Advanced Program Restructuring for High-Performance Computers with Polaris*

William Blume Ramon Doallo Rudolf Eigenmann John Grout
Jay Hoeflinger Thomas Lawrence Jaejin Lee David Padua
Yunheung Paek Bill Pottenger Lawrence Rauchwerger Peng Tu

Center for Supercomputing Research and Development,
University of Illinois at Urbana-Champaign,
415 Computer and Systems Research Laboratory
1308 W. Main Street,
Urbana, IL 61801-2932, USA
(217) - 333-4223
Fax: (217) - 244-1351
padua@csrd.uiuc.edu

Abstract

Multiprocessor computers are rapidly becoming the norm. Parallel workstations are widely available today and it is likely that most PCs in the near future will also be parallel. To accommodate these changes, some classes of applications must be developed in explicitly parallel form. Yet, in order to avoid a substantial increase in software development costs, compilers that translate conventional programs into efficient parallel form clearly will be necessary. In the ideal case, multiprocessor parallelism should be as transparent to programmers as functional level parallelism is to programmers of today's superscalar machines. However, compiling for multiprocessors is substantially more complex than compiling for functional unit parallelism, in part because successful parallelization often requires a very accurate analysis of long sections of code. This article discusses recent experience at Illinois on the automatic parallelization of scientific codes using the Polaris restructurer. Also, the article presents several new analysis techniques that we have developed in recent years based on an extensive analysis of the characteristics of real Fortran codes. These techniques, which are based on both static and dynamic parallelization strategies, have been incorporated in the Polaris restructurer. Preliminary results on parallel workstations are encouraging and, once the implementation of the new techniques is complete, we expect that Polaris will be able to obtain good speedups for most scientific codes on parallel workstations.

Keywords: Compilers, Program Restructuring, Automatic Parallelization, Parallel Computing.

*The research described is supported by Army contract #DABT63-92-C-003. This work is not necessarily representative of the positions or policies of the Army or the Government.

1 Introduction

Parallel computing is widely accepted today as an enabling technology for many areas of science and engineering. As the limits of semiconductor technology are approached, parallel computing is also our best investment for continuing performance gains. Industry is quickly learning how to build powerful parallel computers. Programming these machines, however, remains a difficult task. The development of effective software techniques for parallel programming is the greatest "Grand Challenge" of high-performance computing today.

Unless this software challenge is met, the acceptance of parallel computers will continue to be slow and progress in computational science and engineering will be hampered as a result. From the perspective of the ordinary user, a powerful strategy to meet this challenge is to develop compilers to translate conventional programs into parallel form. Such compilers will enable the execution of the vast numbers of existing programs on new parallel machines, thus allowing a seamless transition into parallel computing. Furthermore, with the support of such compilers, new programs could be developed in the familiar sequential paradigm such that the programmer is freed from the complexities of explicit, machine-oriented parallel programming.

The importance of compilers for parallel computers is widely recognized and has been a very active area of research and development for the last several years. In this article, we report our recent experience in one such project at the University of Illinois. This work centers around Polaris, an experimental translator of conventional Fortran programs for a variety of parallel computers, including shared-memory multiprocessors and scalable machines with a global address space such as the Cray T3D.

In this article, we first briefly describe the restructuring techniques developed for Polaris and then present an evaluation of their effectiveness. In the last section, we discuss a number of new techniques that are necessary to improve the quality of the target parallel code.

The discussion of the techniques is brief, due to space limitations, but includes references to papers describing them in more detail. Although we did not include an extensive list of references, also due to space limitations, a more complete list can be found in [1], a recent survey paper.

2 Techniques Implemented in Polaris

Before starting the Polaris project, our group conducted a study of the effectiveness of automatic parallelization of Fortran programs [4]. The collection of analysis and transformation techniques now implemented in Polaris was primarily based on the conclusions of that study.

In that study, the Perfect Benchmarks, a collection of conventional Fortran programs representing the typical workload of high-performance computers, were automatically transformed by KAP, a state-of-the-art Fortran restructurer, for execution on the Alliant FX/80 and Cedar, a multiprocessor built at Illinois. Although the obtained speedups were satisfactory for a few of the programs, the restructurer failed to deliver any significant speedup for the majority of the programs.

The main reason for KAP's failure was its inability to successfully analyze and transform multiply-nested loops involving reasonably complex operations. In retrospect, this was not surprising because KAP was originally developed for vectorization. Commercial multiprocessors, such as the Alliant FX/80, were relatively new in the late 80s when our study was conducted. While vectorization isolates loop kernels and transforms them into vector operations, multiprocessors require the parallelization of large loop nests in order to amortize the overhead associated with concurrent execution. Vectorizers, in their process of isolating loop kernels and transforming them into vector operations, focus primarily on innermost loops, loops that perhaps have been moved to the innermost position as a consequence of loop-header interchanging. Multiprocessors, in contrast, focus on parallelizing the outermost loops in order to attenuate the effect of the overhead associated with the initialization of loop iterations as well as to increase the fraction of the code executed in parallel.

We determined that extending the four most important analysis and transformation techniques traditionally used for vectorization led to much greater speedups. Next we briefly discuss each of these four techniques and the extensions we found necessary to obtain good multiprocessor speedups.

2.1 Dependence Analysis

For a loop to be parallelizable, it is necessary and sufficient to determine that there are no *cross-iteration dependences*, that will arise when two different iterations access the same memory location and one or both accesses are a memory write.

Many techniques have been developed to determine the absence of cross-iteration dependences. These consider in turn every pair of references to the same array that are potential sources of cross-iteration dependences. The subscript expressions are analyzed to determine whether the dependence may actually exist. To guarantee correct code, dependence analysis techniques conservatively assume cross-iteration dependences when they are unable to accurately analyze the subscripts.

We illustrate these ideas using the loop:

```
        DO I=1,N
R:        A(2*I) = ...
S:        ...  = A(2*I)
T:        ...  = A(2*I+1)
        END DO
```

Cross-iteration dependences are possible between the statements R and R, R and S, and R and T. Notice that there cannot be a cross-iteration dependence involving S and T because there is no write to array A in either statement.

One simple dependence test, the *Equality Test*, can be applied to only singly-nested loops. The Equality Test determines that there are no cross-iteration dependences between two array references whenever the subscripts are identical expressions of the loop index and the coefficients are constant during execution of the loop. In the previous loop, the Equality Test will determine that there are no dependences between the statements R and R and between R and S. However, it will leave open the possibility of a cross-iteration dependence between R and T.

Another simple test is the *GCD Test*. It determines whether an equation involving both subscript expressions has an integer solution. Thus, to analyze the potential dependence between R and T, the GCD test considers the equation $2i = 2i' + 1$. Clearly, if there are no integer solutions to this equation, R and T will never access the same element of A.

The GCD Test determines that there are no integer solutions to the equation when the greatest common divisor of the coefficients does not divide the independent term. Thus, the previous equation has no integer solution because $GCD(2, 2) = 2$ does not divide 1. This shows that there are no dependences between `R` and `T`. Notice that in the loop above, the GCD Test does not disprove the two potential dependences disproven by the Equality Test. Therefore, both tests are necessary to obtain accurate results. In practice, parallelizing compilers apply a variety of these dependence tests in sequence.

The GCD Test requires not only that the coefficients be loop invariant, but also that their values be known at compile-time. Most techniques, including those in the version of KAP used in our previous study, have similar requirements. In fact, several important loops in our study were not parallelized because the values of the coefficients in the subscript expressions were not known at compile-time. We have developed the *Range Test* [2, 3], a dependence analysis algorithm with symbolic algebra capabilities. To determine when there are no cross-iteration dependences, this test makes use of information on the possibly symbolic range of values that a subscript expression may assume. To compute the range of values, we have implemented a *demand-driven* strategy built on top of a gated-SSA representation of the program [11]. The Range Test successfully analyzes a number of complex access patterns, including those where every iteration accesses regularly spaced array subsections.

2.2 Privatization

Temporary variables are often used inside loops to carry values between statements. An example is variable `T` in the loop:

```
DO I=1,N
  T = a(i) +1
  b(i) = T**2
  c(i) = T + 1
END DO
```

Temporary variables usually cause cross-iteration dependences because they are read and written on several loop iterations. However, when the temporaries are always assigned earlier in the same iteration in which they are read, they are said to be loop private, and replicating them enables both vectorization and parallelization. Thus,

the loop obtained by replacing each occurrence of `T` with a vector element, such as `TV(I)`, can be vectorized and parallelized. When translating for multiprocessors, another option is to declare private temporary variables local to the loop body. This would make a copy of the temporary variable for each processor cooperating in the parallel execution of the loop. In the case of the previous loop, a parallel version would be the following:

```
PARALLEL DO I=1,N
PRIVATE T
  T = a(i) +1
  b(i) = T**2
  c(i) = T + 1
END DO
```

For vectorization, it is often sufficient to identify loop private scalars. For many years vectorizers have used techniques for scalar privatization. However, in the case of multiply-nested loops, arrays are often used as temporaries. For example, consider array `U` in the loop:

```
NX = N + 1
DO I=1,N
  DO J=1,NX
    U(J) = ...
  END DO
  ...
  DO J=1,N
    ...  = U(J)
  END DO
END DO
```

An array is loop private if each element of the array that is read in the loop is first assigned within the same loop iteration, or is never assigned across the whole loop. Identifying private arrays is of great importance for parallelization. The lack of array privatization techniques was one of the primary causes of the low speedups obtained in our earlier study.

In Polaris, we have implemented a privatization technique to deal both with scalars and arrays [10]. This algorithm is substantially more complex than the traditional scalar privatization algorithm because it is necessary at each point within the loop to keep track of the array sections assigned earlier in the iteration and of the array section that is read. Furthermore, the array section bounds are often symbolic and therefore require symbolic

algebra algorithms to determine that assigned array sections cover the array sections that are accessed. It is also often necessary to transform the section bound expressions into a form that is appropriate for comparison. For example, in the last loop it is necessary to determine that NX has the value N+1 in order to determine that U(1:NX), the section of the array assigned in the first inner loop, covers U(1:N), the section read in the second loop. To this end, we use a demand-driven algorithm similar to the one mentioned earlier.

2.3 Induction Variable Recognition

Induction variables have integer values and are incremented by a constant on every loop iteration. An example is variable J in the loop:

```
J = 0
DO I=1,N
  J = J + 2
  U(J) = ...
END DO
```

The presence of induction variables precludes parallelization for two reasons. First, induction variables are read and written on every iteration and thus are the source of cross-iteration dependences. Second, a subscript expression involving induction variables cannot be analyzed by most dependence tests because they can handle only subscript expressions that are linear expressions of the loop index with loop invariant coefficients.

To avoid these problems, all occurrences of an induction variable are expressed in terms of the loop index, i.e., they are replaced by their closed form solution which can be evaluated concurrently for any value of the loop index. For example, the previous loop would be transformed into:

```
J = 0
DO I=1,N
  U(2*I+J) = ...
END DO
IF (N≥0) J=2*N+J
```

where the increment to the induction variable has been removed and, as a consequence, the induction variable is no longer an obstacle to parallelization.

Current vectorizers transform only induction variables that can be expressed in terms of a single loop index. However, in multiply-nested loops, induction variables could be incremented at several different levels of nesting which, when transformed, become complex expressions in terms of several loop indices. For example, in the loop:

```
        J = 0
        DO I=1,N
          ...
S:        J = J + 1
          ...
          DO K = 1, I
            ...
T:          J = J + 1
            ...
          END DO
        END DO
```

all occurrences of `J` after `S` could be replaced by `I + I*(I-1)/2`, and those after `T` within the inner loop by `I + I*(I-1)/2+K`.

Techniques to identify and replace induction variables in multiply-nested loops have been implemented in Polaris [6]. The implementation uses information on the range of values of variables and is supported by symbolic algebra algorithms.

2.4 Reduction Variable Recognition

Reduction variables are those incremented within the loop body, usually by floating-point values that change across iterations. Reduction variables preclude parallelization because they are read and written in different iterations. However, when a reduction variable is only read by the statements performing the increment, the loop can be transformed into parallelizable form. For example, in the following loop, if `Q` is accessed only in statement `S`, it can be transformed into parallelizable form.

```
        DO I=1,N
          ...
S:        Q = Q + A(I)
          ...
        END DO
```

One possible transformation could generate a loop whose parallel form would execute different parts of the iteration space in different processors. In the following loop, each processor computes a partial sum of Q, and the partial sums are added at the end to obtain the final value of Q.

```
        DO I=1,N
          PRIVATE PQ
S:        PQ = PQ + A(I)
          ...
        END DO
        Q = SUM (PQ)
```

Vectorizers considered only simple reductions, as in our example. However, as with Privatization and Induction Recognition, the traditional algorithms face some problems in the case of multiply-nested loops such as the following:

```
DO I=1,N
  DO J = 1, M
    K = func(I,J)
    ...
    A(K) = A(K) + expression
  END DO
END DO
```

Advanced techniques to transform loops like this into parallel form have been implemented in Polaris[6].

3 Effectiveness of Polaris Techniques

We now discuss the effectiveness of the six main analysis and transformation techniques implemented in Polaris. In addition to the four techniques presented in the preceding section, Polaris also applies *Autoinlining* [5] and *Interprocedural value Propagation (IPVP)*. Autoinlining expands all calls within loop bodies to subroutines with fewer than a given number of lines (50 for the experiments reported below). Interprocedural Value Propagation propagates the value of integer subroutine parameters across the entire program. When necessary, Interprocedural Value Propagation also clones subroutines to enable the parallelization of one or more loops. As discussed below, these two transformations, although not as powerful as a comprehensive interprocedural analysis algorithm, have

improved the accuracy of program analysis in several cases. An interprocedural analysis module that uses the demand-driven analysis techniques mentioned above is now being implemented in Polaris. For the experiments reported below, Polaris follows a trivial code generation strategy. After applying the analysis and transformation techniques, all outermost loops not containing cross-iteration dependences are annotated as parallel.

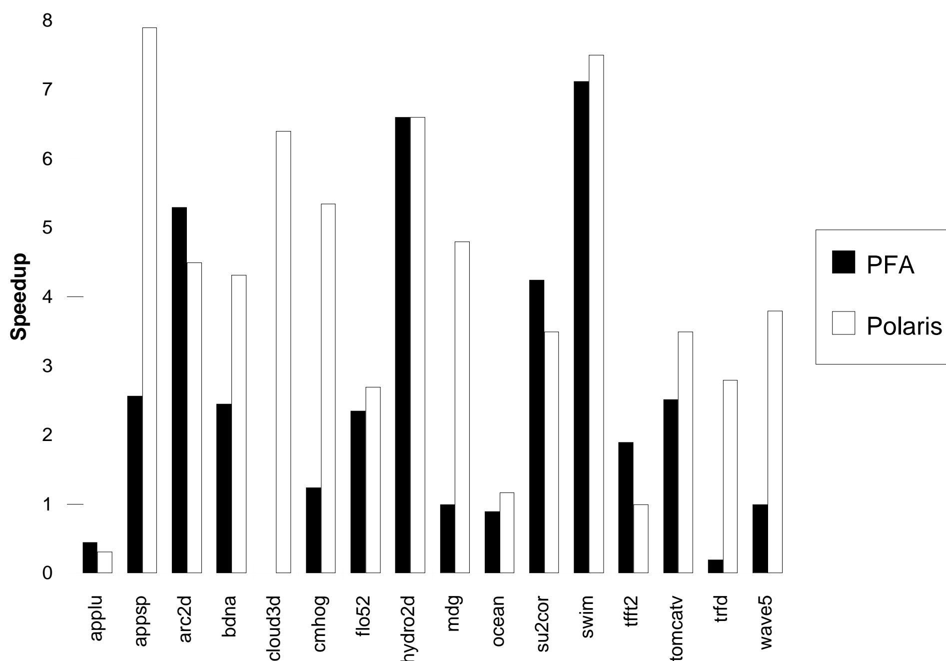

Figure 1: Speedup Comparison Between the SGI PFA Compiler and Polaris

Figure 1 presents a comparison of the speedups obtained by Polaris with those of SGI's PFA compiler. Twelve programs, described in Table 1, were used in the comparison. The programs were executed (in real-time mode for accuracy) on an eight processor SGI Challenge with 150 MHz R4400 processors. Figure 1 shows that Polaris delivers, in many cases, substantially better speedups than PFA.

However, for a few of the programs, the speedups are close to, or even below, 1. More will be said about future strategies for handling these these programs in the next section. In two of the sixteen programs, PFA produces better speedups than Polaris. The reason is that PFA uses an elaborate code generation strategy that includes loop transformations such as loop interchanging, unrolling, and fusion which, when applied to the right loops,

improve performance by decreasing overhead, enhancing locality, and facilitating the detection of instruction-level parallelism. However, the sophisticated code generation strategy has a negative effect on two of the codes, `applu` and `tomcatv`; although PFA detects as much parallelism as Polaris, the code it generates does not take much advantage of it.

Program	Description	Origin	# lines	Serial exec (seconds)
APPLU	Parabolic/elliptic PDE solver.	SPEC	3870	1203
APPSP	Gaussian elimination system solver.	SPEC	4439	1241
ARC2D	Implicit finite-difference code for fluid flow.	PERFECT	4694	215
BDNA	Molecular dynamics simulation of biomolecules.	PERFECT	4887	56
CMHOG	3D ideal gas dynamics code.	NCSA	11826	2333
CLOUD3D	3D model for atmospheric convective applications.	NCSA	9813	20404
FLO52	2D analysis of transonic flow past an airfoil.	PERFECT	2370	38
HYDRO2D	Navier Stokes solver to calculate galactical jets.	SPEC	4292	1474
MDG	Molecular dynamics model for water molecules.	PERFECT	1430	178
OCEAN	2D solver for Boussinesq fluid layer.	PERFECT	3288	118
SU2COR	Quantum mechanics with Monte Carlo simulation.	SPEC	2332	779
SWIM	Finite difference solver of shallow water equations.	SPEC	429	1106
TFFT2	Collection of FFT routines from NASA codes.	SPEC	642	946
TOMCATV	Generates 2D meshes around geometric domains.	SPEC	190	1327
TRFD	Kernel for quantum mechanics calculations.	PERFECT	580	20
WAVE5	Solves particle and Maxwell's equations.	SPEC	7764	788

Table 1: Benchmark codes studied

To evaluate the effectiveness of the six techniques implemented in Polaris, we transformed each program in Table 1 six times, with one technique either turned off or weakened in turn. In the case of Autoinlining and Interprocedural Value Propagation, the techniques were turned off completely. In the case of Privatization, Dependence Analysis, Induction Variable Recognition, and Reduction Recognition, only the advanced components of the techniques were turned off. Thus, for the Dependence Analysis measurement, the Equality and GCD Tests were applied, but not the Range Test. For the Reduction Recognition measurements, only scalar reduction variables were considered. For Privatization, only private scalar recognition was applied. For the Induction Recognition measurement, only induction variables with linear closed forms were taken into account.

The parallel program resulting in each case was then run on the same machine and configuration used for the measurements of Figure 1. The relative importance of each technique for each program can then be determined by comparing the speedups obtained in each experiment with the speedups shown in Figure 1, obtained by applying

all techniques in Polaris.

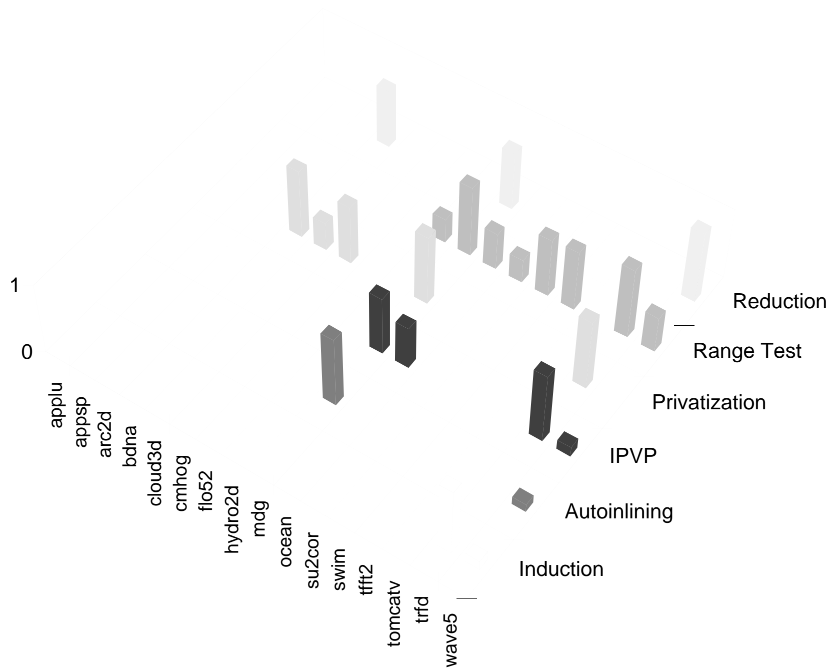

Figure 2: Performance Impact of Disabling Advanced Techniques

Figure 2 presents the results of the six experiments conducted for each code. The height of the bar at the intersection (P, T) is the value $1 - S_{P,T}/S_P$, where $S_{P,T}$ is the speedup obtained for program P by turning off or weakening techniques T, and S_P is the speedup obtained by applying all six techniques. Notice that the larger the value of $1 - S_{P,T}/S_P$, the greater the effect of technique T on the speedup of program P obtained by Polaris.

Figure 3, although similar to Figure 2, differs in that the height of the bar at (P, T) represents, in logarithmic scale, the percentage of the total number of loops in program P parallelized by technique T. From Figure 3, it can be seen that all techniques enable parallelization of loops from many programs in the collection. Although the inspiration for the techniques came from analyzing programs in the Perfect Benchmarks, they seem to be widely useful. As can be seen by comparing the figures, however, a technique may affect several loops in a program but not its overall execution time.

There are two main reasons for this. In some cases the execution time of the loop is small enough to have

little effect on the overall sequential execution time. However, as other loops of the program of which they are part become parallelized and the total execution time decreases as a consequence, their relative importance can increase substantially (i.e. they become the sequential bottleneck). In other cases, as occurs in `appsp`, the loops are nested within other parallel loops, and their parallelization is masked by these other loops [1].

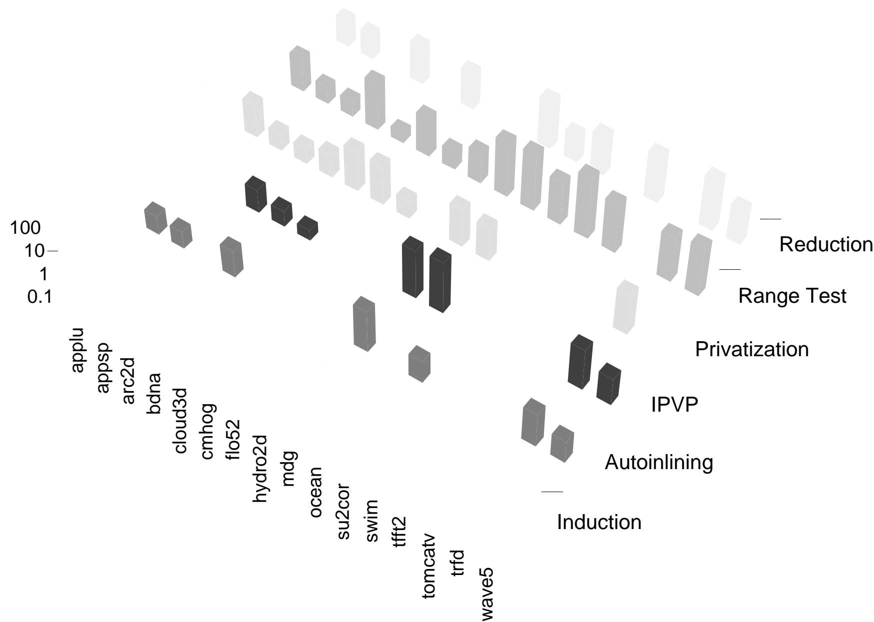

Figure 3: Impact on the Percentage of Parallelized Loops Disabling Advanced Techniques

4 New Techniques for the Detection of All Available Parallelism

In the previous section we discussed the contributions of the techniques currently implemented in Polaris to the detection of parallelism in a collection of sixteen programs. In order to measure the overall success of the Polaris compiler, we have performed a hand-analysis of the codes and extracted all those loops that are in fact parallel but could not be detected as such by Polaris. In very general terms, we regard these loops (weighted by their importance) as the measure of imperfection of the Polaris compiler. It shows how far we are from completing our

[1] on an architecture such as the SGI, which supports only single-level parallelism in the back-end Fortran compiler

task of automatically parallelizing the class of scientific applications represented by our benchmark collection. In order to achieve this goal, we have tabulated the new or improved compiler techniques that would be necessary to fully transform the codes for parallel execution.

In the next two sections, we will elaborate on these new techniques and on their relative importance for the overall parallelization of the codes analyzed, and give our opinion on how difficult they would be to implement.

4.1 Improving Current Techniques

From our hand analysis of parallelizable loops, we can conclude that the four main techniques in Polaris (Dependence Analysis, Privatization, Reduction Recognition and Induction Recognition) need improved implementation and a wider scope. In the following paragraphs, we list the improvements that we believe are necessary for each of the four techniques, give examples of specific loops where they would apply, and present their relative importance across the whole range of benchmarks that we have analyzed. Note here that the importance of the loops listed in Table 2 is based on their corresponding fraction of total program sequential execution time. In fact, that understates their actual importance because, in many cases, these loops represent a much higher fraction of the parallel execution time.

4.1.1 Privatization

Although we have been able to privatize almost all cases encountered within procedures, there is a need to generalize the conditional assignment and use of array elements. In the current implementation of the Polaris privatizer, not all statically available range information is used. For example, in the loop (CLOUD3D, KESSLER, DO_1000) [2], a complex interaction between an `if` statement and the loop limits causes the privatization coverage analysis to fail, thus leading to an overly conservative result. Table 2 gives more examples.

Most importantly, the scope analysis required for the detection of privatizable variables has to be extended interprocedurally. In the current version of Polaris we have tried to widen the scope of the analysis through inlining, but with limited success. We will discuss this issue in more detail in Section 4.2.7.

[2] We use the notation (program, routine, loop label) to identify loops throughout the paper

4.1.2 Reduction Parallelization

Reduction parallelization is performed in two stages. First, a reduction is recognized through pattern matching, and later on, substitute code for a parallel reduction is generated. In essence, this is a simple algorithm substitution. Our current implementation can match only simple forms of reductions. Through our hand analysis of the benchmark set, we have found many min-max reductions too difficult for Polaris to detect. This supports our conclusion that more advanced methods for reduction pattern recognition are necessary to handle the many equivalent ways programmers use reduction variables. Table 2 shows that some programs spend a significant amount of execution time performing reductions.

4.1.3 Induction Variable Substitution

Induction variable substitution is a simple but crucial transformation for loops containing induction variables. Its current implementation in the Polaris compiler covers several cases in which a closed form solution of the induction variable exists. Where we see the need for improvement are cases in which the sequence of values taken by the induction variable is either discontinous or not strictly monotonic. Such cases can occur, for example, when an induction variable is conditionally incremented or conditionally reassigned an initial value inside a loop. In Table 2 we point to some loops displaying this kind of behavior. One possible solution for this problem is to extract (distribute) from the original loop a simple loop that computes all the values of the induction variable (which can be easily parallelized using a parallel prefix algorithm) and then passes them on to the remainder loop.

4.1.4 Symbolic Analysis

Symbolic algebra has proven to be crucial for practically all the techniques in Polaris. For example, comparing symbolic expressions when computing memory access ranges and verifying conditions has been essential for the Range Test and for array privatization. We believe that progress can be made in two directions: by improving the power of symbolic analysis itself, and by postponing difficult or statically unknown cases to run-time evaluation. If an expression cannot be completely analyzed statically, then the evaluation and simplification can begin at compile time and end at run-time.

Since this type of hybrid analysis will enable parallelizing transformations, we believe that its small overhead will be covered by the cases in which it leads to large payoffs. Currently, when we cannot evaluate the value of a crucial variable, we generate a two version loop (one parallel, one sequential) guarded by an `if` statement. Almost all current techniques, especially those involved in dependence analysis, would benefit from such a hybrid scheme.

4.2 New Techniques

In this section, we first discuss how some of the already presented techniques could be generalized to recognize more independent access pattern cases. Then, we present some simple but useful transformations. And, finally, we introduce some experimental techniques.

4.2.1 Enhancing Dependence Analysis

In our hand analysis of the benchmark set we encountered several memory access patterns that are characteristic of certain algorithms that could not be successfully analyzed by conventional data dependence techniques. As previously mentioned, the Range Test can detect certain types of independent access patterns by comparing the range of memory references. It could be made more accurate by the use of more powerful symbolic analysis. Although we use Interprocedural Value Propagation, more exhaustive interprocedural range collection would be required to widen the scope of its effectiveness.

An example of a case that the Range Test was not designed to detect are the loops in OCEAN that perform an FFT. The access pattern of loop FTRVMT_DO_106 displays an interleaved access pattern with a stride that increases by powers of 2. It is completely independent, but the Range Test could not detect this. We believe that a library of dependence tests designed to check for a variety of patterns (similar to the Range Test) could be developed. Within such a generalized approach, pattern matching techniques would play a significant role. Table 2 lists several loops with interleaved access patterns that are not analyzed accurately by the Range Test.

4.2.2 Vector Operations with Overlapping Ranges

Loops equivalent to Fortran 90 vector operations with overlapping ranges are present in many programs we've analyzed, but Polaris detects a cross-iteration dependence for such loops and generates sequential code for them. For example, a loop equivalent to: $A(1:n) = const * A(2:n+1)$ could be easily parallelized by introducing a temporary private array. We believe that it will be a relatively easy task to recognize this type of dependence pattern and generate parallel code. For certain target machines, Fortran 90 instructions may be generated because their native compiler can produce well optimized code for them. In Table 2, we show some of the more important loops that are equivalent to Fortran 90 vector operations with overlapping ranges.

4.2.3 Multiple Exit Loops

As Table 2 shows, loops that might exit prematurely but are otherwise independent can be a significant source of parallelism. A naive parallelization of such a loop could execute too many iterations, thereby modifying data that would not have been changed in a sequential execution. We believe that the problem can be solved in the near future by implementing some of the ideas presented in [9]. Briefly stated, we provide a mechanism through which to undo the modifications of unwanted iterations. We should note that in most of the cases we've analyzed, the abnormal exit condition represents an error guard, meaning that early loop exit is unlikely to occur and that the associated overhead is acceptable.

4.2.4 Parallelizing I/O

Although `DO` loops that perform I/O are quite numerous, we find that most of them are not significant in our compute-intensive benchmarks. In principle, there is no major new technique that would be capable of parallelizing such loops. One approach to solve the problem is to use application level storage and buffering and hoist and/or push I/O operations out of the loop, leaving the remainder available for parallel execution. However, some systems already support I/O within parallel loops. In Table 2, we list some of the loops that perform I/O. We should note that the sequential time reported for them includes the actual I/O activity.

4.2.5 Associative Recurrences

So far, we have discussed how to handle fully parallel loops that we could not transform for concurrent execution. As we can see from Table 2, there are a significant number of loops that include associative linear recurrences that can be parallelized effectively. In order to exploit this parallelism we need to detect the typical dependence pattern of such a recurrence and verify that the operation is associative. We can then apply techniques such as loop distribution and the use of parallel prefix. We believe that this transformation will be the next type of algorithm substitution widely performed by compilers.

4.2.6 Run-Time Techniques

In Table 2, we have a few examples of loops with arrays whose access patterns could not be analyzed because they are indirectly indexed (i.e., they use subscripted subscripts). In the general case, because the necessary information is not statically available, we can make the assumption that no compile time technique will be able to analyze such loops. Furthermore, we believe that this type of access pattern is prevalent in other applications (not introduced in this article) such as modern, sparse simulations and integer programs.

In [7] and [8] we have offered a solution for both the case just described, and for any case in which classical, static compile-time methods fail. In brief, we propose an analysis of the dependence pattern at run-time, when all necessary values become known. We employ a fully parallel algorithm that can either "inspect" the access pattern first, or assume full parallelism and execute speculatively, verifying the correctness of the execution after loop termination. We have obtained very good initial experimental results and intend to pursue this avenue further.

4.2.7 Inline Expansion vs. Interprocedural Analysis

As can be seen in 4, interprocedural analysis is an important technique. As previously shown, we have partially solved this problem through expanding procedure calls based on a somewhat crude heuristic. We were able to use this technique only in the cases where the number of resulting statements within the scope of our analysis did not grow over a certain threshold. As this number of statements increased, program analysis became impossible or impractical to handle. We think that even if expansion were to be demand driven, it would not become a scalable

technique capable of resolving problems in very large programs at a global level. We have also observed that the array linearization performed by expansion either removes array shape information or complicates the subscript expressions enough to prohibit other types of analysis. Table 2 shows important examples of loops that could not be parallelized due to the absence of interprocedural analysis.

In the near future we hope to develop a global interprocedural analysis strategy that will allow us to extract and use interprocedural information on demand.

4.3 A Global Picture

Figure 4 is a first approximation to a global picture of the future needs of parallelizing compiler development. Each of the previously discussed new techniques are depicted as a function of their overall weight (fraction of the total sequential execution time) for all the applications we have studied. In this picture we observe that there are some techniques that would contribute to the parallelization of many programs, while others are important to fewer benchmarks. Although we can detect some clear trends (e.g., that privatization and interprocedural analysis are important for a significant fraction of the programs), each application has specific access patterns requiring specific techniques to detect its parallelism (e.g., OCEAN needs a more advanced data dependence analysis method to detect that its interleaved memory accesses are independent). It also suggests that an across the board effort to develop more compiler algorithms is needed to parallelize sequential programs uniformly. Our conclusions about the relative importance of compiler techniques should be hedged by the fact that most of the benchmarks analyzed are of the 'regular' type and that methods needed to parallelize sparse, dynamic codes will be much more complex, perhaps less efficient and, with the exception of the run-time methods, quite different from those discussed here. On the other hand, we also believe that the techniques this picture presents for regular problems will form a foundation on which to develop techniques for parallelizing irregular problems, which represent a majority of scientific computation.

Finally, much more needs to be done in the area of parallelism exploitation (e.g., data distribution, scheduling and efficient code generation), an area that has not been discussed in this paper.

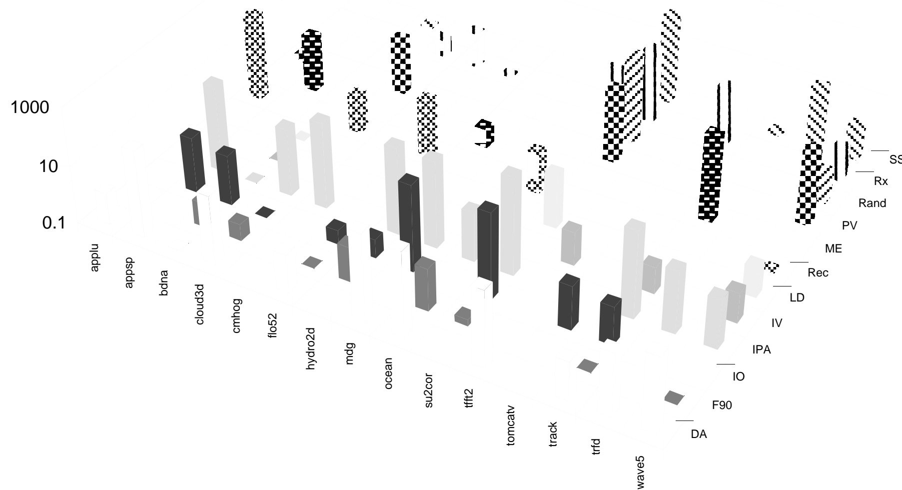

Figure 4: An overview of the requirements for new techniques in several benchmark programs and their relative importance to their full parallelization.

5 Conclusions

The techniques implemented in the first version of Polaris have proven to be quite effective on a variety of programs. From the previous sections it becomes clear that there is no silver bullet, and that a wide range of techniques is necessary to detect as much parallelism as is present in real programs.

Perhaps the most important achievement of our work in the Polaris project has been the demonstration that substantial progress in compiling conventional languages is possible. There have been, and still are, many who do not believe it is possible to develop compilers capable of generating effective parallel code for a wide range of real programs. We disagree and believe that, at least for Fortran and other similar languages such as MATLAB, effective translators will be available within the next decade. Our experimental results and careful hand analysis of real codes strongly support this opinion.

References

[1] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Paralleliza-

tion. *Proceedings of the IEEE*, 81(2), February 1993.

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

[3] William Joseph Blume. *Symbolic Analysis Techniques for Effective Automatice Parallelization*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., June 1995.

[4] 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.

[5] John Robert Grout. Inline Expansion for the Polaris Research Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., May 1995.

[6] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, July 1995.

[7] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. Run-Time Methods for Parallelizing Partially Parallel Loops. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, July 1995.

[8] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[9] Lawrence Rauchwerger and David A. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. *Proceedings for the 9th International Parallel Processing Symposium*, April 1995.

[10] Peng Tu and David Padua. Automatic Array Privatization. In Utpal BanerjeeDavid GelernterAlex Nicolau-David Padua, editor, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.

[11] Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 414 – 423, July 1995.

Technique	Program	Subroutine	Loop	% T_{seq}	Nesting level	P/PP
DA	WAVE5	PARMVR	do50	10.56	1	P
	TFFT2	CFFTZ0	do#1	16.20	2	P
	CLOUD3D	CLOUD3D	do#5	21.42	2	P
	OCEAN	FTRVMT	do106	11.96	2	P
	OCEAN	FTRVMT	do1060	11.96	1	P
	OCEAN	FTRVMT	do#2	11.96	1	P
	OCEAN	FTRVMT	do#4	11.98	1	P
	OCEAN	FTRVMT	do103	11.99	1	P
F90	HYDRO2D	ISMIN	do10	0.76	1	P
	OCEAN	OCEAN	do500	1.25	1	P
	OCEAN	OCEAN	do470	1.67	2	P
IO	APPSP	TXINVR	do#1#1#1	2.78	4	P
	APPSP	TXINVR	do#1#1	2.78	3	P
	BDNA	RESTAR	do15	4.93	1	P
IPA	CLOUD3D	CLOUD3D	do#5	21.42	2	P
	SU2COR	LOOPS	do400	29.65	3	P
	TRACK	EXTEND	do400	32.53	2	P
	TRACK	EXTEND	do300	32.67	3	P
	SU2COR	SWEEP	do220	34.37	4	P
IV	WAVE5	INIT	do170	1.103	1	PP
	TFFT2	RANDP	do120	1.43	1	PP
LD	SU2COR	LOOPS	do300/2	3.36	4	P
	SU2COR	LOOPS	do300	3.37	3	P
Rec	APPLU	BLTS	do#1#1#1#3	5.23	5	PP
	APPLU	BLTS	do#1#1#1#1	12.42	5	PP
	APPLU	BLTS	do#1	23.38	2	PP
ME	TRACK	FPTRAK	do300	8.74	2	P
	TRACK	EXTEND	do400	32.53	2	P
	TRACK	EXTEND	do300	32.67	3	P
PV	WAVE5	PARMVR	do30	5.41	1	P
	CLOUD3D	KESSLER	do1000	7.99	2	P
	WAVE5	PARMVR	do50	10.56	1	P
	SU2COR	SWEEP	do220	34.37	4	P
Rx	TOMCATV	MAIN	do80	5.32	2	P
	SU2COR	SWEEP	do220	34.37	4	P
SS	SU2COR	LOOPS	do400/2	29.89	4	P
	SU2COR	LOOPS	do400/3	32.48	5	P
	SU2COR	LOOPS	do900	36.86	2	P,X

Table 2: Examples of loops whose parallelization requires new techniques. % T_{seq} shows the fraction of total sequential execution time taken by the loop; *Nesting Level* is the global nesting level of the loop starting at the outer level; *P/PP* shows whether the loop is fully parallel or partially parallel; *Technique* shows what new technique is needed to parallelize the loop. The following techniques are called for: *DA* – better dependence analysis; *F90* – Fortran 90 type constructs; *IO* – Input/Output parallelization; *IPA* – interprocedural analysis; *IV* – induction variable recognition and substitution; *LD* – loop distribution; *Rec* – recurrence parallelization; *ME* – multiple exit loop parallelization; *PV* – array privatization; *Rx* – reduction recognition and parallelization; *SS* – subscripted subscripts, run-time methods.