Restructuring Programs for High-Speed Computers with Polaris *

Bill Blume   Rudolf Eigenmann   Keith Faigin   John Grout   Jaejin Lee
Tom Lawrence   Jay Hoeflinger   David Padua   Yunheung Paek   Paul Petersen
Bill Pottenger   Lawrence Rauchwerger   Peng Tu
Stephen Weatherford

Center for Supercomputing Research and Development
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, Illinois 61801-2307
217-333-6223, fax: 217-244-1351
polaris@csrd.uiuc.edu

Abstract

The ability to automatically parallelize standard programming languages results in program portability across a wide range of machine architectures. It is the goal of the Polaris project to develop a new parallelizing compiler that overcomes limitations of current compilers. While current parallelizing compilers may succeed on small kernels, they often fail to extract any meaningful parallelism from whole applications. After a study of application codes, it was concluded that by adding a few new techniques to current compilers, automatic parallelization becomes feasible for a range of whole applications. The techniques needed are interprocedural analysis, scalar and array privatization, symbolic dependence analysis, and advanced induction and reduction recognition and elimination, along with run-time techniques to permit the parallelization of loops with unknown dependence relations.

1 Introduction

Supporting standard programming languages on any kind of computer system is and has been an important issue in computer science. This issue becomes particularly important on parallel machines where programming and language interfaces must address the more complex issue of orchestrating the joint operation of multiple processors. As machine structures evolve, users of such machines must repeatedly learn new machine-specific features and language elements. These issues make the discipline of supporting standard languages, and thus automatic program transformation, an interesting and important area of research.

For many years our research group has been working toward the goal of making parallel computing a practical technology. Parallelizing compilers have played an important role in this quest. The present project has its early roots in a compiler evaluation effort of the late '80s in which we determined that, despite their success on kernel benchmarks, available compilers were not very effective on whole programs [8, 5]. Based on these results we initiated an effort to hand-parallelize complete programs in order to identify effective program transformations [8, 7]. The need for a representative set of real programs for use in this evaluation was satisfied in part through the Perfect Benchmarks® effort, a joint benchmarking initiative of the University of Illinois and many other institutions [2].

As a result of these efforts we found that not only could real applications be parallelized effectively, but the transformations could also be automated in a parallelizing compiler. One task remained: we had not actually implemented these transformations and thus had not delivered the final proof that parallelizing compilers could be considerably improved.

To address this issue we implemented the prototype **Polaris** compiler and evaluated its effectiveness. Polaris is a source-to-source automatic restructurer which uses advanced analysis techniques to de-

*Research supported in part by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

tect parallelism and subsequently generates machine-specific output for a given target architecture. Polaris currently produces output for the Cray T3D and SGI Challenge multiprocessors.

The Polaris implementation includes a sizable basic infrastructure for manipulating Fortran programs, which we describe in Section 2. The analysis and transformation techniques implemented in Polaris are discussed in Section 3. Finally, in Section 4 we present an evaluation of Polaris.

2 Internal Organization

The aim in the design of Polaris' internal organization [9] was to create an internal representation (IR) that enforced correctness, was robust and, through high-level functionality, was easy to use.

Our view of the IR is that it is more than just the structure of the data within the compiler. We also view it as the operations associated with this data structure. Intelligent functionality can frequently go a long way toward replacing the need for complex data structures and is usually a more extensible approach. Thus, we chose to implement the data-portion of the IR in the traditional, straightforward form of an abstract syntax tree. On top of this simple structure, however, we built layers of functionality which allow the IR to emulate more complex forms and higher-level operations.

We chose to implement Polaris in the object-oriented language C++ as it both allowed us structural flexibility and gave us desired data-abstraction mechanisms. Operations built into the IR are defined such that the programmer is prevented from violating the structure or leaving it in an incorrect state at any point during a transformation. Transformations are never allowed to let the code enter a state that does not correspond to proper Fortran syntax. The system also guarantees that the control flow graph is consistent through automatic updates of this information as a transformation proceeds. The automatic consistency maintenance has decreased the time required to implement new optimizations in Polaris because potentially time-consuming error determination procedures are avoided.

Features that have been implemented in order to make the system robust and to maintain consistency include:

- Extensive error checking throughout the system through the liberal use of assertions. Within Polaris, if any condition or system state is assumed, that assumption is specified explicitly in a `p_assert()` (short for "Polaris assertion") statement which checks the assumed condition and reports an error at runtime if the assumption is incorrect.

- Detection of object deletion when that object is being referenced from another part of Polaris. If a deleted object is referenced, Polaris will abort with an internal consistency error. This, and associated problems, are avoided by the reference counting of all objects stored in Polaris collections.

- The detection of aliased structures (structure sharing is not allowed) causes a run-time error. For example, it would be an error to create a new expression and insert it into two different statements without first making a copy of it.

- A clear understanding of who is responsible for the deallocation of an object. The creator of an object is the initial "owner" of the object. By convention, ownership is transferred when a pointer to an object is passed as an argument to a method or function call. On the other hand, when a a C++ reference to an object is passed as an argument, ownership is not being transferred. The owner of an object is responsible for its destruction (deallocation). The use of this convention simplifies the programming interface between the basic Polaris infrastructure and passes which are built upon the infrastructure.

We followed the usual object-oriented approach in our implementation in that classes are used to represent the various program structures. These include programs, program units, statements, statement lists, expressions, symbols and symbol tables as well as a complete set of support structures which include parameterized container and iterator classes. Each class provides extensive high-level functionality in the form of member functions used to manipulate the class instances.

Much of the implementation is intuitive and straightforward. The **Program** class, for instance, is a collection of **ProgramUnit**s. Among the included member functions are routines for reading and displaying complete Fortran codes. There are also member functions for adding additional **ProgramUnit**s to a **Program** as well as merging **Program**s.

The **ProgramUnit** class is, similarly, a container for the various data structure elements that make up a Fortran program unit including statements, a symbol table, common blocks, and equivalences.

Statements are simple, non-recursive structures kept in a **StmtList** object. Many high-level member functions are implemented within the **StmtList** class in order to provide the programmer with a powerful and easy-to-use environment. Examples include routines that return an iterator over selected parts of the statement list (such as the body of a `do` loop or all statements of a specific type), as well as member functions which copy, insert, delete, or move any well-formed sublist of statements or specific multi-block statement groups (such as a block-`if` statement framework or a `do-enddo` group).

To maintain complete control of consistency inside the **StmtList** class, the manipulation of statements or lists of statements is restricted by checks during the execution of Polaris. For example, the block to be processed must be entirely well-formed with regard to multi-block statements such as `do` loops and block-`if` statements. Another example is the restriction that deleting a block containing a statement which is referenced from outside the block being deleted is flagged as a run-time error during Polaris' execution.

The programmer can defer this consistency management by using a **List<Statement>** for the modification. In this way, a section of code can be manipulated that is inconsistent during its creation and modification but is checked for consistency when it is incorporated into the program. These safeguards create an environment where traditionally time-consuming errors are automatically recognized and can be more quickly identified.

The data fields declared in the base **Statement** class (and which, as a result, exist in all statements) include sets of successor and predecessor flow links, sets of memory references, and an **outer** link that points to the innermost enclosing `do` loop. Whenever practical we implemented the member functions such that any modification to a statement results in the updating of affected data in order to retain consistency.

Each derived statement class may declare additional fields. The **DoStmt**, for example, declares a **follow** field which points to its corresponding **EndDoStmt** as well as fields for the index of the loop and the init, limit, and step expressions. Each statement class also declares a number of member functions which include, for example, a routine that returns an iterator which traverses all of the expressions contained in the statement. Along with similar member functions in the **Expression** class, this makes it easy, for instance, to traverse all the expressions in a loop body during dependence analysis.

Expressions and symbols are implemented in much the same way as statements in that an abstract base class declares structures common to all elements and specific classes are derived from the base. The base **Expression** class includes, for instance, member functions for such operations as retrieving type and rank information, simplification and structural-equality comparison. Polaris has very powerful routines to test the structural-equality of expressions, as well as pattern-matching and replacement routines. These are based on an abstract **Wildcard** class, which is derived from **Expression**. To perform pattern matching, one simply creates a pattern expression (an expression that may contain wildcards anywhere in the tree) and compares this pattern to an expression using the equality matching member function. These functions have proven to be powerful and general and form the basis for a higher-level tool for pattern matching and replacement named "Forbol" [19].

3 Analysis Techniques

In the following sections we will discuss many of the techniques that have been built into the current version of Polaris.

The first is the method of interprocedural analysis used for this stage of Polaris' development. We have chosen to implement inline expansion for several reasons: (1) it provides us with the most information possible, (2) it allows existing intraprocedural techniques to be used, (3) it allows the calling overhead of small routines to be eliminated. However, it is well known that compile-times grow inordinately large for fully-inlined programs, and we are currently implementing a comprehensive interprocedural analysis framework as well.

The second technique is the recognition and removal of inductions and reductions. Generalized forms of these recurrences have been found to have an impact on the performance of the Perfect Benchmarks.

The third technique is symbolic dependence analysis for the recognition of parallelism. Traditionally, dependence analysis has been numerical in nature. By exploiting the ability to reason about symbolic expressions we have shown that many previously intractable cases can now be automatically analyzed.

The fourth technique is scalar and array privatization. This is one of the fundamental enabling techniques. Through advanced flow-sensitive analysis we can determine when arrays and even array sections can be replicated to reduce the storage that must be

shared among the processors.

The final technique discussed in this paper is a run-time method for finding and exploiting parallelism. Even with the advanced symbolic analysis techniques and transformations that we have implemented, we find that sometimes the control flow of a region or the data dependence patterns are a function of the program input data. For these cases we are developing run-time methods for the recognition and implementation of parallelism. These techniques are based on speculative execution, and are currently being implemented in Polaris [16].

3.1 Inline Expansion

The Polaris inliner is designed to provide three types of services: automatic inline expansion of certain subprograms for analysis, selective inline expansion of subprograms for code generation, and selective modification of subprograms.

Interprocedural analysis is a requirement for effective automatic parallelization. In Polaris, we chose to use automatic inline expansion of certain subprograms to aid full flow-sensitive interprocedural analysis, which is especially important for privatization. Driver routines are provided by Polaris to automatically mark certain call sites for inline expansion, and to either expand at all call sites at which inline expansion has been requested, or to expand at all call sites at which inline expansion was not forbidden.

For complete inline expansion, the inliner driver is passed a collection of compilation units: one is designated as the top-level program unit. The driver repeatedly expands subroutine and function calls in the top-level program unit. This allows the inliner to handle complex argument redimensioning and retyping by generating equivalences.

To minimize the cost of complete inline expansion, the transformations to inline a given subprogram into a "top-level" routine (one which will remain after all inline expansions have been performed) are split into site-independent and site-specific sets.

The first time each subprogram is to be expanded into a "top-level" routine the inliner creates a corresponding "template" object and performs all site-independent transformations (e.g., variable renaming) on it. Then, when an individual "top-level" call site is to be expanded, the inliner copies the corresponding "template" object into a "work" object, performs site-specific transformations (e.g., formal to actual remapping) on its source, then moves that transformed source into the "top-level" routine at the call site.

In most cases, the inliner can map formal arrays directly into the corresponding actual array in the top-level program. Occasionally, a formal array must be mapped into an equivalent, linearized version of the actual array. In practice, the range test (Section 3.3) has been able to overcome the potential loss of dependence accuracy caused by linearization in most cases.

After the analysis of a program using automatic inline expansion, the template objects created during analysis can be used as a base for performing selective inline expansion of subprograms for code generation. The information derived from performing this analysis can also be used to guide selective modification of subprograms using such techniques as cloning, loop embedding, and loop extraction [10].

All of the programs that we have tested were inlined successfully by Polaris. Some constructs are not easily expressible in Fortran after inline expansion. The constructs which are not fully supported involve the need for expressing an equivalence between non-conforming formal and actual parameters. A typical case (passing a REAL actual array to a COMPLEX formal array, an example that occurs in the Perfect Benchmarks) is handled automatically, but requires the favorable assumption that the actual array references are on double-word boundaries.

3.2 Induction Variable Substitution and Reduction Recognition

Induction and reduction variables form *recurrences*, which inhibit the parallel execution of the enclosing loop. Each loop iteration computes the value of the variable based on the value assigned in the previous iteration. In data dependence terms, this forms a cycle in the dependence graph, which serializes the loop.

Reduction variables most often accumulate values computed in each loop iteration, typically of the form `sum = sum + <expression>`. Because the "+" operation is associative, partial sums can be accumulated on parallel processors and summed at the end of the loop. Due to the limited precision of Fortran variables, this transformation may introduce numerically different results, and therefore in Polaris, as in most compilers, the user has the option of disabling this transformation. However, we have not found this to be a problem in our benchmark suite.

Polaris uses a directive to flag potential reductions of the form:

$$A(\alpha_1,\alpha_2,\ldots,\alpha_n) = A(\alpha_1,\alpha_2,\ldots,\alpha_n) + \beta$$

where α_i and β are expressions that do not contain references to A, A is not referenced elsewhere in the

loop (outside of other reduction statements), and n may be zero (i.e., A is a scalar variable). Polaris initially recognizes candidate reductions of this form using the Wildcard class discussed in Section 2. After flagging candidate reduction statements, the data-dependence pass later analyzes and removes the flags for those statements which it can prove have no loop-carried dependences.

More than one reduction statement can occur in a loop and they may sum into different array elements in different loop iterations. Reductions which sum into a single array element or into a scalar are termed **single address reductions**. Those which sum into different array elements are termed **histogram reductions**. The Polaris compiler recognizes reductions of both types and solves them in one of three forms: blocked, private, or expanded [14].

Induction variables form arithmetic and geometric progressions which can be expressed as functions of the indices of enclosing loops. A simple case is the statement `K=K+1`, which can be deleted after replacing all occurrences of `K` with the initial value of `K` plus the trip count as a function of the loop index.

Current compilers are able to handle induction statements with loop invariant right-hand-sides in multiply nested "rectangular" loops. In our manual analysis of programs we have found two additional important cases: one, when induction variables are a function of other induction variables (we term these *cascaded* induction variables), and two, when induction variables occur within *triangular* loop nests.

The following example shows a triangular loop nest that contains a cascaded induction variable. Polaris transforms this code into the parallelizable form shown (in Section 3.3 we will show how our dependence test investigates these non-linear subscript expressions). Notice that the use of cascaded induction variables (`K1` and `K2` with initial values 0) cause an unusually large code expansion. In order to address the need for strength reduction in cases like this we are implementing a scheme which assigns initial closed-form values to private copies of induction variables at each parallel loop header, leaving uses in the remainder of the loop body in their original (i.e., unsubstituted) form.

The Polaris induction variable substitution algorithm performs three steps in order to recognize and substitute induction variables (the following considers only additive induction variables; however multiplicative inductions are solved as well [13]):

1. Locate candidate induction statements by recognizing recurrence patterns of scalar variables which are incremented by either an enclosing loop index, an expression containing other candidate induction variables, or any loop-invariant expression.

2. Compute the closed form of the induction variable at the beginning of each loop iteration (and the last value at the end of the loop) as functions of the enclosing loop indices. The total increment incurred by the induction variable in a loop body is first determined, and then this expression is summed across the iteration space of the enclosing loop. If an inner loop is encountered while computing this increment, the algorithm recursively descends into the inner loop and first computes the closed form of the induction variable for the inner loop.

3. Substitute all occurrences of the induction variables. The substituted value is the closed form expression for the induction variable at the loop header plus any increments encountered up to the point of use in the loop body.

```
do I = 1,N               do I = 1, N
  do J = 1,I               do J = 1, I
    K1 = K1 + 1              X((I^4 - 2I^3 + 3I^2 -
    K2 = K2 + K1     =>        2I + (4I^2 - 4I)J +
    X(K2) =...                 4J^2 + 4J)/8) = ...
  end do                   end do
end do                   end do
```

Figure 1: Substitution of cascaded inductions

3.3 Symbolic Dependence Analysis

Data dependence analysis is crucial to determine what statements or loops can be safely executed in parallel. Two statement instances are data dependent if they both access the same memory location and at least one of these accesses is a write. If two statements do not have a chain of dependence relations connecting them, then they can be executed in parallel. Also, a loop can be executed in parallel, without the need for synchronization between iterations if there are no dependences between statement instances in different iterations.

There has been much research in the area of data dependence analysis. Because of this, modern day data dependence tests have become very accurate and efficient [12]. However, most of these tests require the loop bounds and array subscripts to be represented as

a linear (affine) function of loop index variables; that is, the expressions must be in the form $c_0 + \sum_{j=1}^{n} c_j i_j$ where c_j are integer constants and i_j are loop index variables. Expressions not of this form are called *nonlinear* (e.g., they have a term of the form $n * i$ where n is unknown). Techniques have been developed to transform nonlinear expressions into linear ones (e.g., constant propagation and induction variable substitution), but they are not always successful.

In our experience with the Perfect Benchmarks, such nonlinear expressions do occur in practice. In fact, four of the twelve codes (i.e., DYFESM, QCD, OCEAN, and TRFD) that we hand-parallelized would exhibit a speedup of at most two if we could not parallelize loops with nonlinear array subscripts [4]. For some of these loops, nonlinear expressions occurred in the original program text. For other loops, nonlinear expressions were introduced by the compiler. The two most common compiler passes that can introduce nonlinearities into array subscript expressions are induction variable substitution and array linearization. An example of how induction variable substitution can introduce nonlinear array subscripts is shown in Figure 2. This loop nest, a simplified version of the nest OLDA/100 in TRFD, accounts for about 70% of the code's sequential execution time.

```
X0 = 0
do I = 0, M-1                 do I = 0, M-1
 X = X0
 do J = 0, N-1                 do J = 0, N-1
  do K = 0, J-1                 do K = 0, J-1
   X = X+1
   A(X) = ···           ⇒        A(K + 1 + (I(N² + N)
  end do                            +J² − J)/2) = ···
 end do                         end do
 X0 = X0 + (N² + N)/2          end do
end do                        end do
```

Figure 2: Induction substitution in TRFD

3.3.1 Range Test

To handle such nonlinear expressions, we have developed a symbolic dependence test called the *range test* [6]. The range test can be viewed as an extension of a symbolic version of Triangular Banerjee's Inequalities test with direction vectors [1, 20]. In the range test, we mark a loop as parallel if we can prove that the range of elements accessed by an iteration of that loop does not overlap with the range of elements accessed by other iterations. We determine whether these ranges overlap by comparing the minimum and maximum values of these ranges. To maximize the number of loops found parallel using the range test, we symbolically permute the visitation order of the loops in a loop nest when computing their ranges.

The computation of the minimum and maximum values of a symbolic array access expression can be quite involved. For example, the maximum value of the expression $f(i) = n * i$ for any value of i, where $a \le i \le b$, can be either $n * a$ or $n * b$, depending upon the sign of the value of n. So, to compute the minimum or maximum of an expression for a variable i, the range test first attempts to prove that the expression is either monotonically non-decreasing or monotonically non-increasing for i. The monotonicity of an expression f is determined by computing the forward difference of the expression $(f(i+1) - f(i))$, then testing whether this expression is greater than or equal to zero, or less than or equal to zero. If $a \le i \le b$, then the maximum of expression $f(i)$ for any legal value of i is $f(b)$ if it is monotonically non-decreasing for i, $f(a)$ if it is monotonically non-increasing for i, and undefined otherwise. The computation of the minimum is similar.

As an example, we will show how to compute the minimum and maximum values of the subscript expression of array A for a fixed iteration of the outermost loop of the loop nest shown in Figure 2. Let $f(i,j,k) = (i*(n^2+n)+j^2-j)/2+k+1$ be the subscript expression for array A. To compute the minimum and maximum values of f for any legal value of the inner pair of loops, we will first determine the minimum and maximum values of f for the innermost loop, which has index k, then determine the minimum and maximum values that the middle loop, which has index j, can take for these minimum and maximum values. Since the forward difference for index k is positive (i.e., $f(i,j,k+1) - f(i,j,k) = 1$), f is monotonically non-decreasing for k. Thus, the maximum value (a_1) that f can take for any value of k is $a_1(i,j) = f(i,j,j-1) = (i*(n^2+n)+j^2-j)/2+j$. Similarly, the minimum value (b_1) of f for index k is $b_1(i,j) = f(i,j,0) = (i*(n^2+n)+j^2-j)/2+1$. For the next loop, the index j is monotonically non-decreasing for both a_1 and b_1, since $a_1(i,j+1) - a_1(i,j) = j+1 > 0$ and $b_1(i,j+1) - b_1(i,j) = j \ge 0$. So the maximum value ($a_2$) that f can take for any legal value of indices j and k is $a_2(i) = a_1(i,n-1) = (i*(n^2+n)+n^2-n)/2$ and the minimum value (b_2) is $b_2(i) = b_1(i,0) = (i*(n^2+n))/2+1$.

By comparing these minimum and maximum values of array accesses, we can prove that there are no loop-carried dependences between these accesses. For example, there cannot be a loop-carried dependence

from $A(f)$ to $A(g)$ for a loop with index i if the maximum value accessed by the jth iteration of index i for f is less than the minimum value accessed by the $(j+1)$th iteration of i for g, and if this minimum value of g is monotonically non-decreasing for i. See [6] for other tests that use these minimum and maximum values of f and g.

Returning to the example for Figure 2, we will now apply the dependence test described above to prove that $A(f)$ does not carry any dependences for the outermost loop. From the previous example, we know that the maximum value that f can take for any legal value of indices j and k is $a_2(i) = (i*(n^2+n)+n^2-n)/2$ and the minimum value is $b_2(i) = (i*(n^2+n))/2+1$. By the definition of the dependence test given above, if we can prove that $a_2(i) < b_2(i+1)$ and b_2 is monotonically non-decreasing for i, $A(f)$ cannot carry any dependences for the outermost loop. Since $b_2(i+1) - a_2(i) = n+1 > 0$, $a_2(i)$ must be less than $b_2(i+1)$. Also, b_2 is monotonically non-decreasing because $a_2(i+1) - a_2(i) = n^2+n > 0$. Therefore, there are no carried dependences for the outermost loop and it can be executed in parallel. The same dependence test can be used to prove that the other loops from Figure 2 also do not carry dependences.

As the previous examples have shown, the range test requires the capability to compare symbolic expressions. For example, we needed to test whether $j > 0$ or $n^2+n > 0$ in the previous examples. To provide such a capability, we have developed an algorithm called *range propagation*. Range propagation consists of the determination of symbolic lower and upper bounds, called ranges, for each variable at each point of the program. The next subsection will describe an efficient way in which these variable ranges may be computed from the program's control flow. Expression comparison using ranges is done by computing the sign of the minimum and maximum of the difference of the two expressions, using techniques similar to those described earlier.

A more complicated example is shown in Figure 3. This loop nest accounts for 44% of OCEAN's sequential execution time. (Interprocedural constant propagation and loop normalization were needed to transform the loop nest into the form shown.) Current data dependence tests would not be able to parallelize any of the loops in the nest because of the nonlinear term $258*x*j$. The range test can prove all three loops as parallel. However, for it to do so, it must apply its tests on a temporary permutation of the loop nest, so that the outermost loop is swapped with the middle loop. This is necessary since the middle loop has a larger stride $(258*x)$ than the stride of the outermost loop (129). This causes an interleaving of the range of accesses performed by two distinct iterations of the outermost loop. By swapping the middle and outermost loops, the interleaving is eliminated, allowing all three loops to be identified as parallel.

```
do K = 0, X-1
  do J = 0, Z(K)
    do I = 0, 128
      A(258XJ + 129K + I + 1) = ···
      A(258XJ + 129K + I + 1 + 129X) = ···
    end do
  end do
end do
```

Figure 3: Simplified loop nest FTRVMT/109

The symbolic capabilities of the range test permit it to handle many of the symbolic expressions we have seen in the Perfect Benchmarks. Most current data dependence tests cannot handle symbolic subscripts [4]. The only drawback of our test (compared to the Triangular Banerjee's test with directions), is that it cannot test arbitrary direction vectors, particularly those containing more than one '<' or '>' (e.g., $(<,<)$). The permutation of loop indices partially overcomes this drawback. (These permutations can be thought of as permutations of the dependence direction vectors tested.) We have found that this limited set of direction vectors, along with the permutation of loop indices, was sufficient to parallelize all of the relevant loop nests in our test suite. An advantage of the range test is that the worst case of the number of direction vectors tested is better than Banerjee's Inequalities with directions, since we test at most $O(n^2)$ direction vectors while Banerjee's Inequalities with directions may test as many as $O(3^n)$ direction vectors.

3.4 Scalar and Array Privatization

Although symbolic dependence analysis will allow us to prove that more references in a loop nest are independent from each other, it will not allow a significantly greater number of important loops to be parallelized without additional transformations. In our experience, the most important of these transformations is *array privatization* [17].

Array privatization is used to eliminate memory-related dependences. It identifies scalars and arrays that are used as temporary work spaces by a loop iteration, and allocates a local copy of those scalars and arrays for that iteration.

To prove that a variable is privatizable, every use of that variable must be dominated by a definition of the variable in the same loop iteration. Determining the dominating definition for a scalar variable is straightforward, since a scalar is an atomic object that can only be read and written as a whole. However, since an array variable is a composite object that can be partially read and written, determining whether an array assignment dominates an array use requires an elaborate analysis of the array ranges. More specifically, the array privatizer must prove that the region of array elements referenced by the use is a subset of the region of array elements defined by the dominating assignment. Symbolic analysis techniques are often required for these region comparisons, since the regions often contain symbolic expressions.

In many cases, determining whether a region in a definition dominates a region in a use can be done using local information. However, in many other cases, it requires more elaborate symbolic analysis using global information.

```
S1 :   M = ...
       ...
S2 :   MP = M * P
       ...
       do I = 1, N
          do J = 1, MP
             A(J) = ...
          end do
          ...
          do K = 1, M
             do L = 1, P
                ...= A(M*(L-1)+K) ...
             end do
          end do
       end do
```

Figure 4: Example of Array Privatization

A simple example where such analysis is necessary for array privatization is shown in Figure 4. To parallelize the `I` loop, array `A` must be privatized. Loop `J` defines the region `A(1:MP)`, while loop `K` uses region `A(1:M*P)`. Thus, to prove that `A` is privatizable, we only need to prove that $\mathtt{MP} \geq \mathtt{M} * \mathtt{P}$. To prove this, we need to find out how the symbolic variables are related from their global *def-use* relations.

In Polaris, we use a demand-driven algorithm based on a Gated Static Single Assignment (GSA) representation to obtain global information [18]. In GSA form, the value of a symbolic variable is represented by a symbolic expression involving other symbolic variables, constants, and *gating functions*.

In the SSA representation, ϕ-functions of a single type are placed at the join nodes of a program flow graph to represent different definitions of a variable coming from different incoming edges. The condition under which a definition reachs a join node is not represented in the ϕ-function. In the GSA representation, several types of gating functions are defined to represent the different types of conditions at different join nodes.

To obtain the GSA form used in Polaris, program variables are renamed such that each time the variable is defined (e.g., reaches a join node) it is given a new name. Then, each time a variable is used, it is named according to which definition reaches it. In the program shown in Figure 4, each variable is assigned only once, so no renaming is necessary to obtain the GSA form. Our algorithm proceeds backwards from use to definition. To prove that $\mathtt{MP} \geq \mathtt{M} * \mathtt{P}$, the algorithm starts at loop `J` and backward-substitutes `MP` with $\mathtt{M} * \mathtt{P}$ as defined in statement $\mathtt{S}_2$. Because the goal is satisfied, the algorithm stops at this point and no further replacements are performed.

```
do I = 2,N
    do J = 1, I - 1
        IND(J) = 0
        A(J) = X(I,J) - Y(I,J)
        R = A(J) + W
        if (R .LT. RCUTS) IND(J) = 1
    end do
    P = 0
    do K = 1,I - 1
        if (IND(K) .NE. 0) then
            P = P + 1
            IND(P) = K
        end if
    end do
    do L = 1,P
        M = IND(L)
        X(I,L) = A(M) + Z
    end do
end do
```

Figure 5: Example from BDNA

A more complicated example of the need for global information is shown in Figure 5, taken from the most time-consuming loop in BDNA. Several intermediate

variables need to be privatized to parallelize the outermost loop in Figure 5. They are the scalar variables `R, P,` and `M`, and the arrays `IND` and `A`. Except for array `A`, it is easy to determine that these intermediate variables are privatizable.

To determine whether `A` is privatizable in loop `I`, it is necessary to determine the range of the use of `A` in loop `L`. By analyzing the subscript and the range of the loop `L`, it is easy to determine that the range is $\{\texttt{A(IND(1))}, \texttt{A(IND(2))}, \ldots, \texttt{A(IND(P))}\}$. The possible dominating definition for `A` is in loop `J`, where `A` is defined for the range `A(1:I-1)`. To prove that the definition in loop `J` dominates all the uses in loop `L`, we need to prove that $\{\texttt{A(IND(1))}, \texttt{A(IND(2))}, \ldots, \texttt{A(IND(P))}\}$ falls in the range of `A(1:I-1)`.

Our GSA based, demand-driven, sparse evaluation algorithm works well in situations like this where it is necessary to propagate values from complicated control structures with conditional assignments and statically assigned symbolic arrays. The analysis determines how many elements of `IND` are defined in loop `K` making use of the fact that the subscript `P` for the assignment to `IND(P)` is a monotonically increasing variable with an initial value of 1 and step of 1. Using a monotonic variable identification technique similar to induction variable identification, the algorithm determines that all the elements in $\{\texttt{IND(1)}, \texttt{IND(2)}, \ldots, \texttt{IND(L)}\}$ are assigned in loop `K`.

Now that the algorithm knows the definition point for $\{\texttt{IND(1)}, \texttt{IND(2)}, \ldots, \texttt{IND(P)}\}$, it can substitute the terms in $\{\texttt{A(IND(1))}, \texttt{A(IND(2))}, \ldots, \texttt{A(IND(P))}\}$ which are loop-variant with their values. Each of them takes on a value of loop index `K`. Because the value of `K` falls in the range `[1:I-1]`, $\{\texttt{IND(1)}, \texttt{IND(2)}, \ldots, \texttt{IND(P)}\}$ will also fall in the same range. Hence all the uses of `A` fall within the range `[1:I-1]` and are therefore dominated by the definition `A(1:I-1)`. Thus, the algorithm determines that the array `A` is privatizable in loop `I`.

3.5 Framework for Run-Time Analysis

The access pattern of some programs cannot be determined at compile time, either because of limitations in the current analysis algorithms or because the access pattern is a function of the input data. For example, compilers usually conservatively assume data dependences in the presence of subscripted subscripts. Although more powerful analysis techniques could remove this limitation when the index arrays are computed using only statically-known values, nothing can be done at compile-time when the index arrays are a function of the input data. Therefore, if data dependences such as these are to be detected, the analysis must occur at run-time. Because of the overhead involved, it is very important that run-time techniques be fast as well as effective.

3.5.1 Runtime Detection of Dependences

Consider a `do` loop for which the compiler cannot statically determine the access pattern of a shared array `A` that is referenced in the loop. Instead of executing the loop sequentially, the compiler could decide to speculatively execute the loop as a `doall` and generate code to determine at run-time whether the loop was, in fact, fully parallel. If the subsequent test finds that the loop was not fully parallel, then it will be re-executed sequentially.

To do this, it is necessary to have the ability to restore the original state when re-execution is needed. One strategy is to save the values of some arrays before starting the parallel execution of the loop and restore these values if the sequential re-execution is needed. However, in our implementation, some of the values computed during the parallel execution are stored in temporary locations and then stored in permanent locations if the parallel execution was correct.

In order to implement such a strategy, we have developed a run-time technique, called the *Privatizing Doall test* (*PD test*), for detecting the presence of cross-iteration dependences in a loop [15]. If there are any such dependences, this test does not identify them; it only flags their existence. In addition, if any variables were privatized for speculative parallel execution, this test determines whether those variables were, in fact, validly privatized. Our interest in identifying fully parallel loops is motivated by the fact that they arise frequently in real programs.

3.5.2 The PD Test

The PD test is applied to each shared variable referenced during the loop whose accesses cannot be analyzed at compile-time. For convenience, we discuss the test as applied to only one shared array, say `A`. Briefly, the test traverses and marks shadow array(s) during speculative parallel execution using the access pattern of `A`, and after loop termination, performs a final analysis to determine whether there were cross-iteration dependences between the statements referencing `A`.

For each iteration, the first time an element of `A` is written during that iteration, the corresponding element in the write shadow array A_w is marked. If,

during an iteration, an element in A is read, but never written, then the corresponding element in the read shadow array A_r is marked. Another shadow array A_{np} is used to flag the elements of A that *cannot* be privatized: an element in A_{np} is marked if the corresponding element in A is both read and written, and is read first, for any iteration.

A post-execution analysis determines whether there were any cross-iteration dependences between statements referencing A as follows. If $\texttt{any}(A_w(:) \cap A_r(:))$[1] is true, then there is at least one flow- or anti-dependence that was not removed by privatizing A. If $\texttt{any}(A_w(:) \cap A_{np}(:))$ is true, then A is not privatizable (some element is read before being written in an iteration). The counter w_A records the total number of writes done to A_w by all iterations, and m_A is the total number of marks in A_w. If $w_A \neq m_A$, then there is at least one output dependence (some element is overwritten); however, if A is privatizable then these dependences were removed by privatizing A. The PD test is fully parallel and requires time $O(a/p + \log p)$, where p is the number of processors, and a is the total number of accesses made to A in the loop.

3.5.3 Performance of run-time techniques

It can be shown that if the PD test passes, i.e., the loop is in fact fully parallel, then a significant portion of the ideal speedup of the loop is obtained. In particular, the speedups obtained range from nearly 100% of the ideal in the best case, to *at least* 25% of the ideal in the worst case (as derived from the parallel model). On the other hand, if the PD test fails, i.e., the loop is not fully parallel, then the sequential execution time will be increased by the time required by the failed parallelization attempt. Since the PD test is fully parallel, this *slowdown* is proportional to $\frac{1}{p}T_{seq}$, where T_{seq} is the sequential execution time of the loop. If the target architecture is a MPP with *hundreds* of processors, then the worst case potential speedups can be very high and the cost of a failed test becomes a very small fraction of sequential execution time. Thus, speculating that the loop is fully parallel has the potential to offer large gains in performance, while at the same time risking only a small increase in the sequential execution time.

In Figure 6, we show experimental results of a Fortran implementation of the PD test on loop NLFILT/300 in a subroutine of TRACK. The measurements were made on an 8-processor Alliant FX/80 machine. The access pattern of the shared array in

[1] any returns the "OR" of its vector operand's elements, i.e., $\texttt{any}(v(1:n)) = (v(1) \vee v(2) \vee \ldots \vee v(n))$.

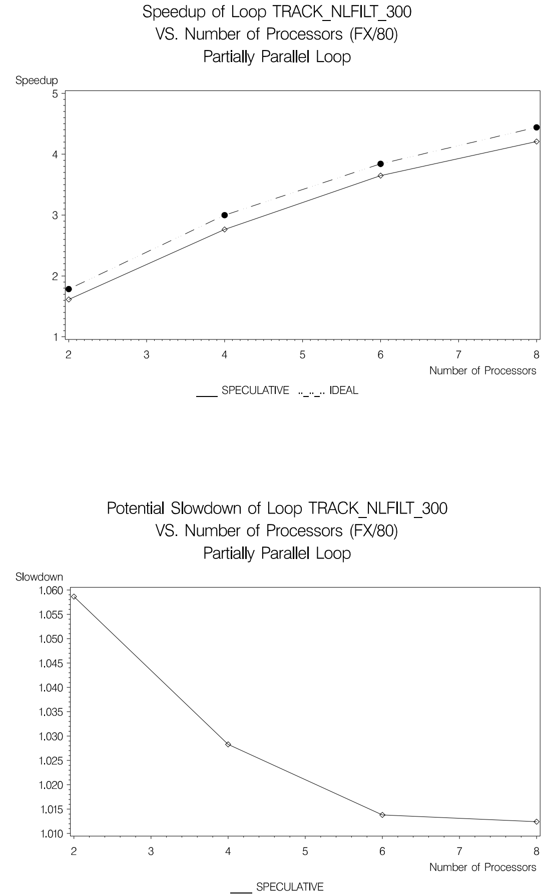

Figure 6: Speedup and Potential Slowdown for NLFILT/300 from TRACK

this loop cannot be analyzed by the compiler since the array is indexed by a subscript array that is computed at run-time. In addition, this loop is parallel for only 90% of its invocations. In the cases when the test failed, we restored state, and re-executed the loop sequentially. The speedup reported includes both the parallel and sequential instantiations. The potential slowdown reflects the increase in total execution time that would have resulted if the PD test had shown that the loop was not fully parallel: it is expressed as the ratio between $(T_{seq} + T_{pdt})$ and T_{seq}, where T_{pdt} is the time required for the PD test.

Our experimental results indicate that our techniques for loops with unknown iteration spaces usually yield significant speedups when compared to the available parallelism in the original loop. The experiments have also shown that the overhead associated with these techniques is generally very small. In addition, we have found that the additional memory re-

quirements do not make these techniques impractical for the programs we have examined.

The implementation of these techniques in Polaris is underway with the SGI Challenge as the first target architecture.

4 Evaluation

We place great importance on the evaluation of our work. As stated in the introduction, we began testing commercial parallelizers in the 1980s and found that when faced with actual scientific programs they performed poorly. Based on an analysis of the causes of these deficiencies, we have identified and implemented several transformations in the Polaris compiler, and now once again we must evaluate where and how we have or have not succeeded.

4.1 The Benchmark Codes

The scientific programs that we used in our previous work were the Perfect Benchmarks® making it natural to use them in our evaluation of Polaris. We also included other scientific programs in order to determine the extent to which our techniques apply to programs in general. For our evaluation efforts we chose six of the 13 Perfect codes, eight codes from the SPEC CFP92 and CFP95 suites, plus two currently-in-use codes that we obtained from the National Center for Supercomputing Applications. Table 1 summarizes the origin, number of lines and serial execution time for each code.

`arc2d` is an implicit finite difference code for fluid flow, `bdna` simulates molecular dynamics of biomolecules, `flo52` analyzes transonic flow past an airfoil, `mdg` is a molecular dynamics model for water molecules, `ocean` solves Boussinesq fluid layer, and `trfd` is a kernel for quantum mechanics calculations.

`applu` is a parabolic/elliptic PDE solver, `appsp` solves systems using gaussian elimination, `hydro2d` calculates galactical jets using the Navier Stokes method, `su2cor` simulates a Monte Carlo quantum mechanics problem, `swim` solves shallow water equations using a finite difference scheme, `tfft2` is a collection of FFT routines from NASA codes, `tomcatv` generates 2D meshes around geometric domains, and `wave5` solves particle and Maxwell's equations.

`cmhog` is a 3D ideal gas dynamics code and `cloud3d` computes 3D models of atmospheric convection.

Program	Origin	Lines of Code	Ser. Time (sec.)
APPLU	SPEC	3870	1203
APPSP	SPEC	4439	1241
ARC2D	PERFECT	4694	215
BDNA	PERFECT	4887	56
CMHOG	NCSA	11826	2333
CLOUD3D	NCSA	9813	20404
FLO52	PERFECT	2370	38
HYDRO2D	SPEC	4292	1474
MDG	PERFECT	1430	178
OCEAN	PERFECT	3288	118
SU2COR	SPEC	2332	779
SWIM	SPEC	429	1106
TFFT2	SPEC	642	946
TOMCATV	SPEC	190	1327
TRFD	PERFECT	580	20
WAVE5	SPEC	7764	788

Table 1: Benchmark codes studied

4.2 The Results

Figure 7 presents a comparison of the speedups obtained by Polaris with those of SGI's PFA compiler. Sixteen programs (described in Section 4.1) were used in the comparison. The programs were executed (in real-time mode for accuracy) on eight processors on an SGI Challenge with 150 MHz R4400 processors. Figure 7 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. Additional strategies are necessary for handling these programs, and are outlined in [3]. 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, `appsp` and `tomcatv`; although PFA detects as much parallelism as Polaris, the code it generates does not take much advantage of it.

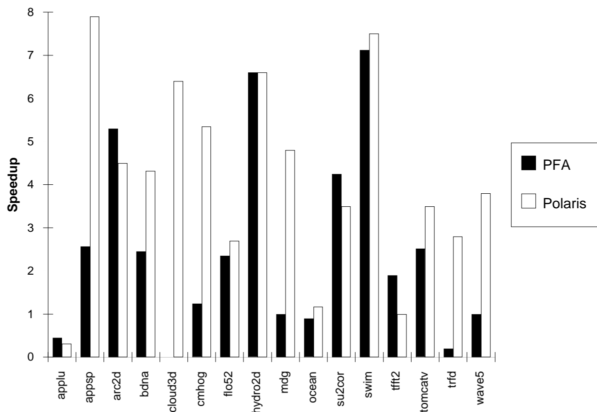

Figure 7: Speedup: Polaris vs. SGI PFA

5 Conclusion

We have presented Polaris, a new parallelizing compiler, developed at the University of Illinois. Polaris includes a powerful basic infrastructure for manipulating Fortran programs and a number of improved analysis and transformation passes, most notably subroutine inline expansion, symbolic analysis, induction and reduction variable recognition and solution, data-dependence analysis, array privatization, and run-time analysis.

The current prototype of the Polaris compiler is able to parallelize our evaluation suite of programs significantly better than available compilers. Polaris' performance is as good as the best manual parallelization efforts that we are aware of in many cases.

Previous attempts at automatic parallelization were successful in a limited number of cases. Now, however, we are successful in half of the codes tested in our evaluation suite, and expect to further increase this percentage using new and extended techniques described in [3]. This is a substantial improvement, and moves parallelizing compilers one step closer to the class of tools considered essential in a parallel computing environment. We have come a significant step closer to the goal of making parallel computing available to a broader user community.

A couple of remaining issues include the representativeness of our program suite and the machine model we are using. We believe that the Perfect Benchmarks, SPEC CFP suites, and the NCSA suite are a good starting point for a truly representative high-performance computer workload. The fact that newly inspected programs from the SPEC and NCSA suites have confirmed our previous findings using the Perfect Benchmarks indicates that we may indeed be converging in our search for the right compiler ingredients.

The output of Polaris is suitable for a wide variety of machines that provide a global address space. Early results of an implementation for the Cray T3D, for example, have shown that even in the absence of a cache-coherence scheme, significant speedups are achievable using the same basic framework for analysis [11]. In addition, due to the similarity in the virtual shared-memory model presented to many users of strictly message-passing machines[2], we expect many of the same techniques implemented in Polaris will apply to these architectures as well. Nevertheless, global address-based features will most likely be part of many parallel machines in the coming years. This is already becoming evident as can be seen by the recent MPP announcements of Cray, Convex, and other manufacturers.

In conclusion, the key innovation provided by Polaris is improved recognition of parallelism. Such recognition is a necessary prerequisite to porting conventional programs to any parallel machine available today.

References

[1] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer. Boston, MA, 1988.

[2] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers. *Int'l. Journal of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.

[3] 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.

[2]For example, HPF directives inserted in Fortran 77 code

[3]CSRD reports are available via anonymous FTP from ftp.csrd.uiuc.edu:CSRD_Info, or the World Wide Web site http://www.csrd.uiuc.edu

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

[5] 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.

[6] 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.

[7] 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.

[8] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Languages and Compilers for Parallel Computing. Fourth International Workshop. Santa Clara, CA. Lecture Notes in Computer Scinece 589*, pages 65–83, August 1991.

[9] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–586, October 1994.

[10] Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. Interprocedural transformations for parallel code generation. *Supercomputing'91*, pages 423–434, 1991.

[11] Yunheung Paek and David Padua. Automatic Parallelization for Noncoherent Cache Multiprocessors. Technical report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercom puting Res. & Dev., April 1996. CSRD Report No. 1485.

[12] Paul M. Petersen and David A. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. *Presented at ICS'93, Tokyo, Japan*, pages 107–116, July 19-23, 1993.

[13] Bill Pottenger and Rudolf Eigenmann. Parallelization in the Presence of Generalized Induction and Reduction Variables. Technical Report 1396, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1995.

[14] 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.

[15] Lawrence Rauchwerger and David Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization . *Proceedings of the 8th ACM International Conference on Supercomputing, Manchester, England*, pages 33–43, July 1994.

[16] 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.

[17] Peng Tu and David Padua. Automatic array privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proc. Sixth Workshop on Languages and Compilers for Parallel Com puting*, volume 768 of *Lecture Notes in Computer Science*, pages 500–521, Portland, OR, August 1993. Springer Verlag.

[18] 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.

[19] Stephen Weatherford. High-Level Pattern-Matching Extensions to C++ for Fortran Program Manipulation in Polaris. Technical Report 1350, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1994.

[20] Michael Wolfe. Triangular Banerjee's Inequalities with Directions. Technical report, Oregon Graduate Institute of Science and Technology, June 1992. CS/E 92–013.