On the Automatic Parallelization of the Perfect Benchmarks®*

Rudolf Eigenmann† Jay Hoeflinger
David Padua

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

Abstract

This paper presents the results of the *Cedar Hand-Parallelization Experiment*, conducted from 1989 through 1992 within the Center for Supercomputing Research and Development (CSRD) at the University of Illinois. In this experiment we manually transformed the Perfect Benchmarks® into parallel program versions. In doing so, we used techniques that may be automated in an optimizing compiler. We then ran these programs on the Cedar multiprocessor (built at CSRD during the 1980s) and measured the speed improvement due to each technique.

The results presented here extend the findings previously reported in [EHLP91]. The techniques credited most for the performance gains include array privatization, parallelization of reduction operations, and the substitution of generalized induction variables. All these techniques can be considered extensions of transformations that were available in vectorizers and commercial restructuring compilers of the late 1980s. We applied these transformations by hand to the given programs, in a mechanical manner, similar to that of a parallelizing compiler. Because of our success with these transformations, we believed that it would be possible to implement many of these techniques in a new parallelizing compiler. Such a compiler has been completed in meantime and we show preliminary results.

1 Introduction

Background and Motivation This paper presents the results of an extensive experiment, the **Cedar Hand-Parallelization Experiment**, conducted from 1989 through 1992 within the Center for Supercomputing Research and Development (CSRD) at the University of Illinois. This experiment formed the basis of our current work on the Polaris parallelizing compilation system. The primary objective of this experiment was to study application programs in order to identify parallelization techniques for optimizing compilers. We took a set a representative application programs, turned them into parallel codes and demonstrated that they can exploit parallel machines efficiently. Although we started from a compiler viewpoint, we broadened our project to consider additional questions, such as whether there is enough parallelism in real programs, whether there are engineering methods other than high-level algorithm replacements

*This work was supported by Army contract #DABT63-95-C-0097 and the U.S. Department of Energy under grant #DOE DE-FG02-85ER25001. This work is not necessarily representative of the positions or policies of the Army or the Government.

†Current affiliation: School of Electrical and Computer Engineering, Purdue University, Indiana.

that can transform these programs into efficient parallel codes, and whether there are tools and methodologies that can guide a programmer in this process. We have commented on these questions in several reports [EHLP91, Eig93, EM93].

According to our main interest, we will emphasize the implications of this project on the design of future compilers. We will discuss program transformations and their automatability in an optimizing compiler. Our discussions will be at the level of the application programmer who has found transformations that are successful in turning programs into parallel form. Some of these transformations are automatable in a straightforward manner. Others will require considerable work to implement the associated program analysis techniques. Some of the described transformations may never be implemented in a parallelizing compiler because they may be too complex or rarely useful, or the compiler may not be able to determine their applicability. Furthermore, we cannot make claims about the efficiency of the new techniques.

The results of our experiment and the early successes of our follow-on compiler implementation project[1] made it clear to us that application experiments such as these are extremely worthwhile efforts, complementary to compiler projects. These studies not only show us what may be implemented in compilers of the future, but they also provide quantitative evidence for urgent questions such as what programs can be implemented efficiently on parallel machines, what tools and methods programmers use to accomplish this, and what the resulting performance is on the given machines.

Overall Results In our experiments we have measured the speed improvements for the Perfect Benchmarks® programs that resulted from our program transformations. We have measured the performance on two machines: the Alliant FX/8 and the Cedar machine built at CSRD during the 1980s. The Alliant FX/8 had a shared-memory architecture with eight vector processors. The Cedar architecture consists of four Alliant FX/8 machines (each called a *cluster*) connected through an additional global shared memory. In this paper, we define "speedup" as the ratio of the best vector/concurrent execution time to the execution time of the serial, scalar program (i.e., using neither vector nor concurrent parallelism, but scalar-optimized). An excellent speedup would be about 16 for the FX/8 and about 50 for the Cedar machine. More details on these architectures can be found in [KDL+93].

Table 1 summarizes our measurements. The column "Automatically compiled" shows the performance gain achieved by the commercial parallelizing compilers KAP and VAST. The column "Manually improved" shows the results of our hand optimizations, which we will discuss in this paper.

Automatic parallelization had limited effectiveness with the technology used by KAP and VAST. VAST was the standard optimizing compiler available on the FX/8 machine (it did not generate code optimized for Cedar). The KAP results in the table were gathered from a version of KAP which we modified to generate code for the Cedar machine[EHJ+93].

Notice that the automatically-compiled speedup for Cedar, cited in the table, is almost always less than the automatically-compiled speedup for a single FX/8. This is the case even though the Cedar machine contained *four* FX/8 machines. The reason for this is that early

[1] The name of the follow-on compiler is Polaris. Because of the substantial time that passed between the first version of this report and its publication, early results of the new compiler have been presented already in several conferences.

program	serial execution time (seconds)	Performance improvement factor Automatically compiled FX/8 (VAST)	Cedar (KAP)	Manually improved FX/8	Cedar
ARC2D	2943	8.7	13.5	10.6	20.8
FLO52	906	9.0	5.5	14.6	15.3
BDNA	969	1.9	1.8	5.6	8.5
DYFESM	663	3.9	2.2	10.3	11.4
ADM	796	1.2	0.6	7.1	10.1
MDG	4134	1.0	1.0	7.3	20.6
MG3D	12164	1.5	0.9	13.3	48.8 [a]
OCEAN	2947	1.4	0.7	8.9	16.7
TRACK	136	1.0	0.4	4.0	5.2
TRFD	864	2.2	0.8	16.0	43.2
QCD	416	1.1	0.5	7.4	20.8 [b]
SPEC77	2375	2.4	2.4	10.2	15.7

[a] The manually improved version of MG3D includes the non-automatable elimination of file I/O.

[b] The manually improved version of QCD includes the difficult-to-automate parallelization of a hand-coded random number generator. Leaving the random number generator serial reduces the speedup to 2.0.

Table 1: Speedups versus serial for Perfect Benchmarks programs on Alliant FX/8 and Cedar

compiler technology typically could parallelize only very small loops and the overhead of starting such small loops on Cedar is larger than the overhead of starting small loops on the FX/8.

Table 2 lists the major techniques that we have applied to obtain the manually-improved results and the gains that can be attributed to individual transformations. The gains are given in terms of the execution time ratio of the program variant where the technique was not applied versus the best variant. More precisely, we computed these ratios as $T_{without}/T_{best}$, where T_{best} is the fully optimized program execution time and $T_{without}$ is the program execution time when the technique is not applied. The individual loop-by-loop timings are given in Tables 3 through 9. In the $T_{without}$ program variants, techniques that are used in commonly available restructurers are applied. Hence the numbers reflect potential performance improvements relative to the state of the art of compiler technology at the time of our experiments.

Related Work The primary contribution of this paper is the quantitative analysis of real program patterns and the discussion of their implications for compiler design. There exist very few similar publications. One related program analysis project was presented in [SH91], where the authors reach qualitatively similar conclusions that there is a need for advanced compiler techniques including symbolic analysis and the privatization of data structures. Other related projects have studied the effectiveness of existing parallelizing compilers or their techniques. A summary of these studies is given in [BENP93].

A second contribution of this paper is the discussion of new parallelizing compiler techniques. There exists a large body of publications on this subject. A survey can be found in [BENP93].

Several papers are directly related to the techniques discussed here. Early results of the presented work [EHLP91] have led to new efforts elsewhere, such as the work on techniques for handling general forms of induction variables [Wol92, HP93] and for analyzing privatizable arrays [Li92, MAL92]. There also has been significant new work in data dependence analysis for parallelizing compilers, including techniques for more exact subscript analysis [Pug92], more efficient analysis in practical situations [MHL91, GKT91], and enhancing tests with symbolic analysis capabilities [HP91].

In our work, we have focused on machines that provide a hardware-supported global address space to the programmer and the compiler. They represent an important and widely-available class of parallel machines. Discussions on complementing techniques for code generation on message passing machines can be found in [For93, GMS+95].

Cedar Fortran The parallel Fortran language that we used for expressing our explicitly parallel program variants is called CEDAR FORTRAN. In our descriptions, we will use the three different types of parallel loops of CEDAR FORTRAN: the XDOALL loop uses a self-scheduling scheme to distribute its loop iterations to all 32 processors of the Cedar machine. In contrast, the SDOALL self-schedules iterations between only four processors (one from each cluster). Usually, inside an SDOALL there is a nested CDOALL loop, which executes its iterations using the cluster's eight processors. CEDAR FORTRAN expresses array vector operations using *triplet notation* similar to Fortran90. By default, CEDAR FORTRAN data is placed in cluster memory. Data can be given a GLOBAL attribute, placing it in the global shared memory. Scalars and arrays declared inside a parallel loop define loop-private data which are given the scope of a single loop iteration.

Notation All programs referred to in this paper are from the Perfect Benchmarks. We will refer to an individual loop within the body of the paper by the notation "PROGRAM-routine/loop_label", or "routine/loop_label", while mentioning the name of the benchmark program involved.

Outline of the paper In Section 2, we will describe the new transformations and quantify their effectiveness on time-critical loops in our program suite. In Section 3, we will give an overview of the transformations applied to each program and note additional changes made to individual programs. We also will describe some of the difficulties that can be expected when implementing these techniques in a compiler. Finally, in Section 4, we will present early results of a new parallelizing compiler which incorporates the proposed techniques.

2 New transformation techniques: performance and relevant code patterns

This section describes, for each of the important transformations applied in our manual experiments with the Perfect Benchmark codes, the resulting performance difference and the program patterns for which these techniques were applied. We also will briefly discuss analysis techniques that may need to be developed to implement the techniques in a parallelizing compiler. The

Technique	ADM	ARC2D	BDNA	DYFESM	FLO52	MDG	MG3D	OCEAN	QCD	SPEC77	TRACK	TRFD
privatize arrays	9.6	1.2	1.4	2.2	1	21	18	3.8	8.2	6.8	6	13.3
parallelize complex reductions	[a]		3.3	2.1	1.1	21	15.2			3.4		[b]
substitute generalized induction variables								8.3				12.7
parallelize loops with non-affine array subscripts				3				11.5				13

Table 2: Importance of program transformations: Increase in execution time when individual techniques were disabled. (e.g., the program MDG slows down by a factor of 21 if arrays are not privatized)

[a] ADM contains reductions in significant loops. Most of them can be parallelized using existing synchronization techniques without excessive overhead.

[b] TRFD contains accumulation operations that would become important for parallelization if advanced induction variable substitution and array privatization were not available.

```
DO  i=1,n                          CDOALL  i=1,n
   DO j=1,m                        REAL work(1:m)
      work(j) = ...                   DO j=1,m
   ENDDO                                 work(j) = ...
   ...                       →         ENDDO
   DO j=1,m                           ...
      ... = work(j)                   DO j=1,m
   ENDDO                                 ... = work(j)
ENDDO                                  ENDDO
                                   ENDDO
```

Figure 1: The array privatization transformation

development of these analysis techniques is beyond the scope of this paper, however. We hope that our results will serve as a solid basis for similar future development projects.

2.1 Array privatization

Data that are used temporarily within a loop iteration, can be privatized to the loop so that each processor participating in the loop execution has separate storage for the data. This resolves many data dependences that would arise if all loop iterations used the same temporary storage for their operations. For example, the loop in Figure 1 uses a temporary array `work`. In the parallel execution of the loop, the array is declared private to the loop. This allocates for each iteration a separate instance of `work`, hence eliminating the conflicts between different iterations trying to access the same temporary storage concurrently.

We have found this temporary usage pattern to be very common in the Perfect codes. This is not too surprising, for it is natural to use temporary data structures in all programs. The

more our optimization efforts sought to parallelize outermost loops in a program, the more likely it was that this temporary usage pattern occurred within a single loop iteration.

Array privatization, in combination with the other techniques described in this paper, enabled many of the most time-consuming loops in the Perfect codes to be run concurrently. Table 3 shows the performance degradation that we measured on the Cedar multiprocessor when not applying the technique. Also listed in Table 3 are all loops in our program suite where array privatization made a significant difference. The loop execution time with all parallelization techniques applied (T_{best}) and without array privatization applied ($T_{without}$) are shown. Column 4 shows the performance difference that resulted from applying array privatization in this loop, measured as $T_{without}/T_{best}$. The last column shows the overall program performance difference made by applying array privatization to this loop. These factors are very high where inner loops cannot be parallelized, and, thus, disabling array privatization serializes the loop. Even where inner loops could be parallelized, the parallelization of outer loops can prove very effective because the overall startup cost is lower for outer loops. Privatized data are allocated in local memory. Hence, the transformation also has the important effect of exploiting local memories. In fact, this was one of the most important methods to exploit the Cedar memory hierarchy.

Array privatization is a natural extension of the scalar privatization technique. Current compilers are able to recognize temporarily used scalar variables and either expand them into arrays indexed by the loop iteration number, or privatize them by declaring them local to a loop. In fact, in previous evaluation experiments, we have found this technique to be the most important one applied by current parallelizing compilers. However, although array privatization techniques have been known for several years and some parallelizing compilers are able to apply the transformation, we have not seen any available compiler solve the important program patterns we have encountered.

Analysis and Program Patterns This section describes the information necessary to determine whether an array is privatizable. It also describes the code patterns where the techniques made a significant difference. Furthermore, it will explain the program analysis that was necessary in our experiments to determine whether array privatization could be applied.

Data items can be privatized to a loop if they are defined in each loop iteration before they are used. To find this information, one needs to analyze definitions and uses of arrays or array sections in each candidate loop. Furthermore, one must make sure that the values of the privatizable data items are not used after the loop, or, if they are used, provisions must be made to transfer the values of the privatized data items to the original item outside the loop. However, in our experiments, we rarely needed such *last-value assignments*. In many cases it was easy to prove that the private arrays were not *live* at the end of the loop. It was more difficult to prove when the arrays being used as temporaries in a loop were subroutine parameters or were in common blocks. Such situations often arise when the programmer is explicitly managing storage by using large arrays declared in the main program. Examples of this are in ARC2D (subroutines `filerx` and `filery`) and TRFD (loops olda/100 and olda/300).

The identification of privatizable arrays is quite simple for arrays that are declared locally in a subroutine that is called within the loop being analyzed. By the Fortran77 language definition, such variables are undefined after the subroutine exits, thus guaranteeing that they

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
ARC2D-filerx/15	7.3	22.0	3.0	1.1
ARC2D-filery/39	3.4	12.0	3.5	1.06
ADM-dudtz/40	3.8	92.5	24	2.1
ADM-dvdtz/40	3.5	76.5	21	1.9
ADM-dtdtz/40	3.8	78.8	21	1.9
ADM-dcdtz/40	2.6	51.7	20	1.6
ADM-dkzmh/30	2.5	37.6	15	1.4
ADM-dkzmh/60	3.8	86.8	23	2.0
ADM-run/20	4.4	72.2	16.5	1.8
ADM-run/30	4.4	72.0	16.4	1.8
ADM-run/40	4.2	72.1	17	1.8
ADM-run/50	3.4	50.8	15	1.6
ADM-run/60	4.4	72.0	16.4	1.8
ADM-run/100	3.2	50.0	15.6	1.5
ADM-wcont/40	2.7	36.2	13.4	1.4
ADM-smooth/10	1.4	18.5	13.2	1.2
BDNA-actfor/240	19.0	62.0	3.3	1.4
DYFESM-mxmult/10	19.0	60.0	3.2	1.7
DYFESM-solvh/20	11.5	26.5	2.3	1.3
MDG-interf/1000	163.0	3792.0	23.2	19
MDG-poteng/2000	13.4	352.0	26.3	2.7
MDG-intraf/1000	1.9	11.4	6.0	1.05
MG3D-migrat/200	264.0	5226.4	19.8	19.7
OCEAN-acac/30	3.3	92.5	28	1.5
OCEAN-ocean/60	0.3	0.05	5.6	0.9
OCEAN-ocean/270	1.3	16.2	12.5	1.0
OCEAN-ocean/340	2.9	30.7	10.6	1.1
OCEAN-ocean/360	2.7	25.6	9.5	1.1
OCEAN-ocean/400	2.3	20.9	9.1	1.1
OCEAN-ocean/420	2.3	25.6	11.1	1.1
OCEAN-ocean/440	2.6	24.5	9.5	1.1
OCEAN-ocean/460	7.7	103.7	13.5	1.5
OCEAN-ocean/480	4.1	91.4	22.3	1.4
OCEAN-ocean/500	2.2	48.0	21.8	1.2
OCEAN-scsc/40	2.5	48.0	19.2	1.2
QCD-measur/3	1.8	2.9	1.6	1.0
QCD-qqqmea/1	3.7	108.6	29.4	6.2
QCD-rotmea/2	3.7	48.4	13.1	3.2
SPEC77-gloop/1000	58.7	743.7	12.7	5.5
SPEC77-gwater/1000	13.5	248.0	18.4	2.6
TRACK-extend/400	4.0	48.9	12.2	2.7
TRACK-fptrack/300	3.0	15.5	5.2	1.4
TRACK-nlfilt/300	3.5	76.7	22	3.8
TRFD-olda/100	8.0	(174)	21.8	9.3
TRFD-olda/300	5.4	(85)	15.7	5

Table 3: Performance impact of the array privatization technique in the time-critical loops. Best loop variants are compared with those where the transformation is not applied. (The numbers in parentheses give the timings without reduction parallelization.)

will not carry values across loop iterations. (There are two exceptions to this: variable values made available to subsequent calls of the same subroutine through `save` statements or through the static allocation option offered by many compilers). DYFESM is an example of a program that contains many subroutine-local arrays that can be privatized.

In general, the identification of privatizable arrays is complex. Often, the privatizable data structure is an array range, which is defined and used by a loop body in simple statements (such as assignments), compound statements (such as inner loops), or both. The ranges are determined by the value of scalar variables whose analysis involves the interprocedural search for their values, relations between variables, and conditions under which such relations hold. The area being written and read within an array is often accessed over a series of sub-ranges. These sub-ranges must be pieced together in the analysis, and it must be shown that the range being read is fully within the range that was written. The loop ocean/420 in OCEAN is an example of this.

In some cases, elements that are adjacent to privatizable array sections are read-only and can be shared by parallel loop iterations. The situation may allow the enlargement of privatized arrays by initialized read-only boundary elements in order to avoid conditional operations in the loop body. Code examples of this were found in ADM (subroutines `dtptz`, `dudtz`, and `dvdtz`).

To analyze array definitions and uses, we had to search interprocedurally and propagate constants, symbolic values, relations between variables, and sometimes information about the values of subscript array elements. In ARC2D-filerx/15, the range of an array (`work`) is defined under a condition (variable `PERIDC`) whose relation to other variables can be symbolically analyzed in the program initialization routine. Also important is a subscript array whose i^{th} element is initialized to *(i+1) mod n*, which can be recognized as a permutation vector of length n.

Another interesting pattern is found in MDG, loop interf/1000. An array (`RL`) is defined and then used under control of two separate IF-statements with different IF-conditions. It can be seen from the program text that the *define*-condition is always true when the *use*-condition is true. This relation can be proven symbolically. The candidate parallel loop calls a subroutine (`cshift`) that defines this relation. A similar situation exists in QCD (measur/3).

To privatize the arrays in loop actfor/240 in BDNA, one has to recognize subscripted subscript patterns, which is very difficult in general. However, in the given situation, all necessary information can be derived from the program text [TP93, BE94b].

2.2 Parallel Reductions

Statements of the type `sum = sum + a(i)` (where `i` is the loop index) form a recurrence pattern that usually must be executed serially. However, because the sum operation is mathematically commutative and associative, a parallel execution is possible by accumulating partial sums on each processor, and then summing the partial results, as shown in Figure 2. The partial results may be summed after the loop or added inside the loop in a critical section. Note that this transformation may change the result, because reordering the sum operations may lead to round-off errors that are different from those in the original program. The programs in the Perfect Benchmarks suite have not been found to be sensitive to such reorderings.

Reordering certain arithmetic operations in order to increase parallelism is a technique

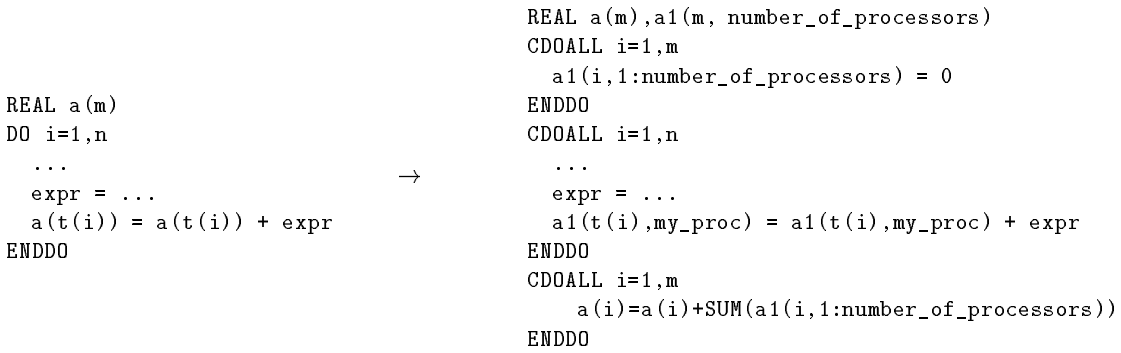

Figure 2: Expanded Parallel Reduction Transformation

known as *tree height reduction* [KBC+74]. Simple reduction operations are recognized by parallelizing compilers and transformed into the appropriate vector or vector-concurrent constructs. We have measured this capability and found it to be one of the most effective ones. Current commercial compilers apply the technique most often where a sum operation is performed on a scalar variable that is not referenced elsewhere in the loop.

However, in our experiments, we have found important loops that contain multiple sum statements adding to the same variable. We have also found loops where this variable is an array whose index may vary and be unknown at compile time. The compilers we evaluated could not parallelize such loops, which was one important reason for their limited performance on the Perfect Benchmarks.[2]

Reduction operations are a subclass of recurrence computations for which there are efficient parallel implementations. General recurrence patterns are of the form $x(i) = a_0(i) * x(i) + a_1(i) * x(i-1) + \ldots + a_n(i) * x(i-n)$ (where i is the loop index). Parallel solvers for general recurrences are well known, and today's parallelizing compilers are capable of replacing these patterns with calls to the appropriate solver routines. However, in our experiments, we have not found any performance gains from such transformations. One reason for this is that the available recurrence solver libraries were not tailored to the specific recurrence patterns that occurred in our test suite, so that sometimes the input arrays had to be rearranged before calling the solver. Even more importantly, the iteration counts for the recurrence loops found in our program suite were too small to amortize the overhead introduced by a parallel recurrence algorithm. Because we have usually found recurrences to be inside parallel loops, this was not a serious problem, and we have not invested much effort to improve these situations. Parallel recurrence solvers may become more effective in other programs by tailoring them to the specific program patterns.

Analysis and Program Patterns Parallelizing compilers usually recognize reduction operations by searching for a pattern defined by the statements and their data dependences. Traditionally, candidate reduction operations must: be confined to a single statement; have a

[2]The VAST parallelizer was able to parallelize loops with multiple statements of the form `TOT(i) = TOT(i) + ...` by placing `await` and `advance` synchronization around them and pulling as much computation out of the critical section as possible. This works well where the body of the parallel loop is sufficiently large.

self-, flow-, output-, and anti- dependence; and accumulate into a scalar variable using a commutative/associative operation. Also, the statement in which they occur must be distributable from the adjacent loop body.

To handle the more complex patterns that we have found in our program suite, it is necessary to deal with multiple accumulation statements and accumulations into array elements within inner loops. Most of the information necessary to do this transformation can be gathered locally in the candidate loop, although in some programs the information needed must be gathered interprocedurally, as in DYFESM-assemr/40. Additional information, such as the length of the accumulated array and the size of the section that may need to be synchronized, is needed to determine the best method for implementing parallel reductions. We have observed that the number of iterations of the enclosing loop determines whether transforming reductions to parallel form increases or decreases the speed of the program. Thus, the number of iterations is crucial information. Also, the techniques for mapping parallel loops to the machine (Section 2.4) are important in finding the proper transformation in each case.

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
MDG-interf/1000	163	3792.0	23.2	19
MDG-poteng/2000	13.4	352.0	26.3	2.6
DYFESM-mxmult/10	19.0	60.0	3.2	1.7
DYFESM-formr0/20	7.0	20.0	2.8	1.2
BDNA-actfor/240	19.0	62.0	3.3	1.4
BDNA-actfor/500	21.0	253.0	12.0	3
FLO52-euler/70	0.5	5.7	11.4	1.1
MG3D-migrat/200	264.0	5226	19.8	19.7
SPEC77-gloop/1000	58.7	743.7	12.7	5.5
SPEC77-gwater/1000	13.5	248.0	18.4	2.6

Table 4: Performance impact of the *parallel reduction* transformation.

Parallel reductions can be implemented in several different ways.

1. *Synchronization in place.* Each sum operation can be made indivisible by protecting it with a lock/unlock pair. If efficient, hardware-supported synchronization functions are available, or if few synchronizations are necessary, then this is a feasible solution. This variant requires the fewest code changes, and in contrast to methods 2 and 3, it does not require the allocation of a temporary array nor its initialization and final sum. (This transformation may require complex interprocedural analysis and code modifications.)

2. *Privatized parallel reductions.* This method builds partial sums in loop-private variables. These variables are then used to update the original variable in a critical region within the loop or in the postamble[2]. Using private partial sum variables improves locality, but still requires a synchronized section.

[2]The postamble is a section of code at the loop end that is executed once by each participating processor.

3. *Expanded parallel reductions.* Using this method, the partial sum variables are expanded by a dimension, which has as many elements as the number of processors, and given a global scope. Each processor accumulates into a slice of this array using its processor number as the slice index. The partial sums can be added to the original variable after the parallel loop. This variant may incur more overhead due to the accumulation into global memory elements, but it yields a completely parallel loop. If the accumulators are array elements, the sum operation to be performed after the end of the loop can be done in a vector-concurrent loop, i.e. a concurrent loop over all array elements with a vector sum operation inside. This translation is illustrated in Figure 2.

With architectures using a page (or cache line) migrating mechanism, care must be taken avoid accessing the same page with multiple processors at the same time, as in method three above. Method two above limits the access of the global array to a single processor at one time, which reduces coherence traffic.

Table 4 presents the effect of this transformation in the same terms as for privatization in Table 3. We used in-place synchronization in DYFESM (mxmult/10) and SPEC77 (gwater/1000 and gloop/1000). Both loops in SPEC77 call specific routines in order to accumulate values. Subroutine `assemr` in DYFESM, which is called inside the parallel loop, contains accumulation statements that update part of the array `MX`.

Privatized parallel reductions were applied in BDNA and MG3D. The pattern in loop migrat/200 of program MG3D is a single statement accumulation. Both loops BDNA-actfor/240 and 350 (the inner loop of nest 500) have simple accumulation patterns, while loop 500 has more complex patterns (using arrays `FAX,FAY,FAZ`).

We have applied expanded parallel reductions in ADM, which has a number of loops (dudtz/40,dvdtz/40,dtdtz/40,dcdtz/40,wcont/40, and hyd/30) with simple accumulation patterns. Alternatively, the accumulations can be synchronized (using current compiler capabilities), which we expect to perform close to the level we achieved with our transformations.

Of further interest are the reduction patterns in MDG and TRFD. In loop MDG-interf/1000, a scalar variable (`VIR`) and three arrays (`FX,FY,FZ`) are used for accumulating values. The sum statements are in an inner loop and operate on different elements of the array for each iteration of the parallel outer loop. In our experiments, a combination of the expanded and privatized parallel reduction scheme was applied. The parallel loop was stripmined[3] into a SDOALL/CDOALL pair and we then applied a combination of privatized and expanded parallel reductions. A similar pattern occurs in subroutine `poteng` where scalar accumulations are parallelized using privatized parallel reductions. In TRFD-olda/100, elements of two arrays (`XRSIQ` and `XIJ`) are being used to accumulate sums. Since this accumulation pattern takes place inside a parallel loop, it did not need to be parallelized. TRFD-olda/300 showed the same pattern for arrays `XIJKS` and `XKL`. In an earlier version of TRFD, where the outer loop was not yet parallelized, parallel reduction transformations proved effective.

2.3 Generalized induction variables (GIV)

In Fortran `DO` loops, array subscripts often use the values of induction variables [ASU86], which are updated in each iteration in the form V = V *op* K, where the value K is loop-invariant.

[3]Stripmining splits a loop into two nested loops.

Such a recursive assignment causes cross-iteration data dependences. If a compiler can solve such a recurrence and express the value of the induction variable in terms of the loop indices as, for example, $V = f(I, J)$, where I and J are loop indices, then uses of V in the loop can be replaced by the expression $f(I, J)$. This eliminates the dependence in the calculation of the induction variable, as well as dependences caused by using V in array subscripts. There are well-known compiler techniques for recognizing and replacing an induction variable whose values form an arithmetic progression. These techniques typically deal with induction variables assigned in the form of V = V + K.

In our experiment with the Perfect Benchmarks, we found induction variables whose values do not constitute arithmetic progressions. Here, we call them *generalized induction variables*, or GIVs. We found two types of GIVs. The first type is updated using multiplication instead of addition, thus forming a geometric progression. The second type is updated using addition, but does not form an arithmetic progression in all points of the loop because the loops are *triangular* (i.e., the inner loop limit depends on the value of an outer loop index).

We have described GIVs in a preliminary report([EHLP91]), which has led to the development of several recognition techniques, as mentioned in Section 1. All these techniques introduce non-linear subscript expressions, which cannot be understood by previous data-dependence analysis techniques. In [BE94a], we have described a new analysis technique that can handle such nonlinear subscripts.

The effect of the Generalized Induction Variable transformation is shown in Table 5. The table shows the same type of information as Tables 3 and 4. Without correct handling of the GIVs, all listed loops can be parallelized only on the inner-most loops. We have measured this inner parallelization to perform worse than serial execution. Because of this, Column $T_{without}$ shows the serial timings. As shown, not applying the parallel reduction techniques results in a 15 to 20-fold slowdown of the loops. Because the loops are time-consuming parts of their respective programs, the corresponding program slowdown would be between the factors of 5 and 10.

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
OCEAN-ftrvmt/109	89	1377	15.5	8.3
TRFD-olda/100	8.0	174.2	21.8	9.3
TRFD-olda/300	5.4	85.3	15.8	5.0

Table 5: Performance impact of the Generalized Induction technique.

Analysis and Program Patterns In order to substitute induction variables, one must first determine the value of the induction variable prior to the loop, find all the induction sites, and determine the loop bounds of the loops enclosing the induction sites. From this information, we can then compute the value of the induction variable at each reference in the loop body.

The first step is the same for all types of loops. The second step is more difficult in triangular loops, although it can be considered an extension of the rectangular loop case. For example, assume a doubly-nested loop with indices I and J, loop bounds M and N, and an induction variable that increments in steps of 1. In each iteration of the outer loop, the induction variable

increases by the iteration count of the inner loop. In the rectangular case, this is a constant N. In a triangular case, where the inner loop goes from 1 to I, it is a variable number I. At the end of the i-th iteration of the outer loop, this amounts to i*N in the rectangular case and $\sum_{i'=1}^{i} i' = i * (i+1)/2$ for a triangular nest. In iteration j of the inner loop, the expression is (i-1)*N+j for the rectangular case and (i-1)*i/2+j for the triangular case.

This is, of course, a simplified computation that assumes an induction variable starting at 0 and incrementing by 1. However, relaxing these constraints is straightforward. Also, note that the expression for the triangular induction variable is not linear.

In TRFD, the loops olda/100 and olda/300 contain additive GIVs, which are incremented inside triangular loops. An additional complication is found in olda/300, where the GIV pattern in the first iteration of the outermost loop differs from all the other iterations. We transformed this case by peeling the first iteration.

In the program OCEAN, loop ftrvmt/109, we found a multiplicative GIV. Further complicating the analysis of this GIV was the use of a complex control flow within the loop. The loop has the following form:

```
         DO 109 jl=1,i2k
             IF      (jl .EQ. 1  ) THEN
S1:             exj=(1.,0.)
                ...
             ELSE IF (jl .EQ. jli) THEN
S2:             exj=CMPLX(0.,sgn1)
                ...
             ELSE
S3:             exj=exj*exk
                ...
             ENDIF
     109 CONTINUE
```

Notice that the two `IF` statements use the value of the loop index in their decision of where to branch. This is crucial because it allows the compiler to determine the order in which the various branches occur. Analysis of the flow can ascertain that statement S1 executes first; followed by statement S3 which executes `jli` -1 times; followed by statement S2; followed by statement S3 which executes another `i2k` $-$ `jli` times. For this analysis to work we must verify that `1`$\leq$ `jli`$\leq$ `i2k`.

With this analysis, it is possible to determine closed forms for `EXJ` for all values of `JL`:

```
          DO 109 jl=1,i2k
              IF      (jl .EQ. 1)   THEN
                 exj=(1.,0.)
                 ...
              ELSE IF (jl .LT. jli) THEN
                 exj = (exk**(jl-1))
                 ...
              ELSE
                 exj = CMPLX(0.,sgn1)*(exk**(jl-jli))
                 ...
              ENDIF
      109 CONTINUE
```

2.4 Techniques that map parallel loops to the machine

The restructuring compiler that we used as a starting point for our hand-optimized codes (KAP/Cedar [EHJ+93]) often was able to discover parallelism, but then mapped it to the machine poorly. In this section, we will show various methods to improve this. Specifically, we will discuss issues of stripmining, loop coalescing, outer loop parallelization, loop fusion, and data localization.

2.4.1 Balanced Stripmining

The stripmining transformation splits a single loop into two nested loops. In Cedar programs, this is usually applied for exploiting multiple levels of parallelism. The naive stripmining method we implemented in KAP/Cedar turns each parallel loop into an SDOALL/CDOALL/vector nest in the following way: an innermost loop executes 32-element vector instructions, an intermediate CDOALL loop executes 8 iterations, and an outermost SDOALL loop iterates over these groups of 8×32. This method produces poor speedup when the number of iterations is small.

The VAST parallelizer, for the Alliant FX/8 machine, splits the iteration space onto processors in a cyclic way. The iterations are divided across processors first. The portions assigned to each processor are then executed by the vector instructions. In this stripmining method, parallel processors tend to access adjacent data elements, which increases spatial locality in the shared cache. This is effective for doing computation within a single cluster.

Our multi-version, balanced stripmining method is similar to the stripmining applied by KAP. However, it first determines what resources are needed. If it can be determined at compile time that the number of iterations is extremely small, a single processor may be allocated to the loop, doing either vector or scalar instructions. A medium number of iterations may warrant the use of a whole Cedar cluster, and loops with large iteration counts may be mapped to the whole Cedar machine.

If the number of iterations cannot be determined from the program, or if it is known that the number of iterations will vary widely over the course of the execution, a multi-version loop can be constructed. The program selects between a single-processor, a single-cluster, and a four-cluster version, depending on the number of iterations. The four-cluster version stripmines the iteration space onto the processors in contiguous blocks to produce vector instructions with a stride of one. Stride-one vector references are beneficial in the Cedar architecture because of the way the prefetch unit operates [GJT+91].

When the original loops are doubly-nested they are coalesced into a single loop first, so that the combined iteration space is available to divide among the processors. The stripmining transformation itself divides the iterations among the clusters first, and then among the processors of a cluster. Finally, it assigns the remaining elements to the vector instructions. This ensures the minimum number of next-iteration operations across clusters and between processors on a single cluster.

Analysis and Program Patterns In order to decide what translation to use for a given loop nest, we need to analyze the size of the loop body and the number of iterations. The size of the loop body will help us determine whether the parallel loop overhead can be amortized by the performance gains of the parallel loop. The number of iterations determines how many next-iteration operations will be done during the loop. We have found that, where it is crucial,

this information often can be derived from the program text. However, in many cases, this information must be gathered from subroutines other than the one being transformed. We will discuss this further in Section 2.5.

Many interesting program patterns can be found in FLO52. Loop eflux/30 was originally a perfectly-nested two-level loop in the serial version of the program. We did the Balanced Stripmining transformation to each loop separately. The outer loop became an SDOALL/serial nest, and the inner loop became a CDOALL/vector nest. In addition, we moved the serial loop inside the CDOALL loop in order to push out the parallel loops as far as possible. The result was a SDOALL/CDOALL/serial/vector nest. These transformations improved the run-time of the loop from 5.7 to 3.9 seconds. Table 6 shows this improvement together with that of other loops of the same program. Loops eflux/20, eflux/30, and dflux/30 were originally singly nested (1D). Loops eflux/10 and dflux 38 were doubly-nested and we have coalesced them before stripmining (2D). In bcwall/30, we applied the multiple-version transformation described above (MV).

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
FLO52-eflux/10	3.9	5.9	1.5 (2D)	1.03
FLO52-eflux/20	1.1	3.0	2.7 (1D)	1.03
FLO52-eflux/30	3.9	5.7	1.5 (1D)	1.04
FLO52-dflux/30	3.2	5.5	1.7 (1D)	1.03
FLO52-dflux/38	0.4	0.5	1.25 (2D)	1.00
FLO52-bcwall/30	1.7	2.8	1.6 (MV)	1.02

Table 6: Performance impact of the Balanced Stripmining technique.

2.4.2 Loop Coalescing

When there are multiply-nested parallel loops whose iteration count is either unknown at compile time or not constant, it may be beneficial to coalesce the loops into one single loop. A simple example of this is the nest mnlbyx/50 in DYFESM, where the bounds of two out of three loops are unknown.

We have found a more complex loop pattern that warrants coalescing in ftrvmt/109 of OCEAN. In every invocation of the doubly nested loop, the loop body executes 64 times. However, the loop is triangular and the number of iterations in the two loops vary, as shown in Table 7. When parallelizing a single loop of this nest, the parallel performance gain is limited due to the small iteration count in half of all loop executions. In this situation, it is advantageous to coalesce the two levels and make a single parallel loop out of them. For Cedar, this loop may become an XDOALL. This transformation has been described in [Hoe92a].

Once these loops were coalesced, the number of iterations became large enough that every invocation of the loop could exploit the Cedar resources. As a result, the speed of the loop doubled. The loop nest accounts for approximately 50% of the serial program execution time.

	Numbers of iterations						
Inner loop	64	32	16	8	4	2	1
Outer loop	1	2	4	8	16	32	64

Table 7: Number of iterations in OCEAN-ftrvmt/109 loop nest.

2.4.3 Outer loop parallelization and loop fusion

Within a Cedar cluster, fast communication hardware is available to start, terminate, and synchronize parallel activities. However, global memory is the only medium for inter-cluster communication. Because of this, it is important to select loops with either a high number of iterations or a large loop body for parallel execution across the Cedar clusters.

Large loop iteration counts often can be obtained with large input data sets. The Cedar architecture can work efficiently given large data sets, as we have shown in our previous work [EHJ+93]: linear algebra routines working on matrices of size 1000 by 1000 can exploit the 32 Cedar processors well. Although it seems commonly accepted for highly parallel systems to refer to large data sets, ordinary programs may have small iteration counts and thus, the programmer or the compiler may have to find transformations to increase the grain size of their parallel algorithms.

An example of such a transformation is illustrated in Figure 3. The major subroutine of the program FLO52 consists of two loops, each having a sequence of small inner loops. The original version of our compiler parallelized the inner loops only, which is represented by variant A. Variant B shows a program where the two outer loops were parallelized. In variant C, these two loops were fused; thus, the whole subroutine became one parallel loop. The inner loops were also vectorized or stripmined for vector-concurrent execution, when necessary.

The resulting performance gain was 50% on the Alliant FX/8 architecture, as compared to 100% on Cedar. This is due to the difference in startup overhead between the CDOALL and SDOALL loops, and the increased parallelism encompassed by outer loops. The major gain is due to the first step of parallelizing the outer loop instead of the inner one. A small additional gain is due to fusing the two outer parallel loops into a single loop. On the FX/8 machine, where startup overheads are small, this additional gain is not noticeable. The example in Figure 3 shows that compiling a structure of multiple small SDOALL loops into a single SDOALL can result in a significant code improvement on Cedar.

The compiler used in our studies (KAP/Cedar [EHJ+93]) often was able to find large concurrent loops or to interchange loops to an outer position. It failed in other cases when too many potential data dependences were detected or when the outer loop was in a calling subroutine. Our manual analysis has shown that outer loops, in fact, can be parallelized in most programs. Although this is not a transformation per se (it is no different from finding parallelism using all the techniques described so far), it is mentioned here for its importance as a program transformation goal.

2.4.4 Data localization

Data localization techniques are important for reducing latencies and contention of data accesses to the global, shared memory. One of the most important transformations for data localization

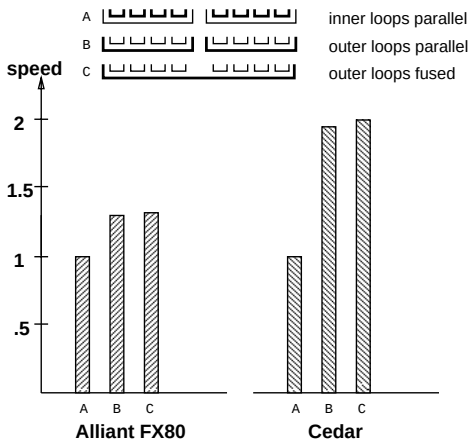

Figure 3: Combining multiple parallel loops into a single one.

is the array privatization technique. Data declared loop-private are placed in cluster memory and, thus, contribute to the exploitation of local memories.

In addition to the array privatization technique, data can be localized by keeping copies of global data in private (cluster) memory. This can be useful within single loops if there are many accesses to these data. If data are produced by one loop and consumed by another, then it may be possible to partition and distribute them to cluster memories across the sequence of parallel loops. We refer to these two techniques as *intra-loop localization* and *inter-loop data partitioning and distribution*, respectively.

Intra-loop localization. If global data are read-only, they can be localized easily by copying them to local memory at the beginning of the loop. This is particularly important if the data are accessed in scalar mode, repeatedly within a loop iteration (or a sequence of iterations assigned to the same cluster), or in vector operations with a large stride. Both these types of accesses suffer from long latencies to global memory in the Cedar architecture. The only Cedar mechanism for reducing such latencies is the vector prefetch, whose applicability is limited to long vector operations with a small stride.

Data that are read and written can also be localized to a loop if there are a number of references to each data element. Multiple references can amortize the cost of copying the data to and from the global memory at the beginning and end of the loop, respectively.

We have applied such transformations in a number of loops, as shown in Table 8. All but the first line show the effect of read-only data localization[4]. The first line measures the effect of the localization of a read-write array with data copy instructions at the beginning as well as the end of the loop.

Inter-loop data partitioning and distribution. Data can be privatized to a loop when its life is confined to a loop iteration. When the lifetime spans several loops, one can attempt to place data partitions onto each cluster memory and to assign corresponding subsets of the

[4]The timing numbers are extrapolated from an experiment run on one cluster only.

PROGRAM-subroutine/loop	access	total loop execution time in seconds T_{best}	$T_{without}$	$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
FLO52-psmoo/40&80	r/w	9.9	19.8	2.0	1.17
FLO52-step/20	r/o	1.7	2.4	1.4	1.01
ARC2D-xpenta/11	r/o	5.8	6.7	1.15	1.01
MDG-interf/1000	r/o	163	187	1.15	1.12
TRFD-olda/100 & 300	r/o	13.4	15.91	1.18	1.13

Table 8: Performance impact of data localization. "r/w" is read/write access and "r/o" is read-only access.

loop iteration spaces to the cluster processors. This works without further communication for data that are read-only or that are read by the same cluster on which they were written.

In our experiments we searched for data partitioning and distribution techniques in particular, since they would have allowed us to take advantage of the Cedar's distributed cluster memory architecture, one of the distinguishing features of this machine. However, we have not been able to identify such transformations as important performance enablers in our program suite. Nevertheless, we believe that data partitioning and distribution techniques will become important for overcoming the increased memory access latencies that are intrinsic to future highly-parallel systems. This is true in particular where access latency ratios of global to local accesses may be substantially larger than in the Cedar architecture.

2.5 Analysis Techniques

In the above discussions of program transformations, we have briefly described program analysis techniques that may become necessary. Further analysis techniques may become important on a more global scope since they benefit many transformations or are basic enablers for detecting parallel loops. Such techniques include interprocedural analysis, data dependence analysis, and the analysis of programs at run-time.

2.5.1 Interprocedural symbolic analysis

One of the most important analysis techniques needed when implementing the described transformations is the investigation of variable values and the propagation of this knowledge to the loops to be restructured. In our manual experiments, we often have used the knowledge that variables stay within certain bounds, that subscript arrays are initialized with values that allow arrays to be privatized and loops to be run in parallel, or that the program takes control paths that guarantee the safe and efficient application of the transformation. We have pointed out the need for such analysis techniques in previous sections. An important observation was that there were only a few cases in which crucial variable values were read from input files and could not be derived from the program text. Often, variables were given values in an initialization routine.

Therefore, in many programs of our suite, sufficient information exists in the program text to do a thorough job of compile-time analysis. We believe that, although the techniques may be complex, it is possible to develop the necessary symbolic analysis technology. These patterns

and possible symbolic analysis techniques are described in more detail in [BE94b].

In most of the Perfect Benchmarks programs, we have found this analysis to be necessary across subroutine calls. We need interprocedural analysis for two reasons. First, a compiler without interprocedural analysis capabilities has to make conservative assumptions about control flow and data dependences in calling and called subroutines, which often prevents it from recognizing parallel loops. Second, there may be variables whose values determine the existence of dependences, and these values may be passed as arguments from other subroutines where they are defined. Again, this prevents parallelization. The compiler we used in our experiments relies on *inlining* [Hus82], which replaces call statements with the text of the called subroutines. This has worked well in a few programs, but we have found that other programs would need more advanced interprocedural analysis capabilities than we had available. For example, the analysis would have to work across subroutine boundaries where arrays passed as parameters are declared with different shapes in the calling and called routines.

All of the transformations and analyses that we performed in our experiments were done *in the presence of* interprocedural analysis. That is, if we needed to know the value of a constant not defined locally, we inspected other subroutines. If we needed to know how data were used and defined in a call tree initiated from a particular loop, we analyzed the called subroutine.

2.5.2 Data-dependence analysis

We have found that many array reference patterns in the important loop nests are relatively simple combinations of the loop variables of the enclosing loops. Nevertheless, these patterns often cannot be investigated by current data-dependence tests, mainly because the coefficients of the index variables are symbolic expressions. Advanced symbolic analysis techniques are needed to propagate relations between variables from the definitions of these variables to the loops that are considered for parallelization. Furthermore, data-dependence tests need to be extended to allow symbolic terms in their mathematical expressions instead of just numeric constants.

One challenge for data dependence tests was encountered in TRFD. The terms generated by the induction variable of the triangular loops described in Section 2.3 are quadratic and, thus, cannot be understood by tests for linear subscripts. For the reader of the original program, it is obvious that the arrays indexed by the induction variable are dependence-free because the induction variable assumes a steadily increasing sequence of values. This knowledge is lost when substituting the induction variable. An approach could be to flag this situation in the induction variable substitution pass and enhance the dependence test to understand this flag. Alternatively, a non-linear test could be devised [BE94a]. Of further difficulty is the fact that the subscript contains symbolic values whose relation to other terms determines whether the loop is independent. Symbolic analysis techniques are important in this situation. Failing to parallelize this loop in TRFD would result in a program slowdown of a factor of 13.

Program DYFESM is another example that could take advantage of new data dependence tests. Subscripted-subscript patterns inhibit the detection of independent loops. However, symbolic analysis can determine that the access patterns formed by the subscript arrays are non-overlapping regions, each having a different length. The program initialization routine needs to be investigated to determine the values given to the subscript arrays, which are read-only from then on. A starting point for realizing such compiler capabilities could be the symbolic

range propagation techniques described in [BE95]. Without recognizing that these loops are independent, only inner loops could be parallelized, resulting in a program slowdown of a factor of 3.

2.5.3 Run-time analysis techniques

The described analysis may be complex or even undecidable at compile time. In these situations it may be useful to insert run-time tests that choose between a fully parallel and a serialized loop nest, depending on whether the values of the variables produce a dependence in the loop.

For example, in many loops within OCEAN (covering 65% of the serial execution of the program), the subscripting expressions and loop bounds contain variables and a linear combination of loop indices, which makes compile-time dependence analysis difficult. All of the arrays used in these loops are singly-dimensioned, but the subscript expressions used involve the loop indices of all the loops in the nest. The subscript expressions are such that they can be considered a linearization of a three-dimensional array into a single dimension. One difficulty in analyzing this situation is that, in different executions of the same loop nest, the innermost loop sometimes corresponds to the right-most dimension of the three-dimensional array and sometimes to the left-most dimension.

In one experiment, we implemented a test, which we call the *linearization test*, that checked whether the array was being used in this linearized manner. This involved checking whether each successive inner dimension indexed entirely within the next outer dimension. If this is the case, then there are no dependences on the references to those arrays. The result of this experiment is the loop shown in Table 9. A more detailed description of the test can be found in [Hoe92b].

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
OCEAN-ftrvmt/109	89.3	1376.9	15.4	7.3
OCEAN-csr/20	10.5	172.0	16.4	1.9
OCEAN-ftrvmt/116	10.8	99.5	9.2	1.5
OCEAN-acac/30	3.3	92.5	28.0	1.5
OCEAN-acac/40	4.0	79.0	19.8	1.4
OCEAN-scsc/30	2.3	59.3	25.8	1.3
OCEAN-rcs/20	3.4	57.7	17.0	1.3

Table 9: Performance impact of a run-time data-dependence test in the OCEAN program.

We have found that interprocedural symbolic analysis techniques also would enable the compiler to prove that the loops in Table 9 are dependence-free. Since the run-time test described above does not introduce significant overhead, the resulting performance would look similar to the one shown in Table 9. Thus, the linearization test could be applied either interprocedurally at compile time, or intraprocedurally with run-time tests. The trade-off between these two methods is doing a more complex compile time analysis versus introducing a small execution overhead for the run-time test and generating code that is more difficult to read.

3 Optimization summaries of the Perfect Benchmarks codes

In the previous section, we described transformation techniques that were applicable to several codes in our program suite. This section will summarize these transformations for each code and note additional individual changes that were made to the programs. For each code we will briefly discuss the expected difficulties for automating the transformations. Our experiments did not include the program SPICE. SPICE has been discussed in a related project, which found run-time parallelization techniques to be applicable to the irregular structure of this code [RP95].

The short descriptions of the problems being solved in the Perfect Benchmarks are taken from [Poi90]. Detailed optimization reports can be found in individual technical reports available from the University of Illinois[5]. A more thorough discussion of the automatic parallelization done by both KAP/Cedar and VAST is in [BE92]. Both restructurers were similar in terms of the resulting code and performance.

ADM "is a three-dimensional fluid flow code that simulates pollutant concentration and deposition patterns in lakeshore environments by solving the complete system of hydrodynamic equations. The advection-diffusion equation for the transport, diffusion, and deposition of pollutants is also included in the model [Poi90]." The program code is 6104 lines of Fortran77 code and consists of 97 subroutines. The execution time is spread evenly throughout the program; 90% of the execution time is spent in 23 subroutines. Almost all of these subroutines contain three or fewer loop nests, all of which are important. Because of this, 90% of the program's execution time is spread across 31 loop nests. Thus, a large number of loops need to be parallelized to get significant program speedups. However, 11 of these 31 important loop nests contain subroutine calls. Almost all of these calls are made to subroutines containing important loop nests.

KAP was unable to improve ADM significantly because of many subroutine call statements and the small iteration counts of the parallel loops. For the six most important loop nests, iteration counts ranged from 1 to 16 and four of them were singly nested. For other parallelizable loop nests, the iteration counts were 13–15 or 64. Most of these loops were singly-nested. KAP/Cedar stripmined all singly-nested parallel loops without checking the profitability, which introduced more overhead than performance gain. The best speedup that KAP was able to get was 1.65. Even by manually parallelizing the loops without call statements, we were unable to get speedups much greater than two.

The full manual optimization of ADM led to a speedup of 10.1 on Cedar. All of the most time-consuming loop nests in this code were transformed into concurrent loops. The main transformations applied to gain outer concurrent loops were array privatization and reduction parallelization. To automate these transformations, a compiler would have to recognize definition/use patterns interprocedurally. Parts of arrays that are read-only would have to be separated from privatizable read-write parts. Another issue is the recognition of the low iteration counts of certain loops in order to disable their high-overhead parallel execution. Several such iteration counts depend on input data in ADM, which necessitates the application of techniques such as run-time analysis or the coalescing of several nested parallel loops.

[5]CSRD Technical Report Series, Dept. of Computer Science, 1304 W. Springfield Ave., Urbana, IL 61801

ARC2D "was developed at NASA/Ames and run on a Cray X-MP. It is a robust, general-purpose, implicit finite-difference code for analyzing fluid flow problems. It solves the Euler equations. ARC2D can be used for steady and unsteady flows, but only for inviscid flows [Poi90]." The code contains 4,000 lines of Fortran77 in 74 subroutines.

ARC2D shows the best speedup from automatic parallelization on both FX/8 (8.7) and Cedar (13.5). There are about 30 loop nests that need to be parallelized well for good overall performance. KAP parallelizes all loop nests. Loop interchange was important in several cases.

The additional improvements that were applied manually include balanced stripmining methods, finding outer parallel loops, and localizing scalar data accesses which refer to global memory. These improvements resulted in substantially improved speedups of 10.6 for the FX/8 and 20.8 for the Cedar machine.

The key to automating these transformations is expected to be array privatization, which is needed to find the outer parallel loops. The necessary definition/use analysis requires advanced techniques for interprocedural propagation of symbolic values, relations, and conditions under which relations hold. Similar symbolic analysis steps will be necessary to find whether the privatized arrays are accessed outside the loop.[6]

BDNA "makes use of the BIOMOL package for performing molecular dynamics simulations of biomolecules in water. This package aims at an understanding of the hydration, structure, and dynamics of nucleic acids and, more broadly, the role of water in the operation of biological systems [Poi90]." The code contains 4,000 lines of Fortran77 in 76 subroutines.

BDNA gains insignificant speedup from automatic parallelization by KAP (1.9). VAST does somewhat better on this code on the Alliant FX/8 machine because it could parallelize simple reduction operations better than KAP. There are two loop nests that dominate the computation. KAP could vectorize only the innermost loops of these nests, whereas VAST generated vector-concurrent versions.

Both loop nests could be manually parallelized at the outermost level with a resulting speedup of 5.6 and 8.5 for the FX/8 and Cedar machines, respectively. The first loop (actfor/240) was parallel after privatizing many scalar and array variables. The other loop nest (actfor/500) was parallel after dealing with several reduction operations. It is interesting to notice that, on the FX/8 architecture, the best performance of the actfor/500 nest resulted from vector-concurrent execution of the stripmined inner loop (loop 350) as opposed to full vectorization of this loop and running the outer loop concurrently. This is due to the large amount of loop-private data, which exceeds the cache capacity. This is the only significant code pattern seen in the Perfect Benchmarks suite where stripmining (or "blocking") for cache locality is of benefit on the FX/8 architecture. In the Cedar variant of BDNA, this is no longer an issue because the inner loop is stripmined for cluster parallelism, while the outer loop is spread across Cedar. The computational part of BDNA speeds up by a factor of 13 as a result of the manual transformations. However, the overall speedup is 8.5 because of an input/output loop (Restar/15), which becomes a significant serial bottleneck and takes 30% of the parallel execution time.

The most difficult step in automating these transformations is expected to be the definition-use analysis for privatizable arrays. In one case, the content of an index array needs to be

[6] This is not necessarily a requirement for parallelization, but can simplify the generated code.

analyzed in order to privatize.

DYFESM "is a two-dimensional finite element code for the analysis of symmetric anisotropic structures. An explicit leap-frog temporal method with substructuring is used to solve for the displacements and stresses, along with the velocities and accelerations at each time step [Poi90]." The code contains 7,600 lines of Fortran77 in 113 subroutines.

Automatic parallelization by KAP yielded a speedup of 4 on the Alliant FX/8 over the unoptimized code. This is the third-best performance in the automatic column. However, this is only 1.2 times the performance of the vector-only code. On the Alliant, the performance achieved by VAST is only about 2.5 because of the overhead introduced by the more aggressive parallelization of reduction operations. DYFESM has very small data sets, which is one of the impediments to good performance. It is also the reason for the automatically parallelized version on Cedar yielding a speedup of only 2.2. The compiler we used does not distinguish small loops that are below the benefit threshold for parallel execution. On the FX/8 machine, this is a less significant problem because the startup of a parallel loop is comparatively fast. On Cedar, the loop startup latency is more significant, leading to greater performance degradation.

Manual improvements include the replacement of a matrix multiplication[7] by a library routine, and the transformation of many of the outer loops into concurrent form; for one loop (mxmult/10), a critical section was created to coordinate parallel reduction operations (see Section 2.2). Furthermore, arrays were privatized to some of the outer, now concurrent loops. The resulting improved speedups are 10.3 and 11.4 for FX/8 and Cedar, respectively.

A major difficulty in automating the transformations is the analysis of subscripted subscripts. DYFESM accesses arrays through table arrays in all its computation. The content of some of these subscript arrays can be derived from the program text. In other cases, such as the major loops mxmult/10 and formr0/20, the patterns probably can not be investigated at compile time.

FLO52 "This two-dimensional code provides an analysis of the transonic inviscid flow past an airfoil by solving the unsteady Euler equations[Poi90]." The code contains 2,000 lines of Fortran77 in 64 subroutines.

KAP produced the best speedup numbers of any Perfect Benchmarks program for FLO52 running on the FX/8 (9.0). The speedup of FLO52 on Cedar produced by KAP was second best (5.5, behind the speedup of ARC2D). The reason for these numbers is that most of the loops in FLO52 are simple, with relatively straightforward indexing patterns and no subroutine calls. The most basic dependence analysis can determine that these loops are parallel. Three of the most important loops contained a number of inner loops, for which KAP tried to find the optimal configuration in terms of distributing and interchanging them. Given appropriate compiler options that extended the length of the search, KAP was able to find the best configuration.

The manual efforts to optimize FLO52 centered on parallelizing outer loops in three cases (psmoo/40, psmoo/80, and step/20); fusing two of the loops (psmoo/40 and psmoo/80); privatization; turning off recurrence recognition; and improving scheduling for several parallel loops, as explained in Section 2.2 and 2.4. KAP had not parallelized these outer loops because of

[7] This was not done automatically by our compilers. However, the technology is well known and is not covered in this paper.

an internal error in the interface between two compiler passes. The speedups after the manual effort for FX/8 and Cedar are 14.6 and 15.3, respectively.

The recurrence recognition in KAP had to be turned off to reduce overhead. Reasons for this are described in Section 2.2. The loop fusion used in psmoo/40 and psmoo/80 was straightforward, except that some data written between the two loops had to be privatized and redundantly computed to enable the fusion.

The privatization done within FLO52 was for both arrays and scalars. There were many instances in which a global scalar was read-only in a loop. The Cedar architecture has the capability to prefetch vector loads from global memory. However, the prefetch unit is not used for scalars; therefore, it can be advantageous to copy a global scalar to a location in each cluster, and use it from there within a cross-cluster parallel loop. It was necessary to collect all references to an array within a loop nest and then piece together the ranges in which it was referenced, in order to ensure that the condition for privatization was met. There was also a case (in `psmoo`) in which an array could be fully distributed to the clusters for the duration of a single loop. One quarter of the array was copied to each cluster `COMMON` area in each iteration of an SDOALL loop. The region was then copied back to the proper global location at the end of the iteration.

Finally, some scheduling improvement was due to the balanced stripmining technique, described in Section 2.4.1. We expect that the transformations we applied to FLO52 will be straightforward to implement in an automatic translator.

MDG "is a molecular dynamics model for 343 water molecules in the liquid state at room temperature and pressure. The code uses the Matsuoka-Clementi-Yoshimine configuration interaction potential for rigid water-water interactions and extends it to include the effects of intra-molecular vibration. MDG can be used to predict a wide variety of static and dynamic properties of liquid water[Poi90]." The code contains 1200 lines of Fortran77 in 51 subroutines.

MDG has no speedup from automatic parallelization. None of the major loops in this code were parallelized, because KAP detected data dependences.

The two most time-consuming loops (interf/1000 and poteng/2000) were transformed manually into parallel loops mainly by privatizing some arrays and recognizing reductions that can be done in parallel. Other transformations done were localizing read-only scalar accesses(interf/1000), eliminating induction variables (interf/1000,poteng/2000), loop coalescing and removing storage-related dependences (predic/1000), and parallelizing a loop that wasn't detected as independent by KAP (correc/1000). These changes resulted in drastic improvements of 7.3 for the FX/8, and 20.6 for the Cedar machine.

The primary challenge in parallelizing MDG automatically is to detect all privatizable variables. In Section 2.1, we have described these patterns, which require advanced symbolic analysis techniques.

MG3D "is a seismic migration code used to investigate the geological structure of the Earth[Poi90]." The code contains 2800 lines of Fortran77 in 64 subroutines.

KAP parallelized most loop nests not containing subroutine calls or input/output statements, except for those loops within certain routines (`cpass`, `cpassm`, `rpass`, and `rpassm`) that contain potential dependences. KAP left those loops serial since their parallelization depends

on the run-time values of certain variables. There was an insignificant speedup of 1.5 for the FX/8 machine, and even a slowdown to 0.9 for Cedar.

The manual effort required extensive interprocedural analysis, array privatization, and parallel accumulation to parallelize the major loop (migrat/200); recognition that a temporary file can be placed in memory; dead code elimination; and symbolic subscript analysis to vectorize loops in the FFT routines (`cpass`, `cpassm`, `rpass`, and `rpassm`). The resulting performance numbers were among the best ones for the entire suite: 13.3 for the FX/8, and 48.8 for the Cedar machine.

MG3D, as a benchmark code, has some properties that may become important for an advanced compiler, but which are likely to be different in a real code. First, MG3D uses two temporary files, whose I/O operations can be eliminated by replacing them through memory operations. Unfortunately, the original code does not delete these files at the end; thus, this transformation cannot be performed without changing the effect of the program (because there is the possibility that the file will be used later). However, this optimization was applied in our hand experiments, enabling the outermost loops to be parallel. Second, part of the data set for MG3D is generated in the code and is rather artificial, so that an advanced compiler could recognize the values (e.g., through constant propagation) and perform further optimizations. This was not exploited in our experiments.

OCEAN "solves the dynamical equations of a two-dimensional Boussinesq fluid layer. The code is needed in order to study the chaotic behavior of free-slip Rayleigh-Benard convection[Poi90]." The code contains 3400 lines of Fortran77 in 70 subroutines.

KAP was able to parallelize only inner loops and trivial perfectly-nested loops. The most important loops in the code (covering 65 % of the serial execution) were not parallelized by KAP due to the form of the subscript expressions and the use of scalar variables as expression coefficients and loop bounds. On the FX/8 machine, this resulted in a slight speedup of 1.4, whereas it resulted in a slowdown to 0.7 on Cedar.

VAST was able to parallelize an extra level in the same nests where KAP could parallelize only the innermost level. VAST accomplished this by setting a logical variable holding a condition that is true when the loop is dependence-free. The loop was made parallel and the logical variable then controlled the issue of **await** and **advance** around the loop body. In some nests, several dependence conditions were evaluated and tests were made for all of them inside the parallel loop to determine whether to issue the **await**s and **advance**s.

Our manual optimization effort improved the OCEAN speedups to 8.9 on FX/8, and 16.7 on Cedar. As with many other applications, the key was to parallelize many of the outer loops. To do this, we worked on three types of loops. The first type was a group of loops using variables as coefficients in the subscript and loop bound expressions. This group required a run-time dependence test inserted prior to the loop, which chose between a serial and a parallel version of the loop, and, in one case, the coalescing of a triangular loop nest into a singly-nested XDOALL, as described in Section 2.4.2. The second type was a group of loops which required interprocedural analysis to parallelize the loops. The third type was two small, simple, and often-used loops, whose number of iterations varied widely. We generated multiple optimization versions of these loops, such that cluster and cross-cluster parallelism was exploited only when the number of iterations was large enough.

One issue with implementing these techniques is to determine when it is of benefit to apply

multi-version loops and when the overhead offsets the benefit. It seems clear that advanced symbolic analysis techniques that work interprocedurally can significantly reduce the work necessary to detect parallelism at run-time. However, further work is necessary to determine the extent to which this is possible.

QCD "This quantum chromodynamics (QCD) code uses a Monte-Carlo technique to update the complex 3x3 matrices that represent the action of the gluons. Since the code ignores the effect of dynamical fermions, it may be considered a pure-gauge model in the *quenched* approximation[Poi90]." The code contains 2,300 lines of Fortran77 in 69 subroutines.

KAP was able to parallelize only minor, inner-most loops in QCD, stripmining them to be SDOALL/CDOALL/vector. For most of these loops, the low iteration count prevented more than one processor from being used since 32 iterations are applied to the vector instruction first. KAP failed to parallelize most loops due to the large number of subroutine calls throughout the loops in the program. The resulting speedup was insignificant.

VAST performed well in loop observ/2 by recognizing multiple summation statements (a series of statements like `TOT(1) = TOT(1) + ...`), putting them in a synchronized region, and pulling out all extraneous calculations from the parallel loop. VAST also was able to stripmine a **dotproduct** reduction in loops projec/2 and projec/4.

We had to analyze QCD interprocedurally in order to parallelize its outer loops. A simple tool allowed us to make a summary of interface variables that are read and written within each called routine, including all inner routines. Data dependence analysis based on this summary, in conjunction with induction variable analysis, allowed us to parallelize most of the outer loops in the program. Once again we duplicated subroutines to choose a good transformation for each different calling context.

Subroutine `pranf` contains a random number generator producing a dependence cycle that could not be eliminated. This dependence serialized approximately half the execution of the program. In our best-performing program variant, we ignored this dependence, thus effectively replacing the random number generator with a parallel variant. This changed some of the program output, and in fact, the built-in verification test reported "invalid". Only knowledge of the application allows us to determine that the program is behaving correctly. These improvements resulted in a speedup of 20.8 on the Cedar machine. A variant without replacing the random number generator yielded a speedup of only 1.8.

SPEC77 "is a global spectral model for simulating atmospheric flow. The code was originally developed at the National Meteorological Center (NMC). Only the forecast module of the NMC code is used in the benchmarking[Poi90]." The code contains 3,900 lines of Fortran77 in 99 subroutines.

KAP was unable to transform most major loops due to the large number of subroutine calls in this program. Typically, KAP could parallelize nothing beyond innermost loops, yielding a speedup of 2.4 on both machines.

Our manual experiments improved these numbers for the FX/8 and Cedar machines to 10.2 and 15.7, respectively. Some advanced manual transformations were done to the code, including converting COMPLEX arrays to double-sized REAL arrays, eliminating subroutine parameters, replacing algorithms, and replacing multiple calls to a subroutine with a single call. However, the transformations that produced the greatest reduction in execution times were the simplest

ones, all of which are important for other codes as well. They include the parallelization of loops with subroutine calls, array privatization, and parallelized accumulation. Other transformations were of importance in individual loops, such as advanced loop interchanging, and the fusion of loops with different loop bounds.

The most critical transformation to automate is the parallelization of a search routine. This routine uses the last found position as a starting point for the next search, which causes a data dependence that is difficult to break. However, it seems possible to develop advanced compiler techniques to recognize whether in the given situation, this dependence can be ignored. Serializing this dependence would slow down the best version of the code by a factor of about 2.5.

All the analysis techniques we have already described are important for this code as well. In addition, the initialization of SAVEd variables had to be recognized and moved out of parallel loops, or synchronized. Interprocedural analysis techniques include the propagation of maximum variable values.

We applied sequential optimizations to SPEC77 prior to its parallelization. The major change was to replace the formatted read of an input file with an unformatted read. Again, this is a questionably automatable technique for which the compiler may make suggestions to the user, at best. Replacing the formatted I/O by unformatted I/O improves the best performance of the code by a factor of two.

TRACK "This missile tracking code is used to determine the course of a set of an unknown number of targets, such as rocket boosters, from observations of the targets taken by sensors at regular time intervals. The targets may be launched from a number of different sites[Poi90]." The code contains 4300 lines of Fortran77 in 66 subroutines.

KAP and VAST were unable to speed up this code. From an application viewpoint, the problem consists of a number of independent tasks that track missiles, which would seem highly-parallelizable. The serious data dependences that effectively serialize this code stem from storage management functions that are called from all tracking tasks.

Correspondingly, the most important manual transformation was to allow these storage management functions to be called in parallel. To adhere to the sequential semantics of the code, sort functions had to be inserted to rearrange the order of storage allocation. This transformation is difficult to automate, though not impossible.

Another important transformation was to parallelize loops containing `return` statements. In general, when no data dependences exist in such loops, they can be parallelized as long as we can guarantee that no changes to global data occur due to iterations of the parallel loop which would not have executed in the serial version of the program.

One typical reason for having `return` statements in a loop is to return an error code which signals a fatal error, causing the program to abort. This is the case in TRACK. Interprocedural flow analysis could determine that whenever such a `return` statement is executed, the program later executes a `stop` statement without making use of any of the variables modified within the loop, and therefore any iterations which get executed in the parallel loop, but not the serial loop, would have no important effect. This allows such a loop to be parallelized. In TRACK, we were even able to move the error condition test and the `return` statement out of such loops, then parallelize the loops without concern for unwanted side-effects.

There were some performance-limiting factors which we could not eliminate. A significant

number of loops contain `read` and `write` statements and these become a serial bottleneck once the computational loops have been parallelized. There is also a lack of vectorizable statements. So in this program, concurrent execution cannot make use of vector instructions.

Our efforts yielded a speedup of 4.0 and 5.2 for the FX/8 and Cedar, respectively.

TRFD "is a kernel simulating the computational aspects of a two-electron integral transformation and part of the HONDO quantum mechanical package. The evaluation of these types of integral transformations is a necessary first step in computing correlated wave functions and is used in determinations of molecular electronic structure[Poi90]." The code contains 500 lines of Fortran77 in 42 subroutines.

KAP, again, could parallelize only inner or perfectly-nested loops within TRFD. It was able to parallelize the inner loop of intgrl/140 by placing **await/advance** synchronization around the assignment of a value into an array which was indexed by a subscripted array. This resulted in a modest speedup of 2.2 on the FX/8, but a performance degradation to 0.5 on Cedar.

VAST, again, was able to parallelize one level beyond KAP in the innermost loops by calculating the condition for the presence of a dependence and then synchronizing around that dependence, if it occurs at run-time. VAST expanded a scalar within intgrl/140 in the same situation where KAP privatized the scalar to produce a slightly simpler code. This did not affect the performance significantly, however.

The manual optimization of this program resulted in a drastic speedup improvement: 16.0 for the FX/8, and 43.2 for the Cedar machine. The manual transformations for TRFD include generalized induction variable analysis (GIV) for computing the value of an induction variable within a triangular loop nest, and array privatization. The loop intgrl/140 was parallelized at the outermost level after we determined that the form of initialization of the subscripting array made subscripted-subscript access parallelizable.

We described this situation in an earlier paper [EHLP91].

4 Early results of a new generation parallelizing compiler

The objective of this paper was to show new transformations that can improve future parallelizing compilers. We have already completed the first version of such a compiler, called Polaris, that incorporates some of these techniques. In this section, we present early results that demonstrate Polaris' success and areas for potential improvement. The specific transformations and their implementations are not discussed in this paper. We refer interested readers to [TP93], [BEH+94], [BE94b], and [PE95], which describe the details of the algorithms used. The sole purpose of this section is to give some evidence that the automation of the hand transformations outlined in this paper is a feasible goal, and to point out problems that may be hard to solve.

Table 10 lists the Perfect Benchmarks and indicates, for each program, whether Polaris is able to recognize all the significant parallel loops in Tables 3 through 9. As shown in Table 10, Polaris can recognize all significant parallel loops in about half of the programs. For the remaining programs, we indicate which issues need to be resolved in order to succeed in the parallelization. For details on these issues, the reader is referred to the description of the individual codes in Section 3.

Table 10 shows that significant progress in automatic parallelization is possible with the techniques described in this paper. Only two of the Perfect Benchmarks could be parallelized

program	successfully parallelized (• = fully) (◦ = partially)	open problems
ADM	[1]	Interprocedural array section analysis, using run-time knowledge.
ARC2D	•	
BDNA	•	
DYFESM	[1]	Analysis of subscripted subscripts
FLO52	•	
MDG	•	
MG3D		Converting temporary file I/O to temporary data structures.
OCEAN	◦ [1]	Loop coalescing. Improved range analysis
QCD		Substitution of a random-number algorithm.
SPEC77		Substitution of a search algorithm.
SPICE	[1]	Analysis of subscripted subsctripts and irregular control flow.
TRACK		Parallelization of memory allocation algorithms.
TRFD	•	

[1] These issues are being addressed in ongoing Polaris work[BDE+96, BEH+94]

Table 10: Success and unsolved problems in the automatic parallelization of the Perfect Benchmarks achieved by a prototype of the Polaris parallelizing compiler.

successfully when we began our project. Now there is success with almost 50% of the codes. In some cases, we have already succeeded in solving "hard" problems, such as the analysis of subscripted subscripts in BDNA and the symbolic comparison of guarded array sections in MDG. Other difficult issues are left open for future research. Some of these issues appear tractable as compiler technology evolves and as increasing computer speeds enable time-consuming compilation algorithms. An example of this is the analysis of subscripted subscripts in DYFESM. Other issues may seem resolvable only after the programmer becomes more involved in an interactive compilation scenario. An example of this is the random number generator in QCD, which can be replaced by a parallel algorithm (this changes the sequential semantics of the program, however, and can be authorized only by the user).

An important remaining question is whether the techniques derived from the experiments with the Perfect Benchmarks on the Cedar machine will carry over to new machines and programs. In order to investigate this, we added to Polaris the capability of generating code for the SGI Challenge machines and tested many additional programs taken from other suites, such as the SPEC95 benchmarks and two "Grand Challenge Applications".

Figure 4 compares the resulting speedups obtained by Polaris with those of SGI's PFA compiler. It shows sixteen benchmark programs, from three different sources. From the Perfect Benchmark suite we used ARC2D, BDNA, FLO52, MDG, OCEAN, and TRFD. From the SPEC95 benchmarks we chose applu, appsp, hydro2d, su2cor, swim, tfft2, tomcatv, and wave5. From applications in use by computational scientists at NCSA[8], we chose CMHOG and CLOUD3D.

The programs were executed on an eight-processor SGI Challenge with 150 MHz R4400

[8] National Center for Supercomputing Applications

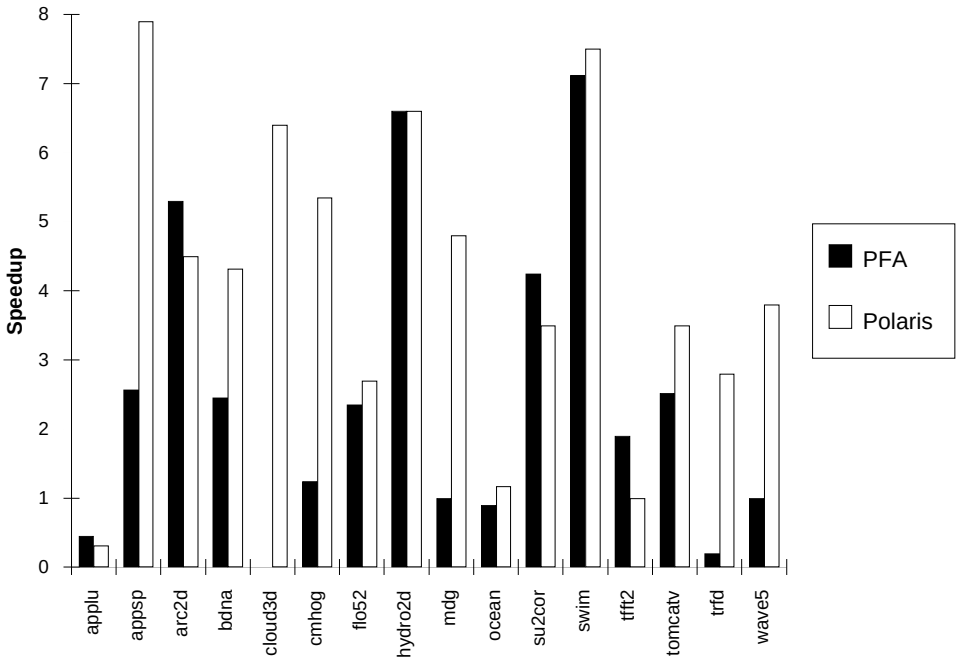

Figure 4: Speedup Comparison Between the SGI PFA Compiler and Polaris.

processors, located at NCSA. Figure 4 shows that Polaris delivers, in many cases, substantially better speedups than PFA. For a few of the programs there is little speedup, and in one case there is a slowdown. We have studied and discussed the reasons for this in [BDE+96]. We found that Polaris is significantly more successful in identifying parallel loops. However, in the programs where PFA identifies the important parallel loops equally well, its additional techniques for improving the parallel code make a difference. These transformations include loop interchanging, unrolling, and fusion. When applied to the right loops, they can improve performance by decreasing overhead, enhancing locality, and facilitating the detection of instruction-level parallelism. However, sometimes these optimizations are applied over-aggressively and cause a negative effect. This is the case on `applu` and `tomcatv`.

5 Conclusions

We have described the transformations that we applied to the Perfect Benchmarks programs in order to gain significant parallel performance. We have noted the expected challenges in automating these techniques in a parallelizing compiler. Among the most important techniques that we have found are array privatization, parallel reductions, generalized induction variable recognition, and symbolic or run-time data-dependence tests. All techniques need powerful interprocedural analysis capabilities.

We have drawn several conclusions from this work:

Real programs are parallelizable. In the suite of programs we have examined, we found that, without exception, significant performance improvements can be gained by transforming the programs into parallel form. Most transformations are simple and do not require algorithm change. This is an important basis for all further findings, since it refutes claims that ordinary application programs are not amenable to parallel computing.

The study of real programs is of crucial importance for compiler research. We have taken an approach to compiler design that is in stark contrast to other methods. Rather than implementing new compiler capabilities and then assessing their merit, we have optimized real programs by hand and described the transformations that make a difference. Our results show that this approach can be very successful. We have identified several new transformation techniques that improve the performance of our programs significantly. The resulting compiler proves to be powerful not only on the originally studied program suite, but also on new programs and new machines.

Finding parallelism is important for success on all multiprocessors. The techniques we found are important for all types of parallel architectures. We have mainly considered transformations that can identify and generate parallel loops. The transformation of such loops is necessary for successful parallel implementation of any program on any parallel machine. Although our work has been done in the context of the Cedar shared-memory machine, its applicability goes much beyond Cedar to both present and future multiprocessors. Early results of a new compiler that incorporates the proposed techniques show substantial performance gains

over state-of-the art compilers on an SGI Challenge machine. Similar results were reported on a Cray T3D machine [PP96] and a Sun multiprocessor [EPV96].

Privatization may make data distribution less important. Much recent work on programming techniques for distributed-memory machines has focused on data-distribution techniques. In our study we have not found such techniques to be of significant importance. Instead, privatization techniques, which provide a natural way of placing data with the referencing processors, may deserve more consideration. If large amounts of data are privatized, less needs to be shared, and less needs to be distributed.

These techniques apply to other programs. The more our work with the Perfect Benchmarks applies to other programs, the more important it turns out to be. When testing its applicability to additional programs, not in the Perfect suite, we have found that the new techniques **do** apply. In fact, in many cases we have not been able to improve the Polaris-optimized code manually. Nevertheless, our future work will include studies with even more realistic applications, such as programs found among the "Grand Challenge" applications and the SPEChpc96 benchmarks [EH96].

Parallelizing compilers can be improved significantly. Perhaps the most important conclusion for the field of our direct interest is that this work has given strong indication that substantially more powerful parallelizing compilers can be built. We believe this is very significant because there appears to be a growing belief that this is not possible. Some people even have used our previous work as evidence that despite many years of compiler research and development, commercial parallelizers are very limited in their effectiveness on real programs. That conclusion was premature, on their part, because our work now indicates that there is potential for much improvement of automatic parallelization.

Acknowledgments

Some of the experiments described in this paper were done by Greg Jaxon and Zhiyuan Li while they were members of our research group at CSRD. Their contributions were essential for the success of this project.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[BDE+96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, William Pottenger, Lawrence Rauchwerger, and Peng Tu. Advanced Program Restructuring for High-Performance Computers with Polaris. Technical Report 1473, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1996.

[BE92] William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions of Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[BE94a] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proc. of Supercomputing '94, Washington D.C.*, pages 528–537, November 1994.

[BE94b] William Blume and Rudolf Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. *Proc. of the 1994 Int'l Conference on Parallel Processing*, pages II233 – II238, August, 1994.

[BE95] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proc. of the 9th Int'l Parallel Processing Symposium*, pages 357–363, April 1995.

[BEH+94] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994.

[BENP93] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[EH96] Rudolf Eigenmann and Siamak Hassanzadeh. Benchmarking with real industrial applications: The SPEC High-Performance Group. *IEEE Computational Science & Engineering*, 3(1):18–23, Spring 1996.

[EHJ+93] Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David Padua. Restructuring Fortran Programs for Cedar. *Concurrency: Practice and Experience*, 5(7):553–573, October 1993.

[EHLP91] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science, 589*, pages 65–83, August 1991.

[Eig93] Rudolf Eigenmann. Toward a Methodology of Optimizing Programs for High-Performance Computers. *Proc. of the Int'l Conf. on Supercomputing 1993, Tokyo, Japan*, pages 27–36, July 20-22, 1993.

[EM93] Rudolf Eigenmann and Patrick McClaughry. Practical Tools for Optimizing Parallel Programs. *Proc. of the 1993 Simulation Multiconference on the High-Performance Computing Symposium, Arlington, VA*, March 27 - April 1, 1993.

[EPV96] Rudolf Eigenmann, Insung Park, and Michael J. Voss. Are parallel workstations the right target for parallelizing compilers? *Proc. of the 9th Workshop on Languages and Compilers for Parallel Computing*, August 96.

[For93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical report, Rice University, Houston Texas, May 1993.

[GJT+91] K. Gallivan, W. Jalby, S. Turner, A. Veidenbaum, and H. Wijshoff. Preliminary Basic Performance Analysis of the Cedar Multiprocessor Memory Systems. *Proc. of the Int'l Conference on Parallel Processing 1991, St. Charles, IL*, I:71–75, August 12-16, 1991.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. *Proc. of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[GMS+95] M. Gupta, S. Midkiff, E. Schoenberg, B. Seshadri, D. Shields, K.Y. Wang, M.M. Ching, and Ton Ngo. An HPF compiler for the IBM SP-2. *Proc. of Supercomputing'95*. ACM Press, New York., San Diego, California, 1995.

[Hoe92a] Jay Hoeflinger. Coalescing Triangular Loops. Technical Report 1364, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, January 1992.

[Hoe92b] Jay Hoeflinger. Run-Time Dependence Testing by Integer Sequence Analysis. Technical Report 1194, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1992.

[HP91] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. *Parallel and Distributed Computing: Advances in Languages and Compilers for Parallel Processing, MIT Press, Cambridge, MA*, pages 310–330, 1991.

[HP93] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs. *Proc. of the 6th Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[Hus82] Christopher Alan Huson. An In-Line Subroutine Expander for Parafrase. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Sci., Dec., 1982.

[KBC+74] D. Kuck, P. Budnik, S-C. Chen, Jr. E. Davis, J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle. Measurements of Parallelism in Ordinary FORTRAN Programs. *Computer*, 7(1):37–46, Jan., 1974.

[KDL+93] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U.M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner. The Cedar System and an Initial Performance Study. *Proc. of the 20th Int'l. Symposium on Computer Architecture, San Diego, CA*, pages 213–224, May 16-19, 1993.

[Li92] Zhiyuan Li. Array privatization for parallel execution of loops. *Proc. of the Int'l Conf. on Supercomputing 1992*, pages 313–322, 1992.

[MAL92] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. *Proc. of the 5th Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[MHL91] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *SIGPLAN NOTICES: Proc. of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28*, pages 1–14. ACM Press, 1991.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proc. of the Int'l Conf. on Supercomputing 1995*, pages 444–448, 1995.

[Poi90] Lynn Pointer. Perfect: Performance Evaluation for Cost-Effective Transformations Report 2. Technical Report 964, Univ. of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, March 1990.

[PP96] Yunheung Paek and David Padua. Automatic parallelization for noncoherent cache multiprocessors. *Proc. of the 9th Workshop on Languages and Compilers for Parallel Computers*, August 96.

[Pug92] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[RAP95] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. Run-Time Methods for Parallelizing Partially Parallel Loops. *Proc. of the Int'l Conf. on Supercomputing 1995, Barcelona, Spain*, 1995.

[RP95] Lawrence Rauchwerger and David A. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. *Proc. for the 9th Int'l Parallel Processing Symposium*, April 1995.

[SH91] J.P. Singh and J.L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. *Proc. of the Int'l Symposium on Shared Memory Multiprocessing, Tokyo, Japan*, April 1991.

[TP93] Peng Tu and David Padua. Automatic Array Privatization. *Proc. of the 6th Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science*, volume 768, pages 500–521, August 12-14, 1993.

[Wol92] Michael Wolfe. Beyond induction variables. In *Proc. ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 162–174, 1992.