Multiprocessors from a Software Perspective

Saman P. Amarasinghe
Jennifer M. Anderson
Christopher S. Wilson
Shih-Wei Liao
Brian R. Murphy
Robert S. French
Monica S. Lam

Stanford University

Mary W. Hall

California Institute of Technology

We used the SUIF research compiler to parallelize the SPECfp benchmark programs and obtained the highest recorded SPECfp ratios by running them on an eight-processor machine.

Like many architectural techniques that originated with mainframes, the use of multiple processors in a single computer is becoming popular in workstations and even personal computers. Multiprocessors constitute a significant percentage of recent workstation sales, and highly affordable multiprocessor personal computers are available in local computer stores. Once again, we find ourselves in a familiar situation: hardware is ahead of software.

Because of the complexity of parallel programming, multiprocessors today are rarely used to speed up individual applications. Instead, they usually function as cycle-servers that achieve increased system throughput by running multiple tasks simultaneously. Automatic parallelization by a compiler is a particularly attractive approach to software development for multiprocessors, as it enables ordinary sequential programs to take advantage of the multiprocessor hardware without user involvement. This article looks to the future by examining some of the latest research results in automatic parallelization technology.

Why multiprocessors?

The success of the RISC revolution demonstrates that designers must take state-of-the-art compiler technology into account when creating next-generation machines. The effectiveness of parallelizing compilers, besides being important for existing systems, has significant implications for future machine design. Once multiple processors are transparent to the programmer, multiprocessors will be as easy to use as traditional uniprocessors. We can then evaluate the multiprocessor solely on price/performance and treat additional processors like other performance enhancement features such as the cache organization or the instruction pipeline.

This perspective is particularly significant in the design of future microarchitectures. In the quest for better microprocessor performance, the current trend is to build wider superscalar or VLIW (very long instruction word) machines. As designers add more functional units to a processor, however, fewer programs can use all the units effectively, and the marginal return of the additional hardware decreases. In fact, higher processor complexity can increase a machine's cycle time, which may even slow down the performance of programs that do not take advantage of the increased parallelism.

Only regular numeric programs have successfully exploited instruction level parallelism (ILP) at a higher level than that offered in today's microprocessors. For many of these same programs, automatic parallelization can also produce efficient results on a multiprocessor. The compiler can sometimes even find multiprocessor-usable parallelism in programs that exhibit little ILP. More importantly, a multiprocessor's additional hardware cost is not wasted on inherently sequential applications, which can still run in a multiprogrammed mode to deliver better overall system throughput.

The multiprocessor also offers several hardware implementation advantages over a single large and complex processor. The multiprocessor's modularity makes possible a system design based on a replicated processor core, possibly an already existing one. The reduced complexity of the multiprocessor design makes possible a faster clock speed and a shorter time to market.

Experimental data we obtained using the SUIF (Stanford University Intermediate Format) compiler[1] support the view that multiprocessors can effectively parallelize individual applications. SUIF is a research parallelizing compiler developed at Stanford University over the last seven years. The

0272-1732/96/$5.00 © 1996 IEEE

compiler is effective at parallelizing numeric applications that operate on array data structures. We have recently developed a pointer analysis algorithm that extends parallelization to C and not just Fortran programs. We have also extended traditional parallelism analysis to operate on arrays and across procedures to detect outer-loop parallelism, and have developed a set of locality optimization techniques to make multiprocessor caches more effective. Together, these techniques greatly expand a compiler's ability to use a multiprocessor effectively.

In our experiments, we used SUIF to parallelize the SPEC92 and SPEC95 floating-point applications for Digital Equipment Corporation's AlphaServer 8400 multiprocessor. (The SPEC programs, which let us benchmark computer systems, are not tailored to run on a multiprocessor.) The 8400 uses the 300-MHz Digital 21164 Alpha processor, a leader in SPEC performance among microprocessors currently available. As it is hard to achieve speedups on machines with fast processors, our choice of the 8400 presents a greater challenge to parallelization while also making our conclusions more likely to apply to systems with future processors.

SUIF successfully boosted the highest reported SPEC92fp ratio of 506 to 1,016 by parallelizing the code for eight processors. We raised the SPEC95fp ratio from 12.2 on one processor to 38.4 on eight processors. Whereas SPEC results alone are not a perfect indicator of the compiler's effectiveness, our parallelization techniques are general enough to be applicable to a large class of programs. Altogether, these results show that parallelization technology has greatly matured in recent years, and that we can effectively parallelize many numeric applications.

Our results suggest that the multiprocessor, with the help of a parallelizing compiler, offers serious competition to very wide superscalar or VLIW machines. Since we obtained positive results on multiprocessors that already exist, our results provide stronger support for such a conclusion than do simulation results for hypothetical machines. In the near future, it will become feasible to integrate multiple processors on a single chip; a multiprocessor on a chip should provide even better performance for parallelized code, as it should have better support for finer granularity of parallelism.

SUIF parallelizing compiler

The SUIF parallelizing compiler is a fully functional compiler that converts sequential Fortran and C programs into SPMD (single program, multiple data) source code for shared-memory multiprocessors. For the experiments described in this article, we generated a combination of C and Fortran source code; we used C code to coordinate parallel execution and Fortran code for most of the programs' computational sections.

To provide a complete context in which to evaluate new research techniques, the SUIF system includes the conventional parallelization techniques that form the basis for most commercial parallelizers today. It performs data dependence analysis—which determines whether iterations of a loop operate on different array elements—to decide if a loop is parallelizable. It uses techniques that recognize reduction operations (for example, summations and products), convert a scalar variable into private copies on each processor, and transform loop nests (such as interchanging inner and outer loops) for the sake of parallelism and locality. We refer to these analyses and optimizations as "first-generation" parallelization techniques. In 1994, we released a system consisting of these basic parallelization techniques. It is publicly available via the World Wide Web at http://suif.stanford.edu.

Besides these conventional techniques, the SUIF system includes three unique experimental components, which we will also make publicly available once they are stable. First, to extend the scope of parallelization to include C programs, we have developed an interprocedural algorithm that disambiguates pointer variables. Second, since the multiprocessor's unique abilities include executing loops containing complex control flow in parallel, we have extended previous techniques to identify outer-loop parallelism. Finally, as memory behavior can significantly affect performance on a shared-memory architecture, SUIF contains a suite of locality optimization techniques to make multiprocessor caches more effective. The case study boxes pre-

Case study: applu

The primary routines in the applu program calculate the lower and upper triangle solutions to a series of partial differential equations. The following is an excerpt of the code for the main loops in these routines:

```
do k = 2, nz-1
  do j = 2, ny-1
  do i = 2, nx-1
     v(m,i,j,k) = v(m,i,j,k)-omega*(ldz(m,l,i,j,k)*v(l,i,j,k–1)
        +ldy(m,l,i,j,k)*v(l,i,j-1,k)+ldx(m,l,i,j,k)*v(l,i–1,j,k))
     do m = 1, 5
       do l = 1, 5
         tmat(m,l) = ...
     do m = 1, 5
       do l = 1, 5
         ... = tmat(m,l)
```

Good parallel performance on this application requires both the advanced array analysis and the locality optimizations described in this article. Observe that each of the iterations of the three outer loops defines and uses the same array, tmat. The compiler gives each processor a private copy of the array so the processors will not interfere with each other. As there is a data dependence between the accesses of array v in different iterations, SUIF achieves concurrency by pipelining the iterations. Finally, to minimize the frequency of synchronization, the compiler assigns a block of iterations to each pipeline stage. The blocking transformation not only improves cache locality, but can also minimize synchronization and communication in multiprocessors. These transformations yield a speedup of over 2.5 times on a four-processor machine. This example illustrates the importance of a well-integrated set of advanced techniques.

Case study: turb3d

The main computation in the turb3d program is a series of loops that compute three-dimensional fast Fourier transforms (FFTs). While these loops are parallelizable, they all have a complex control structure containing large amounts of code, as shown in Figure A. The boxes represent procedures, and the lines represent procedure invocations. Each parallel loop, as the diagram indicates, consists of over 500 lines of code spanning eight or nine procedures, with up to 42 procedure calls.

To get any significant speedup, we must parallelize these outer loops. The key to discovering the parallelism is interprocedural array analysis. The compiler determines that iterations of the outer loops operate on independent planes of the arrays across the procedure calls. This analysis is difficult because the program contains array reshapes such that the three-dimensional arrays are treated as long vectors in some of the procedures. Once parallelized, turb3d yields a speedup of over 3.5 on a four-processor machine.

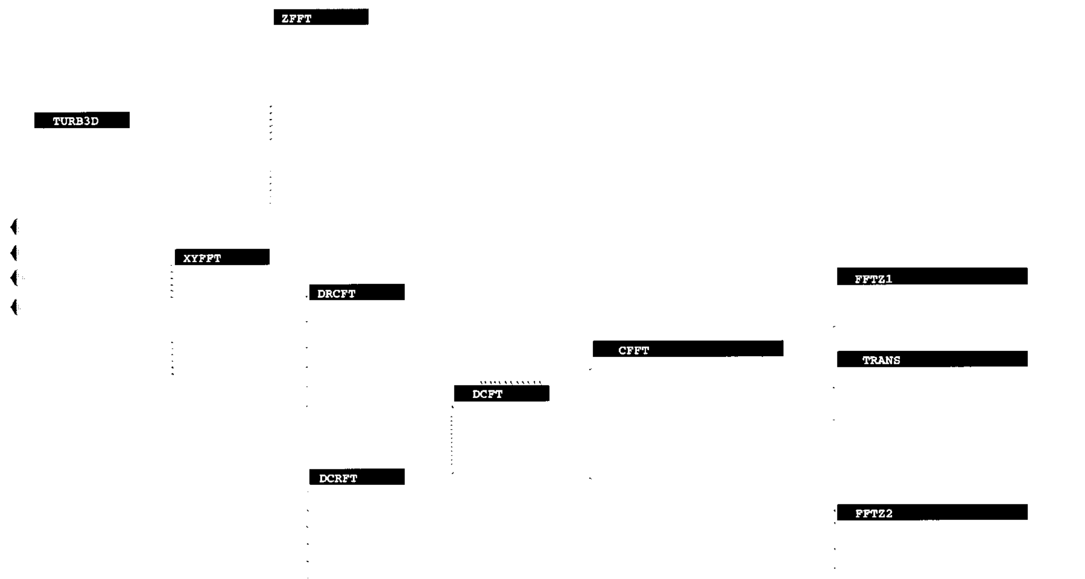


Figure A. Parallelized loops, turb3d, showing complexity only.

sent two examples of our work with the SPEC95fp benchmark suite.

Pointer analysis for C. For automatic parallelization to become generally useful, we must be able to handle popular programming languages. With the exception of Fortran, most programming languages in use have pointer variables that allow programmers to directly store and retrieve data addresses. Designed for low-level programming, the C language allows programmers even the freedom to perform arithmetic operations on the pointers. The presence of pointers makes parallelization difficult, as the compiler must prove that two memory operations are certain to access different locations before it can reorder them in a parallel execution. The analysis to determine if pointer variables might refer to the same location is known as pointer alias analysis.

Pointer alias analysis is one of the hardest problems in compilation. Obtaining precise results requires the compiler to apply the technique interprocedurally, across the entire program. Since the behavior of a procedure may depend on the aliases that hold in the context where it is invoked, precise results also demand a context-sensitive analysis, which considers each calling context separately. Many context-sensitive analyses summarize the effect of each function for efficiency, but pointer analysis is not amenable to such an optimization.

Building upon results generated by many researchers in the area, we have developed a new pointer analysis algorithm that finds context-sensitive pointer alias results efficiently. Our approach is to identify the calling-context aliases relevant to the procedure's behavior and then analyze the procedure once for all calling contexts that have the same relevant aliases. This leads to very fast analysis times and fully context-sensitive results. The analysis can differentiate between pointers to global and stack variables, but can only differentiate pointers to heap data structures allocated at different program points or in different calling contexts. For example, the algorithm can determine that two lists are disjoint if the list elements happen to be created by two different statements in the program, but it cannot determine that the data structures are linked lists. We currently have the technology to parallelize simple array-based computations written in C with about the same effectiveness as if the programs had been written in

Fortran. We expect to see further progress in pointer alias analysis and its applications in the near future.

Finding coarse-grained, loop-level parallelism. Efficient, hand-parallelized codes tend to be dominated by large, outer, parallel loops. Thus, for the compiler to achieve the same effect as hand parallelization, it must successfully parallelize large segments of code that may span many procedures.

Advanced array analyses. Parallelizing an outer loop is not just a matter of adding a "parallel" directive to a loop, because it is often necessary to change the data structures the computation uses. For example, it is very common for each iteration of a loop to assign and then use the same variable. To make the loop parallelizable, the compiler must give each processor a private copy of the variable. As another example, a loop might contain a reduction (for example, computation of a sum, product, or maximum over a set of data elements) that a compiler can parallelize by having each processor compute a partial reduction locally and update the global result at the end. To find parallel inner loops, we can simply privatize scalar variables and transform reductions that write to scalar variables. To find outermost parallel loops, however, we must extend these analyses to array variables.[3,4]

Array privatization analysis is much more difficult than the data dependence analysis found in first-generation parallelizers. For the latter, the compiler need only prove that different loop iterations are operating on different array elements. For the former, iterations may operate on the same locations, but the values one iteration generates must not be used by another.

The SUIF parallelizer also performs array reduction recognition. This technique locates reductions based on commutative operations—like summation, product, minimum, and maximum—that are updates of the same memory location. This approach is powerful enough to recognize commutative updates of indirectly accessed array locations, enabling the compiler to parallelize even reductions that operate on sparse arrays.

Interprocedural parallelization analysis. To find parallelism in outer loops, the compiler must perform analysis across procedure boundaries. A simple way to eliminate procedure boundaries is to perform in-line substitution—replacing each procedure call by a copy of the called procedure—and perform program analysis in the usual way.[5] However, this approach does not work for recursive programs and is not a practical solution for large programs, as program size can increase to an unmanageable extent. Interprocedural analysis, which applies data-flow analysis techniques across procedure boundaries, can be much more efficient by analyzing only a single copy of each procedure.

We have developed an interprocedural parallelizer that incorporates a comprehensive suite of analyses for parallelization.[3] Our compiler includes a set of interprocedural analyses on scalar variables, including dependence and privatization analysis and reduction recognition. It also includes several interprocedural analyses that assist the array analysis, including constant propagation, induction variable elimination, recognition of loop-invariant computations, and symbolic relation propagation. Finally, the compiler performs array data dependence analysis and all the advanced array analyses mentioned earlier in an interprocedural manner. This powerful set of optimizations can parallelize loops spanning hundreds of lines of codes and numerous nontrivial functions.

The interprocedural analysis algorithm we use is both precise and efficient. Our analysis is flow sensitive, which means that it precisely captures the effects of the control flow within each procedure. We use a region-based approach, where the regions of interest are loops and procedures. The algorithm separates analysis of procedure side effects from the propagation of calling contexts to the procedure, so that two passes over the program call graph suffice to complete a flow-sensitive analysis. In addition, we use selective procedure cloning to make the results context sensitive and therefore more precise. This technique replicates the analysis information for a procedure whenever two paths to the procedure contribute very different data-flow information.

Cache optimizations. As processors get faster, the memory hierarchy plays an increasingly important role in determining performance. Microprocessors rely on caches to shorten the effective memory access time, but caches often perform poorly on numeric applications. Since numeric applications have large data sets, a microprocessor often displaces data before it can reuse them. In cache-coherent multiprocessors, data sharing between processors introduces even more cache traffic, exacerbating the problem. Cache misses occur not only when processors share the same data words, but also when they use different words on the same cache line. Moreover, as each processor operates on only a subset of the data, it is more likely that the data it touches will have little spatial locality, further reducing cache benefits.

Cache performance has been a focus of the SUIF compiler from the very beginning of the project. We have developed techniques to partition computation across processors to minimize interprocessor communication.[6] Through unimodular loop transformations and blocking, the compiler reorders the computation each processor executes to enhance data locality.[7] We have also developed an interprocedural algorithm that changes the array data layout to minimize unnecessary cache traffic.[8] The algorithm reorganizes arrays to allocate data accessed by one processor in contiguous locations. For example, the compiler might change the organization of an array from column major to row major, or restructure a 2D array as a 3D array. We developed all the loop and data transformation algorithms within a unified theoretical framework based on linear algebra. Experimental results show that this technique can significantly improve program performance.

Parallelizing SPEC92fp programs

The SPEC92fp benchmark suite consists of 14 floating-point applications, briefly described in Table 1, next page. The standard measure of machine performance is the SPEC ratio, which compares 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 amount of parallelism the compiler can recognize

Table 1. Characteristics of SPEC92fp programs.

Program	Language	No. of lines	Description
alvinn	C	272	Neural network training
doduc	Fortran	5,334	Monte Carlo simulation of a nuclear reactor
ear	C	5,237	Human ear simulation
fpppp	Fortran	2,718	Quantum chemistry code
hydro2d	Fortran	4,448	Astrophysics simulation using Navier-Stokes equations
mdljdp2	Fortran	3,883	Molecular dynamics model (double precision)
mdljsp2	Fortran	4,456	Molecular dynamics model (single precision)
nasa7	Fortran	1,177	Seven floating-point intensive kernels
ora	Fortran	533	Ray tracing
spice2g6	Fortran/C	18,912	Analog circuit simulator
su2cor	Fortran	2,514	Quantum physics code
swm256	Fortran	487	Shallow-water simulation
tomcatv	Fortran	195	Vectorized mesh generation code
wave5	Fortran	15,062	Maxwell's equations and particle equations of motion

Compiler options

We used the flags shown in Table A to compile the programs for the Cray Research C90 processor. The flags represent a straightforward, reasonable use of the C90 compiler but do not necessarily represent the best performance attainable on the C90.

Table A. Cray C90 compiler options.

Program	Compiler options
spice2g6	-Wf'-m4 -astatic' -dp
doduc	-Zv -Wd-e7 -Wf'-m4' -dp
fpppp	-Zv -Wd'-J2 -M800' -Wf'-m4' -dp
ora	-Wf'-m4 -i46 -oinline,aggress' -dp
mdljdp2	-Zv -Wd'-ddc -e78 -M200' -Wf-m4 -dp
wave5	-Zv -Wd-dd -Wf'-a static -m4' -dp
mdljsp2	-Zv -Wd'-ddc -e78 -M200' -Wf-m4 -dp
alvinn	-O3
nasa7	Zv -Wf-m4 -dp
ear	-O3 -h restrict=f,ivdep
hydro2d	-Zv -Wd'-fp -M100 -e78' -Wf-m4 -dp
su2cor	-Zv -Wf-m4 -dp
tomcatv	-Zv -Wf-m4 -dp
swm256	-Zv -Wd-e7 -Wf'-m4' -dp

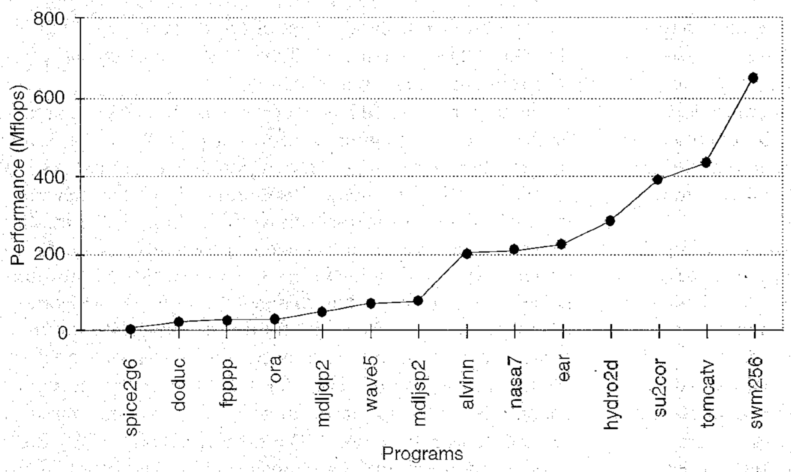


Figure 1. SPEC92fp performance on a Cray C90 vector processor.

Figure 1 shows the performance of the SPEC92fp programs on a Cray Research C90 processor in terms of Mflops (million floating-point operations per second). The Compiler options box lists the flags we used to compile the programs.

The C90 obtained a SPEC92fp ratio of 540. We calculated the Mflops rates shown in Figure 1 using a set of reference floating-point operation counts we obtained by running the optimized program on a simple RISC microprocessor. A reference floating-point count provides a common basis for comparison between machines, since hardware features such as masked vector operations and predicated or speculative executions tend to inflate the number of floating-point operations executed. For the graph, we have sorted the programs in ascending order of Mflops rates achieved.

Our results indicate that the attained Mflops rates vary from a few Mflops for spice2g6 to 646 Mflops for swm256. Roughly half the benchmarks are successfully vectorized, achieving performance in excess of 200 Mflops; the poorly vectorized programs execute at less than 75 Mflops. The vectorizable programs, because of their inherent parallelism, are likely to

depends heavily on the program's structure and how it is written. To get some insight into the program structure and provide a means of calibrating the performance results, we measure how well these programs are vectorized. Successfully vectorized programs tend to be dense matrix computations and contain simple, easily parallelizable loops. Conversely, programs that are not vectorized present a greater challenge to the parallelizer.

run well on a multiprocessor. Thus, to highlight the relationship between program structure and parallelizability, we maintain this order for the SPEC92fp programs when presenting the experimental results.

Analyzing the SUIF parallelizer. We used the SUIF compiler to parallelize the SPEC92fp programs, generating SPMD code in a combination of C and Fortran. We fed this code to the native compiler on the Digital 8400, using the "best" flags as reported by Digital whenever meaningful. If SUIF failed to find significant parallelism in a program, we passed the original sequential program to the native compiler unchanged.

To analyze the success of parallelization, we present three sets of experimental results in Figure 2. Figure 2a shows the parallel coverage, defined as the percentage of the original (sequential) computation found to be executable in parallel. Figure 2b shows the granularity of parallelism, defined as the average sequential execution time of the loops identified as parallel. Finally, Figure 2c shows the relative speedups of the applications on a four-processor Digital 8400 machine. These graphs highlight the importance of the advanced analysis techniques by showing the results we obtained with the full SUIF compiler as well as those we obtained using the first-generation parallelization techniques alone.

Parallel coverage. High parallel coverage indicates that the compiler analysis located significant amounts of parallelism in the computation—a prerequisite to efficient parallel performance. By Amdahl's law, even a program with as much as 80 percent coverage cannot achieve more than a speedup of 2.5 on four processors and 3.3 on eight processors. Figure 2a shows that the first-generation parallelizer finds almost no parallelism beyond vectorizable loops in the Fortran programs.

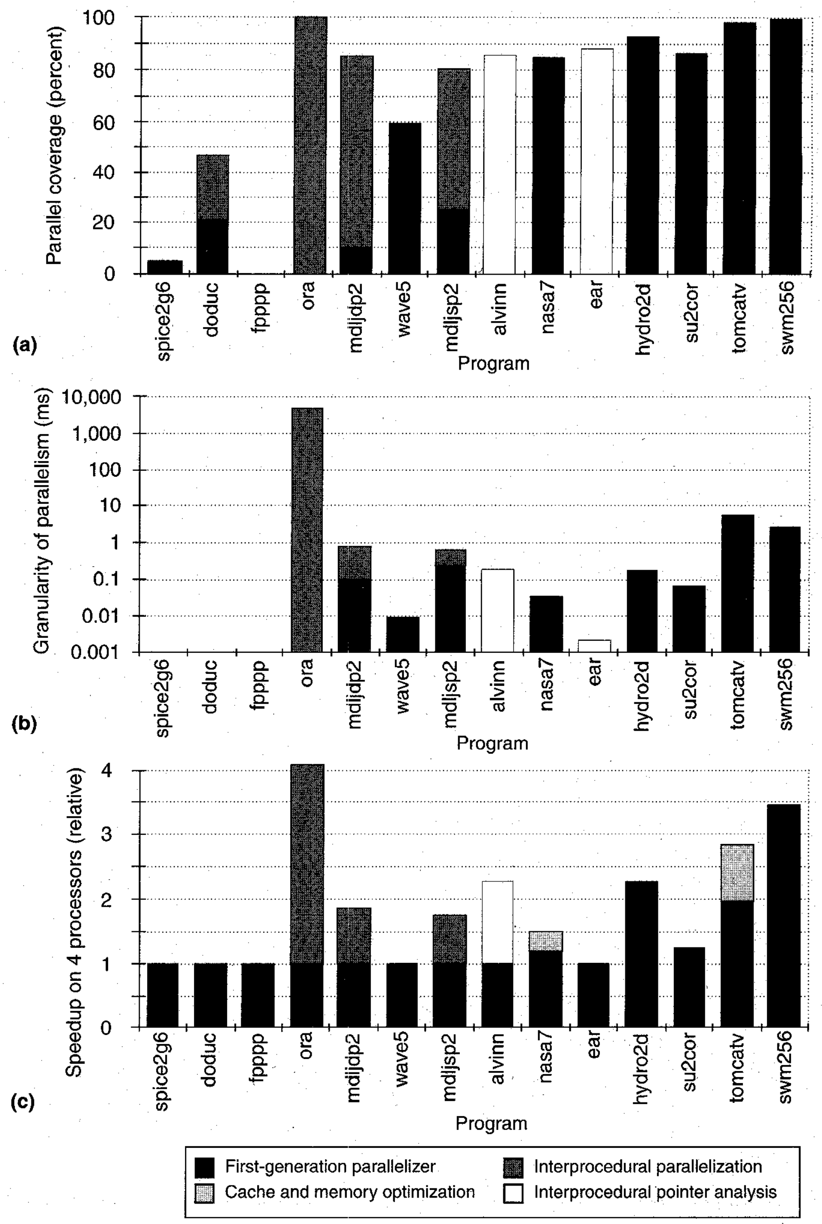


Figure 2. SPEC92fp parallelization with SUIF: parallel coverage (a), granularity of parallelism (b), and speedup on four processors (c).

With all its advanced techniques, SUIF is fairly successful in locating parallelism in SPEC92fp. The C pointer analysis algorithm allows SUIF to locate loop level parallelism in the two C programs (alvinn and ear). SUIF also finds significant parallelism in about half of the nonvectorizable programs as a result of the advanced array analyses and interprocedural analysis. Ten of the 14 SPEC92fp programs have a coverage of at least 80 percent.

SUIF fails, however, to find much parallelism in three of the SPEC92fp programs. Examination reveals that each of these programs is written in a convoluted programming style that obscures the original program semantics. In fact, two of these programs contain no significant loop level parallelism without a major rewrite of the algorithm. In conclusion, SUIF is effective in finding the statically analyzable parallelism in

Table 2. SPEC92fp execution times (in seconds).

Program	Cray C90	Reported*	SUIF parallelized (no. of processors) 1	4	8
spice2g6	498.6	102.5	102.5	102.5	102.5
doduc	15.0	4.9	4.9	4.9	4.9
fpppp	75.8	14.1	14.1	14.1	14.1
ora	32.8	20.0	19.7	4.9	2.5
mdljdp2	21.1	16.0	15.6	8.6	6.9
wave5	14.2	11.8	11.8	11.8	11.8
mdljsp2	21.7	14.1	14.2	8.0	6.4
alvinn	4.8	8.0	8.6	3.5	2.9
nasa7	10.5	26.5	40.0	17.5	17.4
ear	12.9	20.0	20.0	20.0	20.0
hydro2d	6.2	23.8	27.0	10.4	8.8
su2cor	4.3	17.6	22.7	14.1	15.9
tomcatv	1.0	3.7	5.0	1.3	0.7
swm256	8.2	29.0	31.2	8.4	4.6
SPEC ratio	540	506.4	464.7	845.2	1,016.1

* One processor

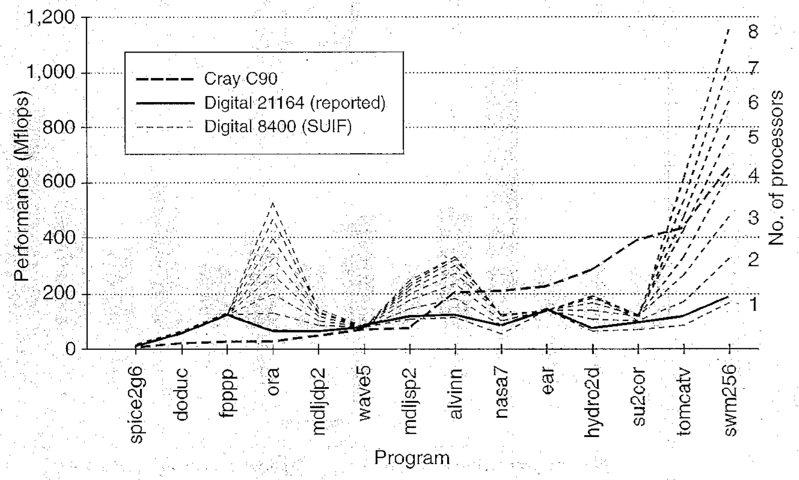


Figure 3. SPEC92fp performance.

the SPEC92fp suite.

Granularity of parallelism. A program with high parallel coverage does not necessarily achieve a high speedup. Synchronization and communication overhead may outweigh the performance gain of executing the loop in parallel. While it is difficult to measure the precise parallelization overhead, we can estimate its potential cost by measuring the granularity of parallelism. Figure 2b shows a log-scale graph of the granularity of parallelism that SUIF and the first-generation parallelizer obtained. Granularity results for mdljdp2 and mdljsp2 for the first-generation compiler are irrelevant as its coverage for those programs is very low (under 30 percent).

The results show that programs with outer-loop parallelism tend to have a higher granularity. The vectorizable codes show a varied granularity of parallelism, ranging from a few microseconds to over a second. Considering that SPEC92fp programs have relatively small data sets and short running times, it is not surprising that some of the granularities are very small.

Relative speedup. Figure 2c shows that the first-generation parallelization techniques can only speed up five of the 14 SPEC92fp programs, and all of them are vectorizable programs. With the advanced research components, SUIF speeds up four more programs, bringing the total number of programs that benefit from parallelization to nine. As we can see from the figures, a program's speedup factor is highly correlated to its parallel coverage and its granularity of parallelism.

Due to its fine granularity of parallelism, ear is the only program that does not speed up despite its high parallel coverage. Although su2cor and hydro2d also suffer from fine-grained parallelism, they do speed up with increased data set size, as demonstrated by the SPEC95fp programs (see following section). The execution times of nasa7 and tomcatv improve due to locality optimizations. Finally, mdljsp2, mdljdp2, and ora demonstrate the success of SUIF in exploiting coarse-grained parallelism.

SPEC92fp on Digital 8400. The Digital 21164 is a quad-issue superscalar microprocessor, with two 64-bit integer and two 64-bit floating-point pipelines. At a clock rate of 300 MHz, its peak performance is 1,200 MIPS or 600 Mflops. The processor has on chip an 8-Kbyte instruction cache, an 8-Kbyte data cache, and a 96-Kbyte combined second-level cache. The memory system allows multiple outstanding off-chip memory accesses. The Digital 8400 is a bus-based, shared-memory multiprocessor containing up to twelve 21164 processors. Besides the two-level caches on chip, each processor has 4 Mbytes of 10-ns external cache. The 256-bit data bus, which operates at 75 MHz, supports 265-ns memory read latencies. We performed our experiments on an 8400 with 1 Gbyte of shared memory configured as two banks with 512 Mbytes each.

Table 2 shows the performance data of the SPEC92fp programs for the Cray C90, the SPEC92fp results reported by Digital for a single 21164 processor, and the performance results obtained by the SUIF compiler for one, four, and eight processors on the 8400. Due to parallelization overheads and the ineffectiveness of the native compiler in dealing with SUIF-generated code, some of the SUIF-generated uniprocessor programs run slower than the vendor-reported times. The SPEC92fp ratio measured on a single processor running the parallel code is only 465—closer to Digital's reported base ratio of 437 than its best ratio, 506. Despite the loss in uniprocessor performance, the SUIF-parallelized code on the 8400 achieves an impressive SPEC92fp rating of 845 for four processors, and 1,016 for eight processors.

Figure 3 shows Mflops rates (with the same reference floating-point operation count mentioned earlier) calculated for a Cray C90 processor, a single 21164 processor using the reported data, and the 8400 running with different numbers of processors. The three programs with the highest parallel coverage and coarsest granularity of parallelism (ora, tomcatv, and swm256) speed up linearly as the number of processors increases. For the rest of the programs, the improvement due to parallelization decreases as the number of processor increases. We expected the phenomenon of decreasing marginal return, since the SPEC92fp programs have small data set sizes; all the programs but spice2g6 run in under 30 seconds on a single 21164 processor.

The experimental results suggest that building multiprocessors out of fast microprocessors is an effective technique for achieving high performance. A single 21164 processor executes the SPEC92fp programs with performance in excess of 60 Mflops consistently, and above 100 Mflops in many cases. It outperforms the C90 on half the benchmark suite and delivers very competitive SPEC92fp performance. The overall performance of the multiprocessor is impressive, with six programs attaining over 200 Mflops on eight processors. A small number of 21164s can beat the C90 on the most vectorizable programs, with the eight-processor system attaining 613 Mflops for tomcatv and 1.15 Gflops for swm256. The multiprocessor further extends the microprocessor's lead over the vector machine for some of the nonvectorizable codes (for example, ora and mdljsp2).

The multiprocessor's weakness is in supporting parallel codes that operate on small vectors, as the su2cor and ear applications illustrate. Today's multiprocessors do not support fine-grained parallelism well; the three levels of private per-processor cache in the 8400 system make data sharing between processors very expensive. Better support, such as a shared second-level cache, can improve future performance as multichip module implementation techniques mature or as the level of integration increases.[9]

Finally, there are always programs that are not amenable to parallelization (for example, spice2g6 and doduc). However, because of complex control flow and data structures, these programs are not amenable to other performance enhancements such as vectorization or instruction level parallelism either. While multiprocessors cannot speed up such individual applications, the processors can execute different programs or instances of the same program at the same time. For example, the usefulness of running multiple Spice simulations at the same time is clear. This option makes the multiprocessor a more versatile architecture than a very wide superscalar machine.

Table 3. SPEC95fp execution times (in seconds).

	Lines			SUIF parallelized (no. of processors)		
Program	(no.)	Description	Reported*	1	4	8
fpppp	2,784	Quantum chemistry code	445.0	445.0	445.0	445.0
apsi	7,361	Pseudospectral air pollution model	151.0	151.0	151.0	151.0
wave5	7,764	Maxwell's equations and particle equations of motion	193.0	193.0	193.0	193.0
su2cor	2,332	Quantum physics code	185.0	227.4	86.2	78.9
applu	3,868	Parabolic/elliptic partial differential equation solver	333.0	380.4	124.3	86.8
mgrid	666	Multigrid solver for computing 3D potential field	258.0	293.1	73.4	48.5
tomcatv	190	Vectorized mesh generation code	241.0	295.2	65.9	44.2
turb3d	2,100	Isotropic, homogeneous turbulence simulation	373.0	383.3	103.8	60.9
hydro2d	4,292	Astrophysics simulation using Navier-Stokes equations	300.0	294.1	76.9	48.0
swim	429	Shallow-water simulation	371.0	420.3	80.9	37.9
SPEC ratio			12.2	11.2	28.4	38.4

* One processor

SPEC95fp

Table 3 describes the SPEC95fp programs and lists the performance results we obtained running SUIF-parallelized codes on one, four, and eight processors. We have sorted

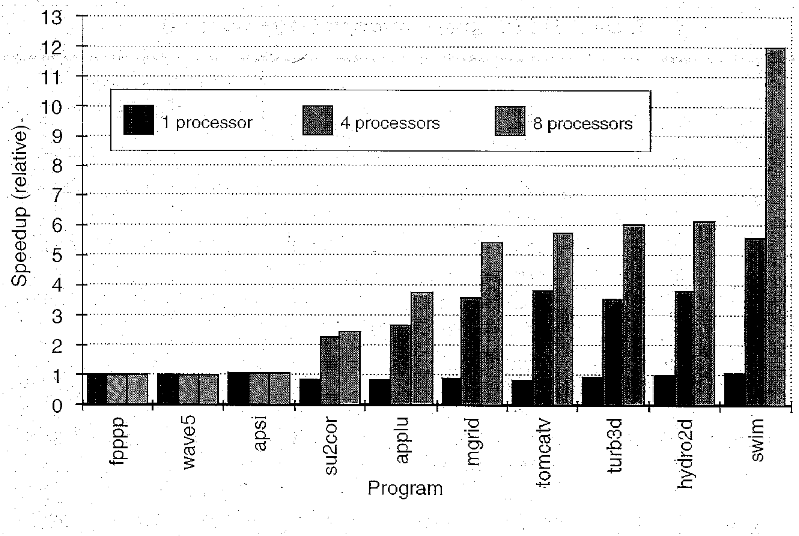


Figure 4. Performance of SPEC95fp.

the programs in ascending order of their speedups, shown in Figure 4.

SUIF speeds up seven of the 10 SPEC95fp programs. Programs fpppp, apsi, and wave5 suffer from poor parallel coverage. The su2cor program has 90 percent coverage, but its average granularity of parallelism is less than 200 ms. For applu, SUIF achieves the speedup shown by parallelizing an outer loop using array privatization and blocking (see the earlier Case study box). The rest of the programs achieved almost a perfect speedup on four processors. In fact, swim speeds up superlinearly primarily because of the improvement in cache performance as each processor accesses less data. The programs continue to show improvement as the number of processors increases to eight.

Our results show that the multiprocessor organization is even more successful in improving the performance of SPEC95fp applications than of SPEC92fp. Moreover, the results for the SPEC95fp suite are more significant, as they are more representative of realistic workloads. Parallelization lowers the SPEC95fp ratio for one processor slightly from 12.2 to 11.2. However, it improves the performance by a factor of 2.3 on four processors, yielding a SPEC95fp ratio of 28.4. On eight processors, there is a gain of 3.2 times, yielding a SPEC95fp ratio of 38.4.

ARE MULTIPROCESSORS VIABLE, from a software perspective, as the next generation of microarchitecture? Automatic parallelization techniques are indeed mature enough to parallelize both Fortran and C programs that contain high degrees of instruction level parallelism. Compilers can now find parallel loops beyond the inner loops and across function calls, and can thus take advantage of the multiprocessor's unique ability to execute different threads at the same time. These outer parallel loops yield coarser granularity, which generally leads to higher performance. Small data set sizes can create parallel loops that are too fine-grained to run efficiently on today's multiprocessors, but future integration of multiple processors on a chip would help alleviate this problem.

In summary, multiprocessors provide a cost-effective and versatile general-purpose computational platform. The modularity of multiprocessors makes them relatively cheap and easy to build, leading to a faster clock and a shorter time to market. For parallelizable applications, they can speed up a single application's execution; for sequential applications, they achieve efficient machine use and deliver higher system throughput by executing multiple programs at a time. Because they are effective for both types of applications, multiprocessors do not suffer as much from diminishing returns of increased complexity as do other architectural techniques for high performance. The multiprocessor is already a proven architecture among mainframes and workstations; it becomes ever more affordable now that multiprocessor PCs are available on the market. As automatic parallelization technology matures and the hardware technology improves, the multiprocessor is a logical candidate for the next-generation microarchitecture.

Acknowledgments

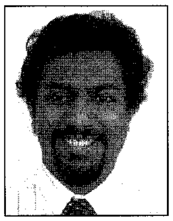
We thank Robert Cohn, Zarka Cvetanovic, Tryggve Fossum, David Hunter, Jeff Lowney, and John Shakshober for giving us access to the Digital multiprocessors and help with the machines. Charles Grassl helped us with the Cray Research C90 performance data, and Ed Bugnion with performance tuning. Finally, for their contributions we thank the rest of the SUIF group—Bob Wilson, David Heine, Chau-Wen Tseng, Patrick Sathyanathan, Alex Seibulescu, and Amy Lim.

The research was supported in part by the Air Force Material Command and ARPA contract F30602-95-C-0098, an NSF Young Investigator award, an NSF CISE postdoctoral fellowship, and fellowships from AT&T Bell Laboratories, DEC Western Research Laboratory, and Intel Corporation.

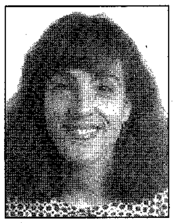
References

1. B.P. Wilson et al., "SUIF: A Parallelizing and Optimizing Research Compiler," *ACM SIGPLAN Notices*, Vol. 29, No. 12, Dec. 1994, pp. 31-37.
2. B.P. Wilson and M.S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Association for Computing Machinery, New York, 1995, pp. 1-12.
3. M.W. Hall et al., "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," *Proc. Supercomputing*, IEEE, New York, 1995.
4. P. Tu and D. Padua, "Automatic Array Privatization," *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, 1993, pp. 500-521.

5. W. Blume et al., "Polaris: The Next Generation in Parallelizing Compilers," *Proc. Seventh Workshop on Languages and Compilers for Parallel Computing,* Springer-Verlag, 1994, pp. 141-154.
6. J.M. Anderson and M.S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* ACM, 1993, pp. 112-125.
7. M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* ACM, 1991, pp. 30-44.
8. 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, 1995, pp. 166-178.
9. B.A. Nayfeh and K. Olukotun, "Exploring the Design Space for a Shared-Cache Multiprocessor," *Proc. 21st Int'l Symp. Computer Architecture,* IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 166-175.

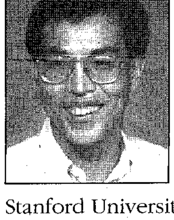


Saman P. Amarasinghe is a PhD candidate in Stanford University's Electrical Engineering Department. His research interests include compilers and computer architecture. Amarasinghe received a BS in electrical engineering and computer science from Cornell University, and an MS in electrical engineering from Stanford University.



Jennifer M. Anderson is a PhD candidate in the Computer Science Department at Stanford University. Her research interests include compiler algorithms, computer architecture, and parallel and distributed computing. Anderson received a BS in information and computer science from the University of California, Irvine, and an MS in computer science from Stanford University.

Christopher S. Wilson is a staff compiler developer in the Stanford SUIF Compiler group. Wilson received a BS in math and computer systems engineering and an MS in computer science from Stanford University.



Shih-Wei Liao is a PhD candidate in the Electrical Engineering Department at Stanford University. His research interests include compilers and computer architecture. Liao received a BS in computer science from National Taiwan University and an MS in electrical engineering from Stanford University.

Brian R. Murphy is a PhD candidate in the Computer Science Department at Stanford University. His research interests include aggressive program analysis and programming languages and environments. Murphy received a BS in computer science and an MS in electrical engineering and computer science from the Massachusetts Institute of Technology.

Robert S. French is a PhD candidate in the Computer Science Department at Stanford University. His research interests include the compilation of event-driven hardware description languages. French received a BS in computer science from the Massachusetts Institute of Technology.

Monica S. Lam is an associate professor in the Computer Science Department at Stanford University. She currently leads the SUIF research project at Stanford University. Lam, recipient of an NSF National Young Investigator award, earned a BS from the University of British Columbia and a PhD in computer science from Carnegie Mellon University.

Mary W. Hall is a senior research fellow in the Department of Computer Science at the California Institute of Technology. Her research interests focus on compiler support for high-performance computing, particularly interprocedural analysis and automatic parallelization. Prior to joining Caltech, Hall was a research scientist at Stanford University working with the SUIF compiler group. Hall received a BA in computer science and mathematical sciences and MS and PhD degrees in computer science from Rice University.

Direct questions concerning this article to Saman Amarasinghe at Gates Building 406, M/C 9030, Stanford University, Stanford, CA 94305; saman@wildhog.stanford.edu.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 165	Medium 166	High 167