to appear in IEEE Micro, April 1996

The Multiprocessor as a General-Purpose Processor: A Software Perspective

Saman P. Amarasinghe, Jennifer M. Anderson, Christopher S. Wilson, Shih-Wei Liao,
Mary W. Hall[†], Brian R. Murphy, Robert S. French, Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford CA 94305

Abstract

Multiprocessor systems have become common place, but little software takes advantage of their capabilities. Automatic parallelization is particularly attractive as it enables sequential code to exploit parallel hardware and realize improved performance, without additional programmer effort. This paper demonstrates that automatic parallelization techniques are now mature enough to parallelize many numeric programs written in both Fortran and C. Using these techniques, the SPEC92fp and SPEC95fp benchmarks were successfully parallelized and run on an 8-processor Digital AlphaServer 8400 machine to obtain the highest recorded SPEC92fp and SPEC95fp ratios. The capabilities of state-of-the-art parallelizing compilers should be taken into account in future processor design. A multiprocessor in combination with a parallelizing compiler may outperform approaches to processor design which attempt to exploit increasing levels of instruction-level parallelism.

1 Introduction

Like many architectural techniques that originated with mainframes, the use of multiple processors in a single computer is becoming popular among 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 the familiar situation where hardware is ahead of software. Because of the complexity of parallel programming, multiprocessors today are rarely used to speed up individual applications, but are mainly used as compute-servers to achieve increased system throughput by running multiple tasks simultaneously. Parallelization performed automatically 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 any user involvement. This article looks to the future by examining some of the latest research results in compiler technology.

The need to take state-of-the-art compiler technology into account when designing next-generation machines is well demonstrated by the success of the RISC revolution. The effectiveness of parallelizing compilers, besides being

---

[†]currently at Computer Science Dept. California Institute of Technology

This 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.

important for existing systems, has significant implications for future machine design. Once the multiple processors are transparent to the programmer, multiprocessors are as easy to use as traditional uniprocessors. The multiprocessor can then be evaluated solely on price-performance, and additional processors are treated just like other performance enhancement features such as the cache organization or the instruction pipeline.

This perspective is particularly significant in the design of future micro-architectures. In the quest for higher microprocessor performance, the current trend is to build wider superscalar or VLIW machines. As more functional units are added to a processor, however, fewer programs are capable of using all of the units effectively, and the marginal return of the additional hardware decreases. In fact, the higher processor complexity can increase the machine's cycle time, which may even slow down the performance of the programs that do not take advantage of the increased parallelism. Currently, only regular numeric programs have been shown to be successful in exploiting instruction level parallelism (ILP) at a higher level than that offered in today's microprocessors. As shown in this paper, many of these programs can also be automatically parallelized to run efficiently on a multiprocessor. The compiler can sometimes even find multiprocessor-usable parallelism in programs that exhibit little ILP. More importantly, the additional hardware cost in a multiprocessor is not wasted on inherently sequential applications, as they can still be run in a multiprogrammed mode to deliver better system throughput. The multiprocessor also offers several hardware implementation advantages over building a single large and complex processor. The modularity of the multiprocessor allows the system design to be based on a replicated processor core, possibly an already existing one. The reduced design complexity of the multiprocessor design makes a faster clock speed and a shorter time-to-market possible.

We support the view that multiprocessors can be used effectively to parallelize individual applications with experimental data obtained using the SUIF (Stanford University Intermediate Format) compiler[7]. SUIF is a research parallelizing compiler developed at Stanford University over the last seven years. The domain in which the compiler is effective is 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.

Our experiments used SUIF to parallelize the SPEC92 and SPEC95 floating-point applications for the Digital AlphaServer 8400 multiprocessor. The SPEC programs are a set of sequential programs that are commonly used to benchmark computer systems and are not applications tailored to run on a multiprocessor. The 8400 uses the 300 MHz Digital 21164 Alpha processor, which is a leader in SPEC performance among microprocessors currently available. As it is hard to achieve speedups on machines with fast processors, this choice presents a greater challenge to parallelization while making our conclusions more likely to apply to systems with future processors. Our experimental results show that SUIF is successful in boosting the highest reported SPEC92fp ratio of 506 to 1016 by parallelizing the code for eight processors. The SPEC95fp ratio is raised 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 many numeric applications can be parallelized effectively.

Our results suggest that the multiprocessor, with the help of a parallelizing compiler, offers serious competition to the design of very wide superscalar or VLIW machines. Since positive results were obtained on multiprocessors that have already been built, they provide stronger support for such a conclusion than do the simulation results of 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.

This paper is organized as follows. We first present an overview of the SUIF compiler in Section 2, including

an explanation of some of its advanced parallelization techniques. In Section 3, we provide some background data on the SPEC92fp benchmark suite, show how well SUIF parallelizes programs in the suite, and show their performance as compared to uniprocessor systems. Section 4 presents similar data for the SPEC95fp suite. We analyze the results and pinpoint several potential problem areas that need to be addressed in future architectural research. This is particularly useful for future multiprocessor-on-a-chip designs because of the many possibilities that are made possible by new processor implementation technology. We close with some concluding remarks in Section 5.

2 The 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) code for shared memory multiprocessors. For the experiments described in this paper, we generated a combination of C and Fortran source code; C code is used mainly to coordinate the parallel activity and Fortran code is used for most of the computational sections of the program.

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 uses *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 to recognize reduction operations (e.g. summations and products), to convert a scalar variable into private copies on each processor, and to transform loop nests (such as interchanging inner and outer loops) for the sake of parallelism and locality. We refer to this collection of analyses and optimizations as the "first-generation" parallelization techniques. A system consisting of these basic parallelization techniques was released and made publicly available in 1994, and can be retrieved via the World Wide Web at `http://suif.stanford.edu`.

Besides these conventional techniques, the SUIF system includes three unique experimental components, which will also be made 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 multiprocessors have the unique ability to execute loops containing complex control flow in parallel, we have extended previous techniques to identify outer-loop parallelism. Finally, as memory behavior can significantly impact performance on a shared-memory architecture, SUIF contains a suite of locality optimization techniques to make the multiprocessor caches more effective. Two case studies from the SPEC95fp benchmark suite are presented in Figure 1 and Figure 2.

2.1 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 even allows programmers 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 they can be reordered in a parallel execution. The analysis to determine if pointer variables can 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 that the pointer analysis be applied 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 can be made efficient by summarizing the effect of each function succinctly, but pointer analysis is not amenable to such an optimization.

Building upon the results by many researchers in the area, we have developed a new pointer analysis algorithm that finds context-sensitive pointer alias results efficiently[8]. Our approach is to identify the calling context aliases that are relevant to the procedure's behavior and then analyze the procedure once for all the 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 a lot of further progress in pointer alias analysis and its applications in the near future.

2.2 Finding Coarse-Grain 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.

The main computation in the SPEC95fp program turb3d is a series of loops that compute three-dimensional FFTs. While these loops are parallelizable, they all have a complex control structure containing large amounts of code, as shown below. The boxes represent procedures and the lines represent procedure invocations. Each parallel loop, as indicated in the diagram, consists of over 500 lines of code spanning eight or nine procedures, with up to 42 procedure calls. It is necessary to parallelize these outer loops to get any significant speedup. The key to discovering the parallelism is interprocedural array analysis. The compiler is able to determine 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 speeds up a 4-processor machine by over 3.5 times

.

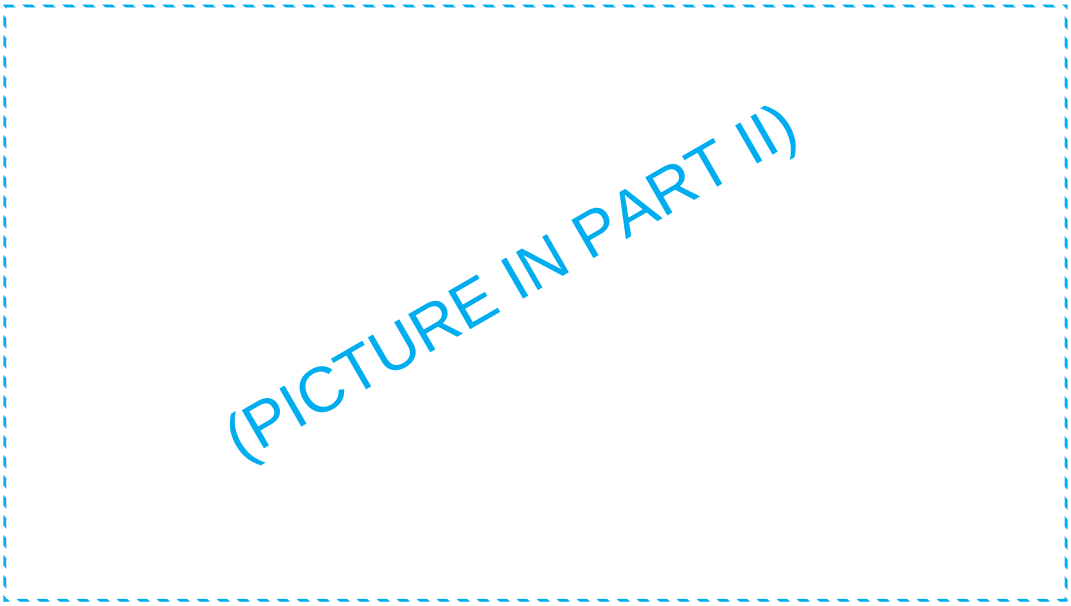

Figure 1 CASE STUDY: Turb3d

The primary routines in the SPEC95fp program applu calculate the lower and upper triangle solutions to a series of partial differential equations. An excerpt of the code for the main loops in these routines is shown below.

```
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 the same array `tmat` is defined and used in each of the iterations of the three outer loops. 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 stage of the pipeline. While the blocking transformation is well-known to be useful for improving cache locality, it is also useful also to minimize synchronization and communication in multiprocessors. These transformations yield a speedup of over 2.5 times on a 4-processor machine. This example illustrates the importance of a well-integrated set of advanced techniques.

Figure 2 CASE STUDY: Applu

2.2.1 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 used in the computation. For example, it is very common for each iteration of a loop to assign and then use the same variable. The compiler must give each processor a private copy of the variable for the loop to be parallelizable. As another example, a loop may contain a reduction (e.g., computation of a sum, product, or maximum over a set of data elements) which 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, it is sufficient to privatize scalar variables and to transform reductions that write to scalar variables. To find outermost parallel loops, however, these analyses must be extended to array variables[4][6].

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 iterations of a loop are operating on different array elements. For the former, it is acceptable that iterations are operating on the same locations, but the *values* generated by one iteration must not be used by another.

The SUIF parallelizer also performs array reduction recognition. The 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 even parallelize reductions that operate on sparse arrays.

2.2.2 Interprocedural Parallelization Analysis

To find parallelism in outer loops, it is essential that the compiler analyze across procedure boundaries. A simple way to eliminate procedure boundaries is to perform inline substitution---replacing each procedure call by a copy of the called procedure---and perform program analysis in the usual way[3]. 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[4]. The compiler includes a set of interprocedural analyses on scalar variables including dependence and privatization analysis and reduction recognition. It also includes several interprocedural analyses to assist the array analysis, including constant propagation, induction variable elimination, recognizing loop invariant computations, and symbolic relation propagation. Finally, the compiler performs array data dependence analysis and all the advanced array analyses above in an interprocedural manner. This set of optimizations has been shown to be powerful enough to parallelize loops spanning hundreds of lines of codes and numerous non-trivial 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. Analysis of procedure side-effects is separated 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 the technique of *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.

2.3 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, data are often displaced before they can be re-used. This problem is exacerbated in cache-coherent multiprocessors as data sharing between processors introduces even more cache traffic. 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 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 the computation across processors so as to minimize interprocessor communication[1]. The computation executed by each processor is reordered via unimodular loop transformations and blocking to enhance data locality[9]. We have also developed an interprocedural algorithm that changes the array data layout in order to minimize unnecessary cache traffic[2]. The arrays are reorganized so that the data accessed by the same processor is allocated in contiguous locations. For example, the compiler may change the organization of an array from column-major to row-major, or restructure a 2D array as a 3D array. All the loop and data transformation algorithms are developed within a unified theoretical framework based on linear algebra. Experimental results show that this technique can significantly improve program performance. We are currently developing the necessary analyses and transformations to make the technique more widely applicable.

3 Parallelization of SPEC92fp Programs

The SPEC92fp benchmark suite consists of fourteen floating-point applications which are briefly described in Table 1. 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.

Programs	Language	Lines	Description
alvinn	C	272	Neural network training
doduc	FORTRAN	5334	Monte Carlo simulation of a nuclear reactor
ear	C	5237	Human ear simulation
fpppp	FORTRAN	2718	Quantum chemistry code
hydro2d	FORTRAN	4448	Astrophysics simulation using Navier Stokes equations
mdljdp2	FORTRAN	3883	Molecular dynamics model (double-precision)
mdljsp2	FORTRAN	4456	Molecular dynamics model (single-precision)
nasa7	FORTRAN	1177	Seven floating-point intensive kernels
ora	FORTRAN	533	Ray tracing
spice2g6	FORTRAN/C	18912	Analog circuit simulator
su2cor	FORTRAN	2514	Quantum physics code
swm256	FORTRAN	487	Shallow water simulation
tomcatv	FORTRAN	195	Vectorized mesh generation code
wave5	FORTRAN	15062	Maxwell's equations and particle equations of motion

Table 3 Characteristics of SPEC92fp programs

The amount of parallelism that can be recognized depends heavily on the structure of the program and how it is written. To get some insight into the program structure and to provide a means to calibrate the performance results, we measure how well these programs are vectorized. Programs that are vectorized successfully tend to be dense matrix computations, and contain simple parallelizable loops. Conversely, programs that are not vectorized present a greater challenge to the parallelizer.

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 programs have been compiled using the flags shown in Appendix I; the flags represent a straightforward and reasonable use of the C90 compiler but do not necessarily represent the best performance attainable on the C90. The SPEC92fp rating obtained is 540. The MFLOPS rates shown are calculated using a set of reference floating-point (FP) operation counts obtained by running the optimized program on a simple microprocessor. A reference FP count is used to provide 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 FP operations executed. The programs are sorted 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 of the benchmarks are successfully vectorized, achieving a 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 run well on a multiprocessor. Thus, we will maintain the order of the SPEC92fp programs when presenting the experimental results so as to highlight the relationship between program structure and parallelizability.

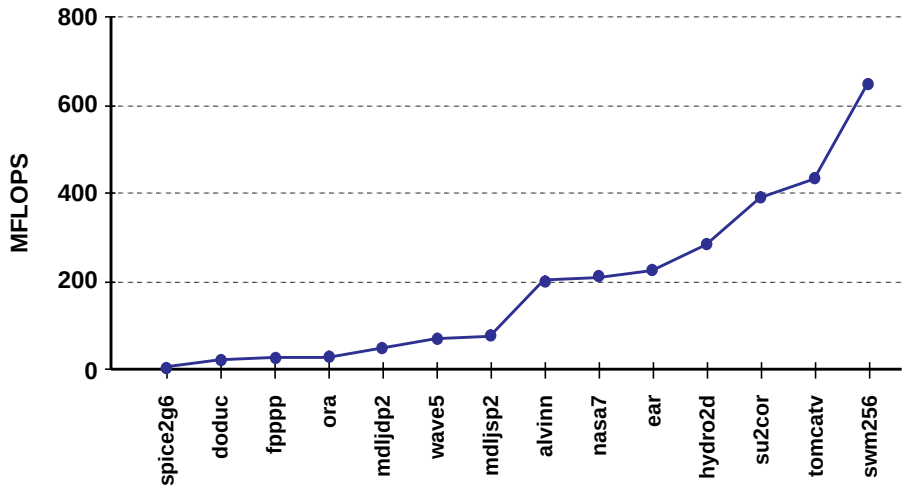

Figure 4: Performance of SPEC92fp on a Cray C90 vector processor

3.1 Analysis of the SUIF Parallelizer

We used the SUIF parallelizer to parallelize the SPEC92fp programs, generating SPMD code in a combination of C and Fortran. This code was fed 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, the original sequential program was passed to the native compiler unchanged.

To analyze the success of parallelization, we present three sets of experimental results in Figure 3. Figure 3(a) shows the *parallel coverage*, defined as the percentage of the original (sequential) computation found to be executable in parallel, Figure 3(b) shows the *granularity of parallelism*, defined as the average sequential execution time of the loops identified as parallel, and Figure 3(c) shows the relative speedups of the applications on a 4-processor Digital 8400 machine. The figures highlight the importance of the advanced analysis techniques by showing the results obtained by the full SUIF compiler as well as those obtained using just the first-generation parallelization techniques.

3.1.1 Parallel Coverage

High parallel coverage is indicative that the compiler analysis has located significant amounts of parallelism in the computation, and is a prerequisite to efficient parallel performance. By Amdahl's Law, even a program with as much as 80% coverage cannot achieve more than a speedup of 2.5 on 4 processors and 3.3 on 8 processors. Figure 3(a) shows that the first-generation parallelizer finds almost no parallelism beyond vectorizable loops in the Fortran programs.

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 non-vectorizable programs as a result of the advanced array analyses and interprocedural analysis. Ten out of the fourteen SPEC92fp programs have a coverage of at least 80%.

SUIF fails to find much parallelism in three of the SPEC92fp programs. Examination reveals that each of these programs has been 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 the SPEC92fp suite.

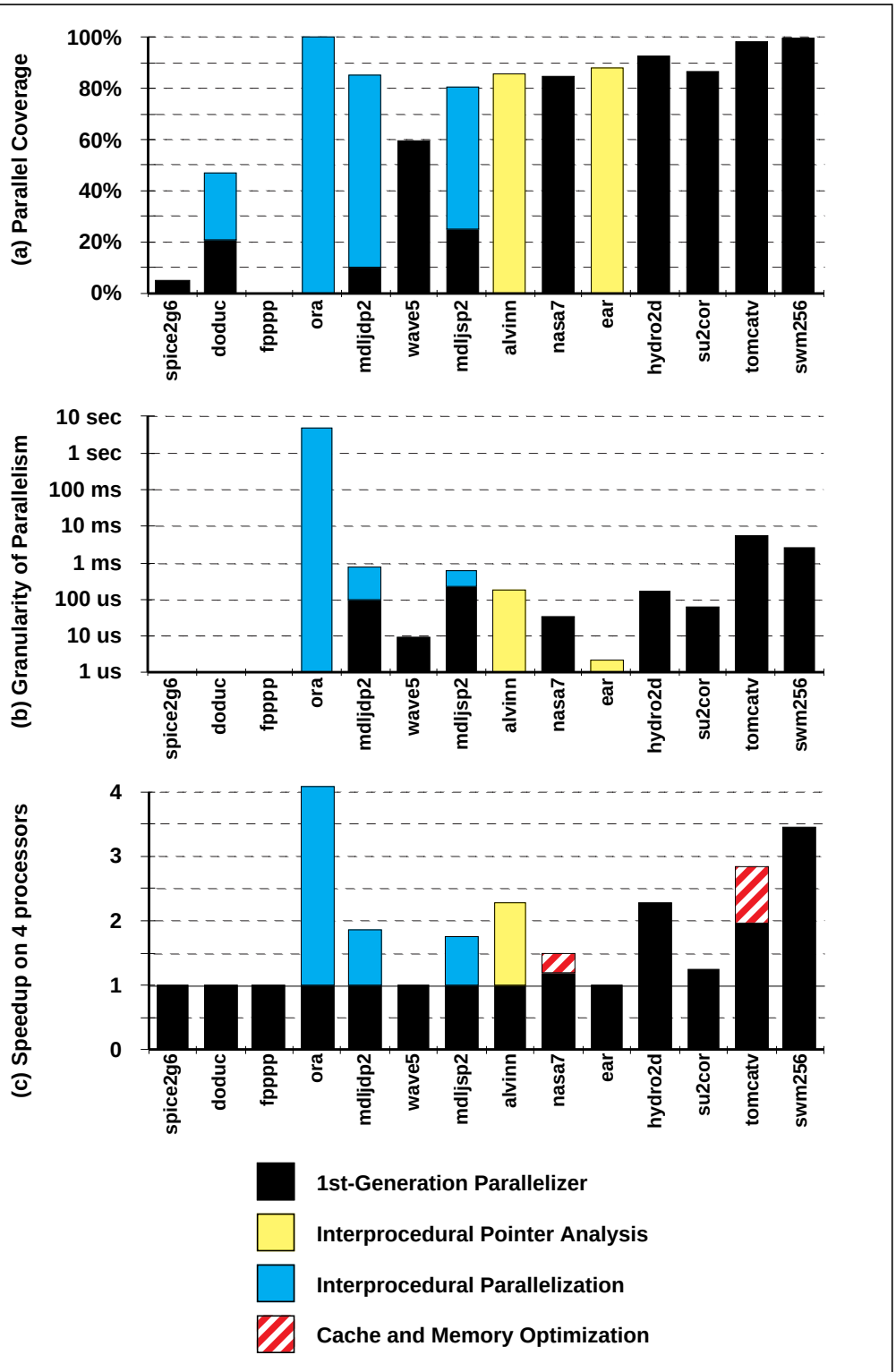

Figure 5: Parallelization of SPEC92fp using SUIF

3.1.2 Granularity of Parallelism

A program with high parallel coverage does not necessarily achieve a high speedup. The 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 3(b) shows a log-scale graph of the granularity of parallelism obtained by SUIF and by the first-generation parallelizer. Note that granularity results for mdljdp2 and mdljsp2 for the first-generation compiler are irrelevant as its coverage for those programs is very low (under 30%).

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.

3.1.3 Relative Speedup

Figure 3(c) shows that the first-generation parallelization techniques can only speed up 4 of the 14 SPEC92fp programs, and all of them are vectorizable programs. With the advanced research components, SUIF speeds up 5 more programs, bringing the total number of programs that benefit from parallelization to 9. 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-grain parallelism, they do speed up with increased data set size, as demonstrated by SPEC95fp programs. The execution times of nasa7 and tomcatv improve due to locality optimizations. Finally, the programs mdljsp2, mdljdp2, and ora demonstrate the success of SUIF in exploiting coarse-grain parallelism.

3.2 Performance of 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 300MHz, it has a peak performance of 1200 MIPS or 600 MFLOPS. The processor has on chip an 8KB instruction cache, an 8 KB data cache and a 96KB 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 4MB of 10ns external cache. The 256-bit data bus, which operates at 75MHz, supports 265ns memory read latencies. Our experiments were performed on a machine with 1 GB of shared memory, configured as two banks with 512 MB per bank.

Table 4 shows the performance data of the SPEC92fp programs for the Cray C90, the SPEC92fp results reported on a single 21164 processor, and the performance results obtained by the SUIF compiler for 1, 4, and 8 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 reported times. The SPEC92fp ratio measured on a single processor running the parallel code is only 465; it is closer to the Digital's reported base SPEC92fp ratio of 437 than its best SPEC92fp ratio of 506. Despite the loss in uniprocessor performance, the SUIF-parallelized code on the 8400 achieves an impressive SPEC92fp rating of 845 using 4 processors, and 1016 using 8 processors.

In addition to the SPEC92fp ratio, it is informative to present the performance results as MFLOPS rates using the same reference FP operation count as discussed in Section 3. Figure 5 shows the MFLOPS rates calculated for a Cray C90 processor, a single 21164 processor using the reported data, and the 8400 running with different numbers of

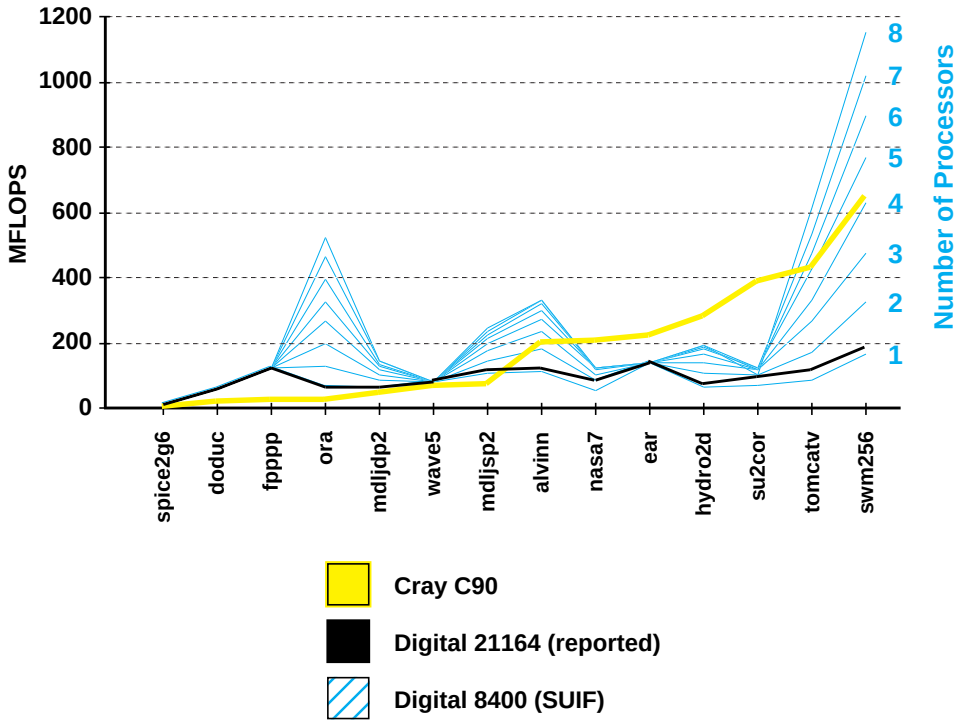

Figure 7: Performance of SPEC92fp

Programs	Cray C90	Reported (1 processor)	SUIF Parallelized		
			1 processor	4 processors	8 processors
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	1016.1

Table 6 Execution times (in seconds) for SPEC92fp Programs

processors. The figure shows that 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 increase. For the rest of the programs, the improvement due to parallelization decreases as the number of processor increases. The phenomenon of decreasing marginal return is expected since the SPEC92fp programs have small data set sizes; all the programs, with the exception of 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 a performance in excess of 60 MFLOPS consistently, and above 100 MFLOPS in many cases. It outperforms the C90 on half of the benchmark suite and delivers very competitive SPEC92fp performance. The overall performance of the multiprocessor is impressive, with 6 programs attaining over 200 MFLOPS using 8 processors. A small number of 21164s can beat the C90 on the most vectorizable programs, with the 8-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 non-vectorizable codes (e.g. ora, mdljsp2).

The multiprocessor's weakness is in supporting parallel codes that operate on small vectors, as illustrated by the su2cor and ear applications. Today's multiprocessors do not support fine-grain 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 be provided in the future as multi-chip module implementation techniques mature or as the level of integration increases[5].

Finally, there are always programs that are not amenable to parallelization (e.g. spice2g6 and doduc). However, with 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 individual applications, the processors can be used to 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.

4 SPEC95fp Performance

Table 1 shows a description of the SPEC95fp programs and the performance results obtained by running the SUIF-parallelized codes on 1, 4 and 8 processors. The programs are sorted in ascending order of their speedups, which are shown in Figure 7

Program	Lines of Code	Description	Reported (1 proc.)	SUIF parallelized Number of processors		
				1	4	8
fpppp	2784	Quantum chemistry code	445.0	445.0	445.0	445.0
apsi	7361	Pseudospectral air pollution model	151.0	151.0	151.0	151.0
wave5	7764	Maxwell's equations and particle equations of motion	193.0	193.0	193.0	193.0
su2cor	2332	Quantum physics code	185.0	227.4	86.2	78.9
applu	3868	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	2100	Isotropic, homogeneous turbulence simulation	373.0	383.3	103.8	60.9
hydro2d	4292	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

Table 8 Execution times (in seconds) for SPEC95fp programs

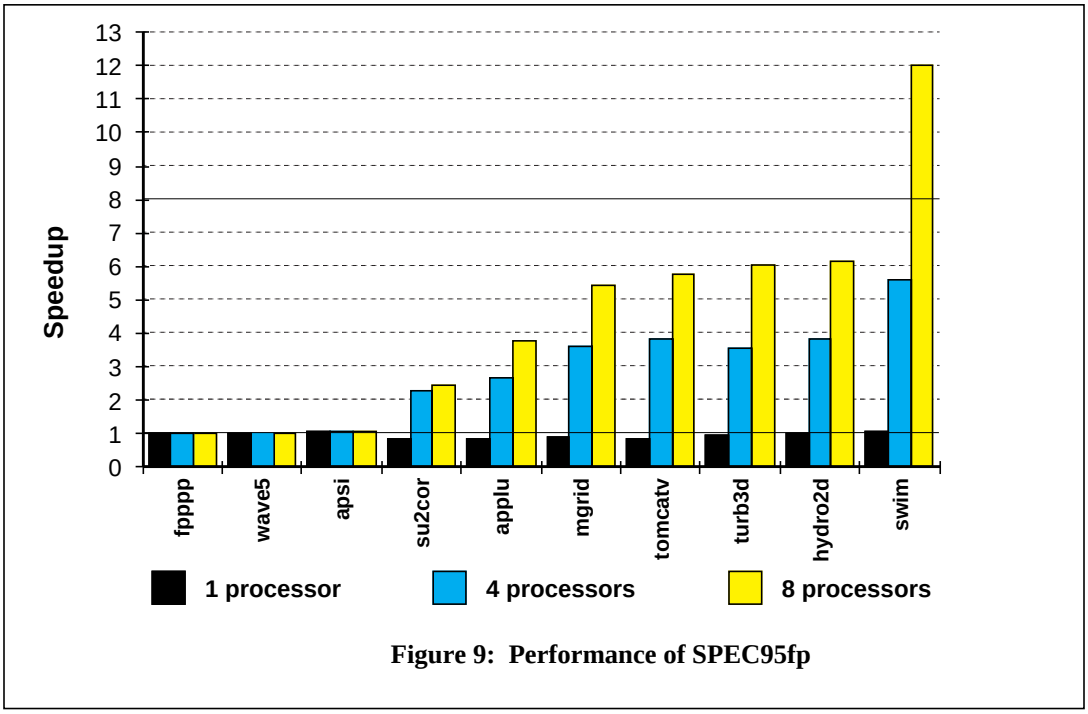

Figure 9: Performance of SPEC95fp

SUIF is able to speed up 7 out of the 10 SPEC95fp programs. The programs fpppp, wave5, and apsi suffer from poor parallel coverage. The program su2cor has a coverage of 90%, but the average granularity of parallelism is less than 200 ms. The application applu achieves the speedup shown as a result of parallelizing an outer loop using array privatization and blocking (see Figure 2). The rest of the programs achieved almost a perfect speedup on four processors; in fact, swim speeds up superlinearly primarily due to the improvement in cache performance as each processor is assigned less computation. 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 that of SPEC92fp. 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 4 processors, yielding a SPEC95fp ratio of 28.4. On 8 processors, there is a gain of 3.2 times, yielding a SPEC95fp ratio of 38.4.

5 Summary and Conclusions

This article examined the question of whether multiprocessors are viable as the next generation micro-architecture, from a software perspective. It demonstrated that automatic parallelization techniques are indeed mature enough to parallelize both Fortran and C programs that contain high degrees of instruction level parallelism. Compilers now have the capability to 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 make them relatively cheap and easy to build, leading to a faster clock and a shorter

time to market. For parallelizable applications, they enable speedup of a single application's execution; for sequential applications, they achieve a high machine utilization 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 micro-architecture.

Acknowledgments

We wish to 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. We thank Charles Grassl for helping us with the Cray Research C90 performance data. We are also grateful to Ed Bugnion for his help in performance tuning. We thank the rest of the SUIF group, Bob Wilson, David Heine, Chau-Wen Tseng, Patrick Sathyanathan, Alex Seibulescu, and Amy Lim for their contributions.

References

[1] J. M. Anderson and M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 112-125, Albuquerque, NM, June 1993.

[2] 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*, pp. 166-178, Santa Barbara, CA, July 1995.

[3] W. Blume et al., "Polaris: The Next Generation in Parallelizing Compilers," in *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing* (LCPC '94), Ithaca, NY, August 1994.

[4] 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*, December, 1995.

[5] B. A. Nayfeh and K. Olukotun, "Exploring the Design Space for a Shared-Cache Multiprocessor," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 166-175, Chicago, Ill, 1994.

[6] P. Tu and D. Padua, "Automatic Array Privatization," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing (LCPC '93)*, Portland, OR, August 1993.

[7] B. P. Wilson et al., "SUIF: A Parallelizing and Optimizing Research Compiler," ACM SIGPLAN Notices, 29(12), pp. 31-37, December 1994.

[8] B. P. Wilson and M. S. Lam. "Efficient Context-Sensitive Pointer Analysis for C Programs" in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 1-12, San Diego, CA, June 1995.

[9] M. E. Wolf and M. S. Lam. "A Data Locality Optimizing Algorithm," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30-44, Toronto, Canada, June 1991.

Appendix I

Table 10 Cray C90 Compiler Options

Programs	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`