter,* Oct. 1996.
8. M. Haghighat and C. Polychronopolous, "Symbolic Analysis for Parallelizing Compilers," *ACM Trans. Programming Languages and Systems,* July 1996, pp. 477–518.
9. M.W. Hall et al., "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," *Proc. Supercomputing '95,* IEEE CS Press, Los Alamitos, Calif., 1995 (CD-ROM only).
10. P. Havlak, *Interprocedural Symbolic Analysis,* PhD thesis, Dept. of Computer Science, Rice Univ., May 1994.
11. F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical Interprocedural Parallelization: An Overview of the PIPS Project," *Proc. 1991 ACM Int'l Conf. Supercomputing,* ACM Press, New York, 1991, pp. 244–251.
12. K. Kennedy and U. Kremer, "Automatic Data Layout for High Performance Fortran," *Proc. Supercomputing '95,* IEEE CS Press, Los Alamitos, Calif., 1995 (CD-ROM only).

Editors' Note: With the following section, the authors provide an update on the status of the SUIF compiler since the publication of their paper in Computer *in December 1996.*

Addendum: The Status and Future of SUIF

Public Availability of SUIF-parallelized Benchmarks

The SUIF-parallelized versions of the SPECfp95 benchmarks used for the experiments described in this paper have been released to the SPEC committee and are available to any license holders of SPEC (see http://www.specbench.org/osg/cpu95/par-research). This benchmark distribution contains the SUIF output (C and FORTRAN code), along with the source code for the accompanying run-time libraries. We expect these benchmarks will be useful for two purposes: (1) for technology transfer, providing insight into how the compiler transforms the applications to yield the reported results; and (2) for further experimentation, such as in architecture-simulation studies.

The SUIF compiler system itself is available from the SUIF web site at http://www-suif.stanford.edu. This system includes only the standard parallelization analyses that were used to obtain our baseline results.

New Parallelization Analyses in SUIF

Overall, the results of automatic parallelization reported in this paper are impressive; however, a few applications either do not speed up at all or achieve limited speedup at best. The question arises as to whether SUIF is exploiting all the available parallelism in these applications. Recently, an experiment to answer this question was performed in which loops left unparallelized by SUIF were instrumented with run-time tests to determine whether opportunities for increasing the effectiveness of automatic parallelization remained in these programs.[1] Run-time testing determined that eight of the programs from the NAS and SPEC95fp benchmarks had additional parallel loops, for a total of 69 additional parallelizable loops, which is less than 5% of the total number of loops in these programs. Of these 69 loops, the remaining parallelism had a significant effect on coverage (the percentage of the program that is parallelizable) or granularity (the size of the parallel regions) in only four of the programs: apsi, su2cor, wave5, and fftpde.

We found that almost all the significant loops in these four programs could potentially be parallelized using a new approach that associates predicates with array data-flow values.[2] Instead of producing conserv-

ative results that hold for all control-flow paths and all possible program inputs, predicated array data-flow analysis can derive optimistic results guarded by predicates. Predicated array data-flow analysis can lead to more effective automatic parallelization in three ways: (1) It improves compile-time analysis by ruling out infeasible control-flow paths. (2) It provides a framework for the compiler to introduce predicates that, if proven true, would guarantee safety for desirable data-flow values. (3) It enables the compiler to derive low-cost run-time parallelization tests based on the predicates associated with desirable data-flow values.

SUIF and Compaq's GEM Compiler

The GEM compiler system is the technology Compaq has been using to build compiler products for a variety of languages and hardware/software platforms.[3] Within Compaq, work has been done to connect SUIF with the GEM compiler. SUIF's intermediate representation was converted into GEM's intermediate representation, so that SUIF code can be passed directly to GEM's optimizing back end. This eliminates the loss of information suffered when SUIF code is translated to C/FORTRAN source before it is passed to GEM. It also enables us to generate more efficient code for Alpha-microprocessor systems.

SUIF and the National Compiler Infrastructure

The SUIF compiler system was recently chosen to be part of the National Compiler Infrastructure (NCI) project funded by the Defense Advanced Research Projects Agency (DARPA) and the National Science Foundation (NSF). The goal of the project is to develop a common compiler platform for researchers and to facilitate technology transfer to industry. The SUIF component of the NCI project is the result of the collaboration among researchers in five universities (Harvard University, Massachusetts Institute of Technology, Rice University, Stanford University, University of California at Santa Barbara) and one industrial partner, Portland Group Inc. Compaq is a corporate sponsor of the project and is providing the FORTRAN front end.

A revised version of the SUIF infrastructure (SUIF 2.0) is being released as part of the SUIF NCI project (a preliminary version of SUIF 2.0 is available at the SUIF web site). The completed system will be enhanced to support parallelization, interprocedural analysis, memory hierarchy optimizations, objected-oriented programming, scalar optimizations, and machine-dependent optimizations. An overview of the SUIF NCI system is shown in Figure A1. See www-suif.stanford.edu/suif/NCI/suif.html for more information about SUIF and the NCI project, including a complete list of optimizations and a schedule.

References

1. B. So, S. Moon, and M. Hall, "Measuring the Effectiveness of Automatic Parallelization in SUIF," *Proceedings of the International Conference on Supercomputing '98,* July 1998.
2. S. Moon, M. Hall, and B. Murphy, "Predicated Array Data-Flow Analysis for Run-Time Parallelization," *Proceedings of the International Conference on Supercomputing '98,* July 1998.
3. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue, 1992): 121–136.

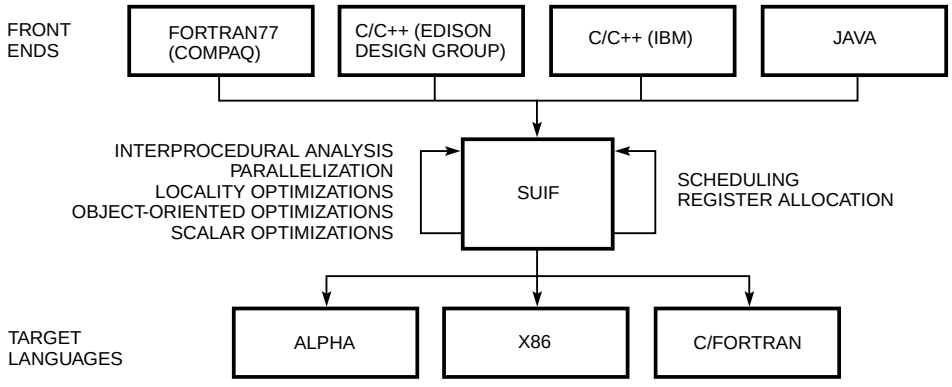

Figure A1
The SUIF Compiler Infrastructure

Biographies

Mary W. Hall
Mary Hall is jointly a research assistant professor and project leader at the University of Southern California, Department of Computer Science and at USC's Information Sciences Institute, where she has been since 1996. Her research interests focus on compiler support for high-performance computing, particularly interprocedural analysis and automatic parallelization. She graduated magna cum laude with a B.A. in computer science and mathematical sciences in 1985 and received an M.S. and a Ph.D. in computer science in 1989 and 1991, respectively, all from Rice University. Prior to joining USC/ISI, she was a visiting assistant professor and senior research fellow in the Department of Computer Science at Caltech. In earlier positions, she was a research scientist at Stanford University, working with the SUIF Compiler group, and in the Center for Research on Parallel Computation at Rice University.

Jennifer M. Anderson
Jennifer Anderson is a research staff member at Compaq's Western Research Laboratory where she has worked on the Digital Continuous Profiling Infrastructure (DCPI) project. Her research interests include compiler algorithms, programming languages and environments, profiling systems, and parallel and distributed systems software. She earned a B.S. in information and computer science from the University of California at Irvine and received M.S. and Ph. D. degrees in computer science from Stanford University.

Saman P. Amarasinghe
Saman Amarasinghe is an assistant professor of computer science and engineering at the Massachusetts Institute of Technology and a member of the Laboratory for Computer Science. His research interests include compilers and computer architecture. He received a B.S. in electrical engineering and computer science from Cornell University and M.S. and Ph.D. degrees in electrical engineering from Stanford University.

Brian R. Murphy
A doctoral candidate in computer science at Stanford University, Brian Murphy is currently working on advanced program analysis under SUIF as part of the National Compiler Infrastructure Project. He received a B.S. in computer science and engineering and an M.S. in electrical engineering and computer science from the Massachusetts Institute of Technology. His master's thesis work on program analysis was carried out with the Functional Languages group at the IBM Almaden Research Center. Brian was elected to the Tau Beta Pi and Eta Kappa Nu honor societies.

Shih-Wei Liao
Shih-Wei Liao is a doctoral candidate at the Stanford University Computer Systems Laboratory. His research interests include compiler algorithms and design, programming environments, and computer architectures. He received a B.S. in computer science from National Taiwan University in 1991 and an M.S. in electrical engineering from Stanford University in 1994.

Edouard Bugnion
Ed Bugnion holds a Diplom in engineering from the Swiss Federal Institute of Technology (ETH), Zurich (1994) and an M.S. from Stanford University (1996), where he is a doctoral candidate in computer science. His research interests include operating systems, computer architecture, and machine simulation. From 1996 to 1997, Ed was also a research consultant to Compaq's Western Research Laboratory. He is the recipient of a National Science Foundation Graduate Research Fellowship.

Monica S. Lam
Monica Lam is an associate professor in the Computer Science Department at Stanford University. She leads the SUIF project, which is aimed at developing a common infrastructure to support research in compilers for advanced languages and architectures. Her research interests are compilers and computer architecture. Monica earned a B.S. from the University of British Columbia in 1980 and a Ph.D. in computer science from Carnegie Mellon University in 1987. She received the National Science Foundation Young Investigator award in 1992.