Idiom Recognition in the Polaris Parallelizing Compiler

Bill Pottenger and Rudolf Eigenmann
potteng@csrd.uiuc.edu, eigenman@csrd.uiuc.edu
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, Illinois 61801-2307 *

Abstract

The elimination of induction variables and the parallelization of reductions in FORTRAN programs have been shown to be integral to performance improvement on parallel computers [7, 8]. As part of the Polaris project [5], compiler passes that recognize these idioms have been implemented and evaluated. Developing these techniques to the point necessary to achieve significant speedups on real applications has prompted solutions to problems that have not been addressed in previous reports on idiom recognition techniques. These include analysis techniques capable of disproving zero-trip loops, symbolic handling facilities to compute closed forms of recurrences, and interfaces to other compilation passes such as the data-dependence test. In comparison, the recognition phase of solving induction variables, which has received most attention so far, has in fact turned out to be relatively straightforward. This paper provides an overview of techniques described in more detail in [12].

1 Introduction

The translation of sequential programs into parallel form is a well-established discipline. An introduction to this topic can be found in [2]. Parallelizing translation techniques deal mostly with loops whose iterations can be executed in parallel if no iteration produces a data value that is consumed by another iteration. In some circumstances such *true dependences* can be removed, for example if the value to be consumed can be expressed directly as a function of the loop index variable(s). We call this an *induction variable* and the direct expression its *closed form*. Another example of removing a true dependence involves an associative and commutative operation performed across loop iterations. Since the order of such operations can be changed, iterations need not wait for the previous iteration to perform its operation, as long as we ensure exclusive access to the operand. Important operations of this type are *reductions*, e.g., accumulating values across loop iterations.

Both induction and reduction idioms are recognized by available parallelizing compilers if they have simple forms. In our previous experiments [7, 8] we have seen the need for recognizing more general forms of these patterns, notably induction variables in triangular loops and in multiplicative expressions, and reduction operations on arrays in *histogram* form. We discuss these variants below.

We have implemented new and powerful idiom recognition and solution techniques in the Polaris compiler [5, 6] which contribute towards substantial performance improvements over previous generations of parallelizing compilers.

* This work is supported by the National Security Agency and by U.S. Army contract #DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

2 Induction Variable Substitution

The following example shows a *triangular* induction variable that occurs inside a triangular loop nest (the inner loop bound depends on the outer loop index). The nest can be translated into parallel form as shown. Available compilers typically are able to substitute the induction variable in the inner loop only.

```
iv = 0
do i = 1, n                          do parallel i = 1, n
   do j = 1, i                          do parallel j = 1, m
      a(iv) = ...          ⇒               a(j + (i^2 - i)/2 - 1) = ...
      iv = iv + 1                       enddo
   enddo                             enddo
enddo
```

Our *generalized induction variable* algorithm performs the complete translation in two steps. The first step recognizes induction variables that match the statement pattern $iv = iv + inc\_expression$ where $inc\_expression$ can contain outer loop indices, other induction variables, and loop-invariant terms. Induction variables may also appear in more than one induction statement, at different levels of a loop nest, as exemplified in section 2.2.

The closed form of the induction variable is computed in the second step. In the following discussion we consider a loop nest containing an induction variable iv. Loop L has n iterations and the variable $iter$ denotes the current iteration. For the sake of simplicity we treat iteration spaces from 1 to n (this is easily derived from the actual loop bounds and step). The first and last statements of loop L are labeled $loopentry$ and $loopexit$, respectively.

Definition 1

Given a loop L, the total increment $\psi_{iv}^{L}(iter)$ of the additive induction variable iv in a single iteration of the body of L is defined as:

$$\psi^L_{iv}(iter) = \sum_{s=loop\,entry}^{loop\,exit} increments\ to\ iv\ in\ s$$

Here s iterates over the statements in loop L from *loop entry* to *loop exit*. The increment to iv is zero if s is not an induction statement. Inner loops are considered a single (compound) statement and their increment is determined using Definition 3.

Definition 2

The *increment at the beginning of iteration* i of the additive induction variable iv is defined as:

$$i@i^L_{iv} = \sum_{iter=1}^{i-1} \psi^L_{iv}(iter)$$

Here $\psi^L_{iv}(iter)$ is summed over the iteration space of L from the first to the $i-1$th iteration.

Definition 3

The *increment* of the additive induction variable iv upon completion of loop L is defined as:

$$i@n^L_{iv} = \sum_{iter=1}^{n} \psi^L_{iv}(iter)$$

Here we are summing $\psi^L_{iv}(iter)$ over the entire iteration space of L from 1 to n. Similar definitions hold for multiplicative induction variables. We refer the reader to the more detailed description[12].

Initially, ψ is computed by a lexical scan which forms a symbolic sum of the right hand side increments at all induction sites of a given induction variable iv. Once ψ has been computed for a given iv, the symbolic expressions $i@i$ and $i@n$ are computed using a symbolic sum function based on Bernoulli numbers [13].

The algorithm proceeds recursively when an inner loop is encountered. The current value for $i@i$ in the enclosing loop is used as the initial value of the iv at entry to the nested loop. The sum ψ is then determined for the inner loop (recursing again as necessary), and the expression $i@i$ is formed for the inner loop. In other words, when iv occurs inside a nested loop, ψ can only be partially calculated for outer loops and becomes fully known in these outer loops only as $i@n$ is recursively returned from inner loop sums.

As the recursion unwinds and loop exits are encountered, the expressions $i@n$ are calculated. These are the last values which will be used outside the loop if the induction variables' live range extends beyond the loop exit.

The overall picture is one of a statement-wise traversal of a given loop body, first summing increments, then multiplying across iteration spaces, all done recursively as necessary.

The final substitution step is straightforward. Briefly, traversing from outside in, each loop header is annotated with the value of the induction variable at entry to the loop. The same annotation is made at the site of each use of the induction variable, including any increments encountered in the body of the loop up to the point of use. Our current implementation does a direct substitution of the closed form.

The algorithm works similarly for multiplicative induction variables. Instead of sum operations, however, products are formed. Again, we refer the reader to [12] for additional detail.

2.1 Zero Trip Test

If the upper bound UB of a loop is less than the lower bound LB, then the loop is not executed, and as a result the computation of the closed form of an induction variable must include analysis techniques capable of detecting zero-trip loops. Our algorithm handles this problem with new techniques termed *Symbolic Range Propagation* [4]. We prove in all cases save one of our application test suite that loops do not have zero trips and hence the transformation is correct. The remaining case requires the use of interprocedural range propagation, which has not yet been implemented. In addition, we have determined that in fact it is sufficient that loops meet the weaker requirement $UB \geq LB-1$. Expressed in terms of Definition 3 above, for example, $UB \geq LB-1 \Rightarrow$ that $i@n^L_{iv}$ is correct. In words this means simply that the last value iv_{last} of induction variable iv in loop L is correct given $UB_L \geq LB_L - 1$. Consider the following example:

```
iv = 0
do i = 1, n
    do j = 1, m                    do parallel i = 1, n
       iv = iv + 2          ⇒          do parallel j = 1, m
       ...                                ...
    enddo                              enddo
    iv = iv + 1                        a(i + 2 * i * m) = ...
    a(iv) = ...                    enddo
enddo
```

Intuitively, one concludes that the condition $m \geq 1$ insures the correctness of the transformation; in fact the transformation is still correct for $m = 0$. We have termed this an *exact zero trip* and have found it important in our application test suite.

2.2 Wrap-Around Variables

A *wrap-around variable* is classically defined as a variable that takes on the value of an induction variable after one iteration of a loop [11]. There is one important case in our application test suite where the recognition of wrap-around loop bounds is a necessary precursor to the solution of an induction variable:

```
m = 0
do i = 1, n
    do j = 1, i
       lb = j
       ub = i
       do k = i, n
          do l = lb, ub
             m = m + 1
             a(m) = ...
          enddo
          lb = 1
          ub = k + 1
       enddo
       m = m + i
    enddo
enddo
```

Note the presence of the loop-variant wrap-around variables lb and ub on the l loop, which must be solved in order to determine the closed form for the induction variable m. After the induction transformation we have:

```
m = 0
do parallel i = 1, n
   do parallel j = 1, i
      do parallel l = j, i
      private m
         m = l + (i + (-9i^2 - 3i^4 + 6i + 6i^3 - 6in -
```

```
                6in^2 + 6ni^2 + 6i^2n^2)/4 - 3n-
                3j^2 - n^2 + 2i^3 + 3j + 3i^2 - 3ij-
                3ji^2 + 3jn + 3jn^2)/6 - 2i + 2ij
            a(m) = ...
        enddo
        do parallel k = 1 + i, n
            do parallel l = 1, k
            private m
                m = l + ((-9i^2 - 3i^4 + 6i + 6i^3 - 6in-
                        6in^2 + 6ni^2 + 6i^2n^2)/4 - 3k - 3n-
                        3j^2 - 3n^2 - 2i + 2i^3 + 3j + 3k^2-
                        3ij - 3ji^2 + 3jn + 3jn^2)/6 - i + 2ij
                a(m) = ...
            enddo
        enddo
    enddo
enddo
```

Employing powerful symbolic manipulation, the induction pass first recognized and then removed the wrap-around loop bounds *lb* and *ub* by peeling the first iteration of the k loop, allowing the solution of the induction on m within the outermost i loop. Note that the k loop now executes an *exact zero trip* when $i = n$, and that the correctness of this transformation has been proven using the techniques described in section 2.1.

As noted in section 2, the value of m has been substituted directly on the right hand side of each remaining induction site, and m is now a loop private variable[1]. As is clear from the example, direct substitution introduces unnecessary overhead in inner loops. For this reason we are currently implementing an on-demand substitution algorithm which incorporates strength reduction of complex expressions of this nature.

3 Reduction Recognition

The following example represents an important pattern of reduction operations that we have found in the analysis of real application programs:

```
do i = 1, n
    do j = 1, n - i
        ...
        fx(i) = fx(i) + exp1
        fx(i + j) = fx(i + j) + exp2
        ...
    enddo
enddo
```

The loop nest accumulates into the array FX. Different iterations accumulate into different array elements. This is what we term a *histogram reduction*. The Polaris-transformed code is shown below:

```
do parallel i = 1, n
    do j = 1, cpus
        fx0(i, j) = 0
    enddo
enddo
do parallel i = 1, n
private p = procid
    do j = 1, n - i
        ...
        fx0(i, p) = fx0(i, p) + exp1
        fx0(i + j, p) = fx0(i + j, p) + exp2
        ...
    enddo
enddo
do parallel i = 1, n
    do j = 1, cpus
        fx(i) = fx(i) + fx0(i, j)
    enddo
enddo
```

3.1 Recognition Pass

The algorithm for recognizing reductions searches for assignment statements within a given loop of the form:

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

where β represents an arbitrary expression and A may be a multi-dimensional array with subscript vector $\{\alpha_1, \alpha_2, \ldots\}$ which may contain both loop-variant and invariant terms. Neither α_i nor β may contain a reference to A, and A must not be referenced elsewhere in the loop outside other reduction statements. Of course $\{\alpha_1, \alpha_2, \ldots\}$ may be null (i.e., A is a scalar variable).

The algorithm recognizes potential reductions which fall into the two classes of *histogram reductions* (where one of the subscript dimensions, e.g., α_1, is loop-variant) and *single-address reductions* (where all dimensions are loop-invariant). The reduction recognition pass of Polaris is based on powerful pattern matching primitives that are part of the Polaris FORBOL environment [14].

In the recognition pass, variables that match the above pattern are flagged as potential reduction variables.

3.2 Data Dependence Pass

The data dependence pass analyzes candidate reduction variables. If it can prove independence, then it removes the reduction flag. This situation occurs if all loop iterations accumulate into separate elements of an array.

3.3 Transformation Pass

In the transformation stage, three different types of parallel reductions can be generated. They are termed *blocked*, *privatized*, and *expanded*. All three schemes take advantage of the fact that the sum operation is mathematically commutative and associative, and thus the accumulation statement sequence can be reordered. Although it is well known that this can lead to roundoff errors, we have not found this to be a problem in our experiments. Polaris includes a switch that allows the user to disable the transformation, which is the common way of dealing with this issue in many compilers.

The first scheme, termed *blocked*, involves the insertion of synchronization primitives around each reduction statement, making the sum operation atomic. In our example the reduction statements could simply be enclosed by lock/unlock pairs, allowing the loop to be executed in parallel. This is an elegant solution where the architecture provides fast synchronization primitives. However, in the machines used in our experiments the synchronization overhead was high, and as a result we focussed our efforts on the remaining methods.

In *privatized* reductions, duplicates of the reduction variable that are private to each processor are created and used in the reduction statements in place of the original variable. The partial sums accumulated by these variables are initialized to zero at loop entry. In this way, the loop can now be executed in a parallel doall fashion without the need for synchronization other than the sum across processors executed prior to final loop exit.

[1] Note also that the third induction site just inside the outermost i loop has been automatically removed

Scheme three is used in our example and it is termed *expanded* reductions. It is similar to the second scheme, but collects the partial sums in a shared array that has an additional dimension equal to the number of processors executing in parallel. All reduction variables are replaced by references to this new, global array, and the newly created dimension is indexed by the processor number executing the current iteration. The transformed loop may be executed fully in parallel. The partial sums are combined in a separate loop following the original one. For histogram reductions this loop can be executed in parallel as well, as shown in the example above.

One thorny issue dealt with in implementing methods two and three was the determination of the size of the partial sum variables for histogram reductions. Often it is not possible to determine this size from the code surrounding the loop. Because of this we again took good advantage of *Symbolic Range Propagation* [4] to determine these sizes.

4 Performance Results

Table 1 shows the speedups we have obtained on an 8-processor set on an SGI Challenge (R4400), relative to the serial execution speed for three codes in our application suite. We measured the overall speedup due to Polaris optimizations, and the diminished speedups after disabling the induction and reduction transformations, respectively. For comparison purposes, the benchmarks under discussion were also compiled and executed using the commercially available SGI parallelizer, Power Fortran Accelerator (PFA), with additional non-default optimization options -ag=a and -r=3.

	MDG	TRFD	TURB3D
serial	1	1	1
PFA	1.0	0.2	4.3
Polaris	5.1	2.8	4.9
(no reduction)	0.2	2.8	3.5
(no induction)	0.1	0.1	4.9

Table 1: Overall Program Speedups

In all programs the Polaris compiler achieves significant speedups. In TURB3D this is about 9% greater than PFA, and in the other programs PFA achieves little or no speedup[2]. The speedups are a result of the transformations discussed in this paper in conjunction with additional techniques described in [6]. As shown by the table entries *(no reduction)* and *(no induction)* above, in all cases the performance drops substantially when these compiler capabilities are switched off.

The performance figures become more clear when considering the most time-consuming loops of the test suite. Table 2 summarizes the idiom recognition techniques applied to these loops and the resulting loop speedups. These results were obtained using a loop-by-loop instrumentation facility built into the Polaris compiler. Column *% serial* shows the percentage time that each loop takes in the serial program execution. All of these loops would be serial without the new capabilities described in this paper.

In Table 2, HR means a histogram reduction was solved in this loop, R, a reduction; IV, an induction variable; GIV, a generalized induction variable; and W, a wrap-around variable.

[2] In fact, results from more recent runs of Polaris which incorporate full inlining bring the speedup figures for TURB3D up to 6

loop	transformation	speedup	% serial
MDG			
interf_do1000	IV, HR	7.6	95
poteng_do2000	IV, R	7.5	2
TRFD			
olda_do100	GIV	4.2	65
olda_do300	GIV, W	3.7	28
TURB3D			
enr_do2	IV, R	6.8	9

Table 2: Key Loop Timings

In the program MDG, subroutine `interf`, loop do_1000 contains multiple induction variables as well as histogram reductions. Polaris first solved the inductions, then applied *expanded reductions* to generate parallel code for the loop. Interf_do1000 is by far the most time consuming loop in the program. Another loop, poteng_do2000, is similar in nature; however, as noted, it contains single-address reduction operations. Polaris flagged these reductions with directives, which were then processed by the back end compiler on the SGI.

In TRFD subroutine OLDA, two loops dominate program execution, do_100 and do_300. Both loops exercise Polaris' generalized induction variable substitution capabilities. Do_300 contains an induction variable that is dependent on a second, *triangular* induction variable. We have termed this a *coupled* induction [13]. The *coupled* induction variable occurs at several nesting levels of nested triangular loops (the innermost site is in a quadruply nested, doubly triangular loop). As a result, the subscript expressions introduced by the substitution pass are non-linear and cannot be handled by previously known data-dependence tests. The new Polaris data dependence test that handles such expressions is described in [3].

Olda_do300 further contains two *wrap-around* variables that need to be recognized in order to solve the inductions properly. This transformation was described in section 2.2. Both loops also contain reduction operations; however, they occur in inner loops that were not parallelized due to the single-level parallelism supported by the SGI FORTRAN compiler.

Loop ENR_do2 contains a *single-address reduction* in a quadruply nested loop. Similar to poteng_do2000, this reduction is flagged by Polaris and solved via privatization by the back end SGI compiler.

5 Related Work

Three notable related contributions to idiom recognition techniques deal with the recognition of induction variables. Haghighat and Polychronopoulos [10] symbolically execute loops and inspect the sequence of values assumed by variables. By taking the difference of consecutive values, the difference of the difference, etc., they can interpolate a polynomial representing the closed form of an induction variable.

Harrison and Ammarguellat [1] do *abstract interpretation* of loops to capture the effect of a single iteration on each variable assigned in the loop. The effect is then matched with pre-defined templates that express recurrence relations, including induction variables of various forms.

Gerlek *et al* recognize induction variables by finding certain patterns in the *Static Single Assignment* representation of the program [9]. They compute the closed form of an induction variable based on a classification of the specific SSA pattern.

All three approaches provide some of the functionality necessary to handle the generalized forms of induction variables that we have seen in real programs. Potentially, they can recognize complex control structures, such as induction variables assigned in different branches of IF statements, which we have not implemented in our algorithm. However, Polaris is the only compiler that has delivered the proof of concept in that it has demonstrated that new induction and reduction variable techniques can contribute to real speedups of real programs. In doing this we have identified necessary parts of both induction and reduction variable transformation techniques that others have ignored or missed: symbolic manipulation tools for generating closed forms of induction variables, program analysis capabilities for determining and disproving zero-trip loops, combining induction variable recognition with wrap-around variable removal, symbolically determining ranges of histogram reductions, and complementing non-linear data-dependence tests. Some of these capabilities are supported by other passes available in the Polaris compiler. Defining proper interfaces between induction and reduction variable handling and these passes is another distinguishing feature of the compiler.

6 Conclusion

The new Polaris compiler is able to gain significant speedups in real programs, and demonstrates the parallelization technology necessary to do so. Two crucial techniques have been described in this paper: the recognition and transformation of induction variables and the parallelization of reduction operations. Our implementation and measurements confirm the great importance of these idiom recognition techniques.

We have demonstrated that the implementation of the necessary generalized forms of induction variable and reduction handling techniques is feasible in a parallelizing compiler and that it is possible to integrate them in such a way that the resulting compiler actually achieves substantial performance results on real programs. This demonstration has not been made previously, and it is particularly significant in that some of the transformations introduce complexities in the generated code (i.e., non-linear expressions) that cannot be readily processed by other compilation passes.

In our evaluation we have also determined that the issue of recognizing complex recurrence patterns and control-flow structures, which was given much attention in related work, was not of primary concern. Instead we have found other issues to be of critical importance, namely, the provision of symbolic computation capabilities for handling closed forms of recurrences, determining iteration counts and variable ranges, and the integration of the idiom recognition pass with other compilation techniques.

References

[1] Zahira Ammarguellat and Luddy Harrison. Automatic Recognition of Induction & Recurrence Relations by Abstract Interpretation. *Proceedings of Sigplan 1990, Yorktown Heights*, 25(6):283–295, June 1990.

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

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

[4] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proceedings of the 9th International Parallel Processing Symposium, April 1995*, October 1994.

[5] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York; also: Lecture Notes in Computer Science 892, Springer-Verlag*, pages 141–154, August 1994.

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

[7] 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.

[8] Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the Automatic Parallelization of the Perfect Benchmarks. Technical Report 1392, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., December 1994.

[9] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *To appear in TOPLAS 95.*

[10] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic Program Analysis and Optimization for Parallelizing Compilers. *Presented at the 5th Annual Workshop on Languages and Compilers for Parallel Computing, New Haven, CT*, August 3-5, 1992.

[11] D. Padua and M. Wolfe. Advanced Compiler Optimization for Supercomputers. *CACM*, 29(12):1184–1201, December, 1986.

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

[13] William Morton Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, December 1994.

[14] Stephen Weatherford. High-Level Pattern-Matching Extensions to C++ for Fortran Program Manipulation in Polaris. Master's thesis, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., May 1994.