Polaris: Improving the Effectiveness of Parallelizing Compilers

William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu and Stephen Weatherford

Center for Supercomputing Research and Development
Coordinated Science Laboratory
University of Illinois

Abstract. It is the goal of the Polaris project to develop a new parallelizing compiler that will overcome limitations of current compilers. While current parallelizing compilers may succeed on small kernels, they often fail to extract any meaningful parallelism from large applications. After a study of application codes, it was concluded that by adding a few new techniques to current compilers, automatic parallelization becomes possible. 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 allow data dependent behavior.

1 Introduction

Supporting standard programming languages on any kind of computer system is and has been an important issue in computer science. For parallel machine architectures this issue is not only more difficult but also crucial to make these machines easy to use. Parallel machines are more intricate and demand a deeper understanding from those users who have to exploit machine features through specific language constructs. As machine structures are evolving rapidly, these users would have to repeatedly learn new machine-specific features and language elements. These issues make the discipline of supporting standard languages, and thus automatic program transformations, an interesting and important research area.

For many years, the Center for Supercomputing Research and Development (CSRD) has been working toward the goal of making parallel computing a practical technology. Parallelizing compilers have been playing an important role in this quest. The present project has its early roots in a compiler evaluation effort of the late 80s, where we have found that despite the success on kernel benchmarks, available compilers were not very effective on large programs [EHLP91, BE92]. New measurements on a representative set of real programs were made possible, thanks to the Perfect Benchmarks® effort, which was initiated by CSRD, with participation from many other institutions [BCK+89].

Based on these observations, we have hand parallelized the program suite as a major new approach to identifying effective program transformations [EHLP91, EHJP92]. As a result we have found that not only can real applications be parallelized effectively, but the transformations can also be automated in a parallelizing compiler. One issue remained: we had not actually implemented these transformations and thus not delivered the final proof that parallelizing compilers can be improved dramatically.

To resolve this issue we have implemented a prototype of the Polaris compiler and have evaluated its effectiveness. The compiler consists of a sizable basic infrastructure for manipulating Fortran programs, which we describe in Section 2. The analysis and transformation techniques investigated in Polaris are discussed in Section 3. Finally, in Section 4 we present the current status of the Polaris prototype compiler.

[1] 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.

2 Internal Organization of the Compiler

The aim in the design of Polaris' internal organization [FHP+93] was to create an internal representation (IR) that enforced correctness, was robust and, through high-level functionality, 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 have chosen 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 can build layers of functionality which allow the IR to emulate more complex forms and provide higher-level operations.

We chose to implement Polaris in the object-oriented language C++ as it both allowed us structural flexibility and gave us the 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 in a transformation. Transformations are never allowed to let the code enter a state that will no longer generate proper Fortran syntax. The system also guarantees that the control flow graph is consistent through automatic incremental updates of this information as a transformation proceeds. The automatic consistency maintenance has drastically decreased the time required to develop new optimizations within Polaris' production system.

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

- A clear understanding of who is responsible for the deallocation of an object. The owner of an object is always responsible for its destruction. In Polaris we use the convention that a pointer is used to indicate ownership, and a reference is used to indicate that the object is not owned.
- The detection and reporting of aliased structures (structure sharing is not allowed) with 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 the object.
- Detection of object deletion when that object is being referenced from another part of Polaris. If the deleted object is subsequently referenced, Polaris will abort with an internal consistency error. Furthermore, dangling pointers and their associated problems are avoided by the reference counting of all objects stored in collections.
- 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 if the assumption is incorrect.

In our implementation, we have followed the usual object-oriented approach 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 includes 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 was intuitive and straightforward. The **Program** class, for instance, is little more than a collection of **ProgramUnit**s. Among the included member functions are routines for reading complete Fortran codes as well as displaying them. There are also member functions for adding additional **ProgramUnit**s as well as merging **Program**s.

The **ProgramUnit** class is, similarly, a holder for the various data structure elements that make up a Fortran program unit such as a statement list, a symbol table, common blocks, and equivalences.

Statements are simple, non-recursive structures kept in a list. There is no notion of statement blocks. However, we have made the implementation flexible enough that member functions which simulate the existence of statement blocks can easily be implemented on top of the current **Statement** class.

The data fields declared in the base **Statement** class (and which, therefore, 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 have 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 initial, limit, and step expressions. Each statement class also declares a number of member functions such as 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 to examine expressions for 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 expression structural equality routines, 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 are used by the reduction recognition pass.

The **StmtList** class also contains much functionality which is frequently used during transformation passes. The **StmtList** class is, intuitively, a list of statements. In addition, it contains an extensive variety of high-level member functions for manipulating this list, all of which include automatic updating of control flow and loop-nesting information.

To maintain complete control of consistency inside the **StmtList** class, the manipulation of statements or statement lists are 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 by a statement outside of the statement block being deleted is flagged as a run-time error during Polaris' execution. The programmer can also defer this consistency management by using a **List<Statement>** for the modification. In this way, we can create a section of code that is inconsistent during its creation and modification but which is checked for consistency when it is incorporated into the program. These safeguards create an environment where traditionally time-consuming errors are immediately recognized and correct code can be quickly created.

3 Transformations

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. After we have gained more experience with the use of inline expansion in Polaris, we are planning on augmenting it with other forms of interprocedural analysis.

The second transformation 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 are now able to be automatically analyzed.

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

The final transformation discussed in this paper is a run-time technique 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 not currently implemented in Polaris but will include inspector/executor [SMC91] style implementations as well as implementations based on speculative execution.

3.1 Inline Expansion

The Polaris inliner is designed to provide three types of services: complete inline expansion of 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 complete inline expansion to allow full flow-sensitive interprocedural analysis, which is especially important for privatization. A driver routine is provided in the Polaris base to perform complete inline expansion for analysis.

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.

The first time a subprogram is to be expanded, the inliner creates a "template" object for each subprogram as it is represented within the top-level program (e.g., with subprogram variables renamed to avoid conflicts with top-level program symbols). At each individual call site, the inliner driver passes this template object back to the inliner, which makes a "work" copy of the object, makes on this copy call site-specific transformations, such as replacing references to formal parameters with actual parameters, and copying the statements and variables of the routine into the top-level program (eliminating the original CALL statement or function call).

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.

After the analysis of a completely inlined program, the template objects created during analysis can be used as a base for performing selective inline expansion of subprograms for code generation. Data collected using complete inline expansion for analysis can be used to guide selective modification of subprograms using such techniques as cloning, loop embedding, and loop extraction [HKM91].

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. An example is the passing of a REAL actual array to a COMPLEX formal array. This is handled automatically, but requires the favorable assumption that the variables are properly aligned in memory.

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 commutative and distributive, 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 some inaccuracy, and therefore in most compilers the user has the option of disabling this transformation. However, we have not found this to be a problem for 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, and α may be null (i.e., A is a scalar variable). The data-dependence pass later removes the flag if it can prove independence. The backend code generator for our target machine produces parallel code that implements flagged variables as reductions. More than one reduction statement can occur in a loop and they may sum into different array elements in different loop iterations. We are currently implementing a transformation to generate code for this case in Polaris.

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 loop index[2].

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 appear in the (right-hand-side) increment of other induction variables (we will call these *coupled induction variables*), and two, when induction variables occur within *triangular* loop nests[3].

The following example shows a triangular loop nest that contains a coupled 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 coupled induction variables (`K1` and `K2`) cause an unusually large code expansion.

```
K1 = 0
K2 = 0                               do I = 1, N
do I = 1,N                              do J = 1, I
   do J = 1,I                              X((I**4-2*I**3+3*I**2-2*I
      K1 = K1 + 1        =>      &           +(4*I**2-4*I)*J
      K2 = K2 + K1               &           +4*J**2+4*J)/8) = ...
      X(K2) = ...                       end do
   end do                            end do
end do
```

The Polaris induction variable substitution algorithm performs three steps in order to recognize and substitute induction variables:

1. Find candidate induction statements by recognizing recurrence patterns of scalar variables which are incremented by either a loop-invariant expression or an expression containing other candidate induction variables. The patterns we have encountered in our programs are relatively simple, and we have implemented a straightforward recognition scheme which finds dependence relationships between induction variables and detects cycles.

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

[2] Assuming a normalized loop

[3] In triangular loop nests, inner loop bounds depend on outer loop indices

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 computes its closed form of the induction variable.

3. Substitute all occurrences of the induction variables. This step is the same as in other compilers. 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.

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 [PP93]. 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* (i.e., 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 [BE94b]. 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 1. This loop nest, taken from 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((I*(N**2+N)+J**2-J)/2+K+1) = ···
      end do                           end do
   end do                           end do
   X0 = X0+(N**2+N)/2
end do                           end do
```

Fig. 1. Simplified version of loop nest OLDA/100 from TRFD, before and after induction variable substitution.

3.3.1 Range Test

To handle such nonlinear expressions, we have developed a symbolic dependence test called the *range test* [BE94a]. The range test is an extension of a symbolic version of Triangular Banerjee's Inequalities test [WB87, Ban88, HP91]. 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 do 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 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 $a \leq i \leq b$, 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, say 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 $a \leq i \leq b$, 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 1. 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 $b_1(i,j+1) - b_1(i,j) = j \geq 0$. 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 [BE94a] for other tests that use these minimum and maximum values of f and g.

Returning to the example for Figure 1, 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 since $a_2(i+1) - a_2(i) = n^2 + n > 0$. 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 1 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 two parts. One part determines symbolic lower and upper bounds, called ranges, for each variable at each point of the program. The other part uses these ranges to compare symbolic expressions. 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 2. 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(258*X*J + 129*K + I + 1) = ···
         A(258*X*J + 129*K + I + 1 + 129*X) = ···
      end do
   end do
end do
```

Fig. 2. Simplified version of loop nest FTRVMT/109 from OCEAN

The range test subsumes Banerjee's Inequalities, which have been shown to be effective [PKK91] for real programs [PP93], in proving loops to be parallel with the assumption that they contain only linear subscript expressions. We expect the range test to be equally effective. In addition, 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 [BE94b]. Our implementation of the range test in Polaris supports these claims. For the evaluation suite of codes for Polaris, described later, we have found that applying the range test alone was sufficient to identify all important loop nests as parallel for these codes.

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* [TP93].

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 needs 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
```

Fig. 3. Example for array privatization

A simple example where such analysis is necessary for array privatization is shown in Figure 3. 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 [TP94], based on a Static Single Assignment (SSA) representation, to obtain global information. To obtain the SSA form, program variables are renamed such that each time the variable is defined 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 3, each variable is assigned only once, so no renaming is necessary to obtain the SSA form. Our demand-driven 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.

A more complicated example of the need for global information is shown in Figure 4, taken from the most time-consuming loop in BDNA. Several intermediate variables need to be privatized to parallelize the outermost loop in Figure 4. 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 $\{\mathtt{A}(\mathtt{IND}(1)), \mathtt{A}(\mathtt{IND}(2)), \ldots, \mathtt{A}(\mathtt{IND}(\mathtt{P}))\}$. 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 $\{\mathtt{A}(\mathtt{IND}(1)), \mathtt{A}(\mathtt{IND}(2)), \ldots, \mathtt{A}(\mathtt{IND}(\mathtt{P}))\}$ falls in the range of `A(1:I-1)`.

Our SSA 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 demand-driven 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 $\{\mathtt{IND}(1), \mathtt{IND}(2), \ldots, \mathtt{IND}(\mathtt{L})\}$ are assigned in loop `K`.

```
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
```

Fig. 4. Example from BDNA

Now that the algorithm knows the definition point for $\{\texttt{IND}(1), \texttt{IND}(2), \ldots, \texttt{IND}(\texttt{P})\}$, it can substitute the loop variant terms in $\{\texttt{A}(\texttt{IND}(1)), \texttt{A}(\texttt{IND}(2)), \ldots, \texttt{A}(\texttt{IND}(\texttt{P}))\}$ 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]`, $\{\texttt{IND}(1), \texttt{IND}(2), \ldots, \texttt{IND}(\texttt{P})\}$ 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 Detecting data dependences at run-time

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

```
do I = 1, 8
   ...        = A(T(I))              T(1:8) = [2 2 2 10 8 8 8 10]
   A(U(I))    = ...                  U(1:8) = [1 3 5  4 7 3 6 12]
   ...        = A(V(I))              V(1:8) = [1 3 2 10 7 3 8 12]
end do
```

	Position in shadow arrays												w_A	m_A
	1	2	3	4	5	6	7	8	9	10	11	12		
A_w	1	0	1	1	1	1	1	0	0	0	0	1	8	7
A_r	0	1	0	0	0	0	0	1	0	1	0	0		
A_{np}	0	0	0	0	0	0	0	0	0	0	0	0		
$A_w \wedge A_r$	0	0	0	0	0	0	0	0	0	0	0	0		

Fig. 5. The PD Test.

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 [RP94]. 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 $\mathtt{A_w}$ is marked. If, during an iteration, an element in `A` is read, but never written, then the corresponding element in the read shadow array $\mathtt{A_r}$ is marked. Another shadow array $\mathtt{A_{np}}$ is used to flag the elements of `A` that *cannot* be privatized: an element in $\mathtt{A_{np}}$ 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 $\mathtt{any}(\mathtt{A_w}(:) \cap \mathtt{A_r}(:))$[4] is true, then there is at least one flow- or anti-dependence that was not removed by privatizing `A`. If $\mathtt{any}(\mathtt{A_{np}}(:))$ 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 $\mathtt{A_w}$ by all iterations, and m_A is the total number of marks in $\mathtt{A_w}$. If $w_A \neq m_A$, then there is at least one output dependence (some element is overwritten); however, if `A` is privatizable (i.e., if $\mathtt{any}(\mathtt{A_{np}}(:))$ is false), 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.

The PD test is illustrated using the loop shown in Figure 5. The access pattern is given by the subscript arrays `T`, `V`, and `U`. Since $\mathtt{A_w}(:) \wedge \mathtt{A_r}(:)$ and $\mathtt{A_{np}}(:)$ are zero everywhere, the loop was a `doall`, but only after privatizing `A` since $w_A \neq m_A$.

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

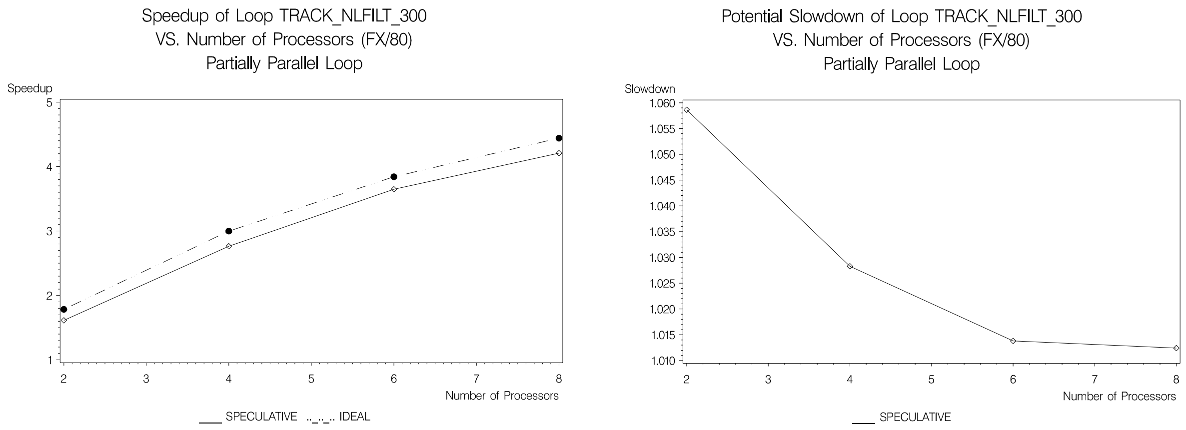

Fig. 6. Speedup and Potential Slowdown for NLFILT/300 from TRACK

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* or, in the future *thousands*, of processors, then the worst case potential speedups reach into the hundreds, 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 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 requirements do not make these techniques impractical for the programs we have examined.

4 Evaluation of Polaris Parallelization

We place great importance on the evaluation of our work. When we began testing commercial parallelizers in the 1980s, we found that they performed well on small, synthetic loops, but when faced with actual scientific programs, they performed poorly. We analyzed them to determine the reasons for the deficiencies and identified several improvements that could be made in them. Now, we are implementing those improvements in the Polaris compiler, and 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®, which made it natural to use them to evaluate Polaris. We also included other scientific programs in order to demonstrate that our techniques apply to programs in general. We chose 6 of the 13 Perfect codes, plus two currently-in-use codes that we obtained from the National Center for Supercomputing Applications for our first evaluation efforts. We have determined that many of the other Perfect codes will require the use of some run-time techniques.

From the Perfect Benchmarks, we chose the programs ARC2D, BDNA, FLO52, MDG, OCEAN, and TRFD. Three of these codes (ARC2D, BDNA, and FLO52) have proven to be at least moderately parallelizable with traditional techniques. The other three (MDG, OCEAN, and TRFD) were poorly parallelized by traditional techniques. The two NCSA codes which we chose were CMHOG and CLOUD3D.

4.2 The Evaluation Metrics

Since we have focused our efforts on "finding parallelism" in this first year of the Polaris project, we devised a simple metric which attempts to quantify how well we achieved that. The metric is called *percent parallel coverage*. During the sequential execution of a program, we record how much time is spent executing each loop, determine the percentage of the overall running time of the sequential program, and call that figure the *percent coverage* of that loop. By adding up the coverages of each parallelized loop, we obtain the percent parallel coverage for the program. This gives us a single figure that portrays the quality of the parallelization. Even so, the percent parallel coverage is only a rough predictor of the eventual speedup that might be obtained from the code.

A second way of evaluating Polaris is to examine how well it does on loops ranked by running time. We ordered the loops in a program by average sequential running time per invocation, then divided the list into the longest-running 10%, the longest-running 50%, and finally all loops contributing at least 0.01% of the sequential running time. We tallied how many loops in each category required only "traditional" parallelization techniques, and how many required new Polaris techniques. The techniques found in the compilers we evaluated were scalar privatization, scalar reductions, recognition of induction variables in rectangular loop nests, and a simple subscript test. The major Polaris techniques were array privatization, array reductions, multi-site reductions, triangular inductions, the range test, and interprocedural analysis (as implemented through inline expansion).

4.3 The Results

Table 1 shows the resulting overall percent parallel coverage produced by the Polaris techniques for each of the eight benchmark codes, and what coverage could be gained by traditional techniques. In all cases, Polaris improved on the traditional techniques, and sometimes, quite dramatically, such as in the case of TRFD, where traditional techniques cover less than 1 percent, while Polaris found parallelism covering 99 percent of the sequential time.

We counted the number of loops in each category which were "intrinsically" serial (i.e., these could not be hand parallelized), how many required at least one of Polaris' techniques to be parallelized, how many could be parallelized by traditional techniques, and the total number of loops. These results were catalogued for the longest-running 10%, 50% and all loops (as described above). The results are displayed for each program in Table 2.

This table shows that in all percent brackets, Polaris' techniques are able to parallelize significantly more loops than traditional techniques. The fact that this ratio is most pronounced in the top 10% loops demonstrates the real impact of the new technology. Quite often we found that Polaris' techniques could parallelize the outer loop in a nest, while traditional techniques were sufficient only for the inner loops.

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, notably subroutine inline expansion, symbolic analysis, induction and reduction variable recognition, data-dependence analysis, array privatization, and run-time analysis.

The current prototype of the Polaris compiler is able to parallelize our first evaluation suite of programs significantly better than available compilers. In many cases this is as good as the best manual parallelization.

Previous attempts at automatic parallelization were only successful for two of the programs in our test suite. Now we are successful in half of the Perfect Benchmarks and we expect to further increase this portion in the near future. This is a substantial improvement which expands the set of programs for which parallelizing compilers are successful from small and simple codes to medium sized and complicated. We have come a significant step closer to the goal of making parallel computing available to the broad user community.

Some remaining issues are the representativeness of our program suite and the machine model we are using. We believe that the Perfect Benchmarks plus the "NCSA suite" are a good starting point for a truly representative high-performance computer workload. The fact that our newly inspected programs have confirmed previous findings about effective parallelization techniques indicates that we are in fact converging in our search for the right compiler ingredients.

The output of Polaris is suitable for machines that provide a global address space. To use Polaris on message-passing machines one will need to develop complementing optimization techniques. We do not plan to develop these techniques since a number of such projects are currently underway. Furthermore, global address-based features will most likely be part of many parallel machines in the near term. This is already starting to happen as can be seen in the recent MPP announcements of Cray and Convex. However, it is important to note that Polaris' innovation is in improved recognition of parallelism, which is a necessary step for porting programs to any parallel machine available today.

References

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer. Boston, MA, 1988.

[BCK+89] 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.

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

[BE94a] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. Technical Report 1345, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., April 1994.

[BE94b] William Blume and Rudolf Eigenmann. Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. Technical Report 1332, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., January 1994.

[5] 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

[EHJP92] Rudolf Eigenmann, Jay Hoeflinger, G. Jaxon, and David Padua. The Cedar Fortran Project. Technical report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res, & Dev., April 1992. CSRD Report No. 1262.

[EHLP91] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[FHP+93] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. Technical report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. and Dev., October 1993. CSRD Report No. 1317, UILU-ENG-93-8038.

[HKM91] Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. Interprocedural transformations for parallel code generation. *Supercomputing'91*, pages 423–434, 1991.

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

[PKK91] K. Psarris, D. Klappholz, and X. Kong. On the accuracy of the Banerjee test. *Journal of Parallel and Distributed Computing*, 12(2):152–157, June 1991.

[PP93] 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.

[RP94] Lawrence Rauchwerger and David Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization . Technical Report 1329, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. and Dev., January 1994.

[SMC91] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991.

[TP93] 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.

[TP94] Peng Tu and David Padua. Demand-Driven Symbolic Analysis. Technical Report 1336, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., Febraury 1994.

[WB87] Michael Wolfe and Utpal Banerjee. Data Dependence and its Application to Parallel Processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[Wea94] Stephen Weatherford. High-Level Pattern-Matching Extensions to C++ for Fortran Program Manipulation in Polaris. Technical Report 1350, Univ of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., May 1994.

PROGRAM	Polaris	traditional
MDG	99.95	68.54
ARC2D	99.94	69.96
TRFD	99.90	<0.01
FLO52	99.79	91.06
BDNA	97.95	32.33
OCEAN	95.42	32.58
CLOUD3D	84.55	43.09
CMHOG	81.93	17.11

Table 1. Percent coverage of the serial execution time of loops that can be parallelized with Polaris techniques and traditional techniques.

	top 10%			top 50%			all loops		
PROGRAM	serial	Pol/trad	total	serial	Pol/trad	total	serial	Pol/trad	total
ARC2D	1	7/3	8	8	32/28	40	10	70/64	80
FLO52	2	4/3	6	4	23/20	27	4	50/43	54
TRFD	0	0/0	1	0	2/0	3	0	4/0	6
MDG	1	2/0	3	3	10/3	14	3	23/16	28
BDNA	2	3/0	5	3	14/8	22	4	28/1	44
OCEAN	0	5/0	6	0	27/5	28	0	54/24	56
CLOUD3D	5	8/4	13	14	51/32	65	24	105/84	131
CMHOG	1	4/1	5	2	21/14	23	4	43/31	47

Table 2. Loop count in the categories: serial, needing Polaris/traditional techniques, total loops, for the evaluation codes. The loops were ordered by their average serial execution time, then divided into the groups: top 10%, top 50%, and all loops with 0.01% or more of the total serial time of the program.