Parallelization in the Presence of Generalized Induction and Reduction Variables

Bill Pottenger and Rudolf Eigenmann
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign*

Abstract

The elimination of induction variables and the parallelization of reductions in FORTRAN programs has been shown to be integral to performance improvement on parallel computers [9, 10]. As part of the Polaris project, compiler passes that recognize these idioms have been implemented and evaluated. Developing these techniques to the point necessary for achieving significant speedups on real applications has prompted solutions to problems that have not been addressed in previous reports. These include analysis capabilities to disprove 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 analysis phase of induction variables, which has received most attention so far, has turned out to be relatively straightforward.

1 Introduction

The parallelization of loops requires resolution of many types of data dependences ([2], [5], [17]). In particular, cross-iteration dependences caused by inductions may prohibit parallel execution. Induction variable substitution is an important technique for resolving certain classes of such dependences. In addition, induction solution provides the environment within which privatization of arrays can take place. In the presence of induction variables in array subscripts, induction solution is a prerequisite to the resolution of cross-iteration dependences on array accesses. When combined with array privatization, induction variable substitution becomes a powerful tool for removing cross-iteration dependences [18].

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 three additional important cases: one, when induction variables appear in the right hand side increment of other induction variables, termed *coupled induction variables*; two, when induction variables occur within *triangular* loop nests[1]; and three, when inductions form a geometric series, termed *multiplicative inductions.*

When an anti-output-flow dependence occurs which cannot be solved by induction substitution there is still opportunity for parallelism using techniques for solving reductions in parallel. Current compilers are able to solve simple scalar reductions and in some cases, single dimensional array reductions with invariant indices. Based on work completed in [10], however, two additional classes of reductions were determined important: *single address reductions*, which occur on a scalar or on an array of one or more dimensions with loop invariant indices; and *histogram reductions*, which occur on arrays with loop variant indices.

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

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

2 Problem Presentation

2.1 What is Induction Variable Substitution?

A class of inductions important to the parallelization of the the Perfect Club Benchmarks®[3] are termed *coupled inductions*. For example:

$K1 = 0$
$K2 = 0$
Do $I = 1, N$
    Do $J = 1, I$
        $K1 = K1 + 1$
        $A(K2) = 0$
    EndDo
    $K2 = K2 + K1$
EndDo

Here the induction variable $K2$ is dependent on the *triangular* induction variable $K1$. After induction variable substitution, the above becomes:

$K1 = 0$
$K2 = 0$
DoAll $I = 1, N$
    DoAll $J = 1, I$
        $K1_{private} = J + (I^2 - I)/2$
        $A(((I^3 + 2 * I)/3 - I)/2) = 0$
    EndDo
    $K2_{private} = ((I^3 + 2 * I)/3 - I)/2$
EndDo

A second class of inductions determined important in [10] involve *multiplicative inductions*[2]. For example:

$K = 1$
Do $I = 1, N$
        $K = K * 2$
        $A(K) = 0$
EndDo

After induction solution, the above becomes:

$K = 1$
DoAll $I = 1, N$
        $K_{private} = 2 * 2^{((-1)+I)}$
        $A(K_{private}) = 0$
EndDo

Collectively, these classes of inductions are a subset of a larger class termed *Generalized Induction Variables*, or *GIVs*.

[2] Important in the Perfect Club Benchmark code OCEAN [14]

2.2 What is Reduction Recognition?

Consider the following example code where A may be either scalar or multidimensional, α may be a multivariate function of the loop indice(s), and β is any symbolic expression:[3]:

Do $I = 1, N$
  $\ldots$
  $A(\alpha(i, \ldots)) = A(\alpha(i, \ldots)) + \beta$
  $\ldots$
EndDo

Given that A is not referenced in either α or β, these reductions can be solved in several ways, one of which we term *privatized reductions*:

DoAll $I = 1, N$
  begin preamble
    $lb = min(\alpha(1 : n))$
    $ub = max(\alpha(1 : n))$
    $A_{private}(lb : ub) = 0$
  end preamble
  $\ldots$
  $A_{private}(\alpha(i, \ldots)) = A_{private}(\alpha(i, \ldots)) + \beta$
  $\ldots$
  begin postamble
    begin critical section
      $A(lb : ub) = A(lb : ub) + A_{private}(lb : ub)$
    end critical section
  end postamble
EndDo

Here we have defined a prologue to the loop termed a *preamble* which contains code executed once per participating processor. Likewise, there is a *postamble* which is executed by each processor upon completion of its slice of the iteration space. The array $A_{private}$ is initialized in the preamble from lb to ub. The total size of this range is determined symbolically based on the access pattern to the array A. Similarly, A is reduced across processors in the postamble.

Additional speedup comes from the fact that the accesses to $A_{private}$ will be local in nature on architectures where the memory is physically distributed[4]. On the other hand, on an architecture in which the memory is not distributed[5], *array expansion* removes the need for synchronization in the postamble:

$lb = min(\alpha(1 : n))$
$ub = max(\alpha(1 : n))$
DoAll $I = lb, ub$
  Do $J = 1, P_{max}$
    $A_{expanded}(i, j) = 0$
  EndDo
EndDo

[3] For the sake of clarity all examples shown involve single dimensional arrays
[4] e.g., the Convex Exemplar
[5] e.g., the SGI Challenge

DoAll $I = 1, N$
    ...
    $A_{expanded}(\alpha(i, \ldots), P_k) = A_{expanded}(\alpha(i, \ldots), P_k) + \beta$
    ...
EndDo
DoAll $I = lb, ub$
    Do $J = 1, P_{max}$
        $A(i) = A(i) + A_{expanded}(i, j)$
    EndDo
EndDo

Here P_k refers to the processor currently executing in a given iteration and P_{max} refers to the number of processors executing in parallel. The initialization and final reduction can both be executed in parallel.

3 Implementation

The induction substitution and reduction recognition passes are implemented on a basic infrastructure provided by the Polaris compiler development environment [7].

Within Polaris, access to the internal program representation is controlled through a data-abstraction mechanism. Operations built onto the internal representation are defined such that the programmer is prevented from violating its structure or leaving it in an incorrect state at any point during a transformation. The system also guarantees the correctness of all control flow information. This is realized through automatic incremental updates of this information as a transformation proceeds [4].

3.1 Induction Variable Substitution

The simplest case of an induction variable takes the form:

$$iv = iv + constant$$

and is handled well by existing parallelizing compilers.

In the more general case considered here, induction variables may take the form:

$$iv_i = iv_i \ \{\pm, *\} \ f(iv_j, iv_k, \ldots, invariant_1, invariant_2, \ldots, loop\_index_i, loop\_index_j, \ldots)$$

The induction variable on the left, iv_i, may be coupled with one or more induction variables when the operation is {+} or {-}[6]. The remaining portion of the right hand side increment is a multivariate function of the enclosing loop indices and other loop invariant variables. In the current implementation the presence of flow control statements other than DO cause a candidate to be rejected. The first step of our algorithm recognizes induction variables that match this pattern.

In a second step, the calculation of the closed form of an induction variable involves a sweep through the statements of the loop. The algorithm recurses upon encountering an inner loop. As the sweep proceeds through each statement, two expressions are formed that represent the increment incurred by the induction variable – relative to the loop start – at the beginning of each loop iteration and at the loop exit, respectively.

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 (from 1 to n). The first and last statements of loop L are labeled $loopentry$ and $loopexit$, respectively.

[6] i != {j,k,...}

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:

$$\psi_{iv}^{L}(iter) \quad = \sum_{s=loop\ entry}^{loop\ exit} increments\ to\ iv\ in\ statement\ s$$

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:

$$i@i_{iv}^{L} \quad = \sum_{iter=1}^{i-1} \psi_{iv}^{L}(iter)$$

Here $\psi_{iv}^{L}(iter)$ 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:

$$i@n_{iv}^{L} \quad = \sum_{iter=1}^{n} \psi_{iv}^{L}(iter)$$

Here we are summing $\psi_{iv}^{L}(iter)$ over the entire iteration space of L from 1 to n.

Similar definitions hold for multiplicative induction variables.

3.1.1 Zero Trip Test

If the upper bound of a loop is less than the lower bound then the loop does not get executed and, hence, the induction variable does not get modified. Because of this, computing the closed form as described could lead to incorrect results. For example, the following transformation is only correct for $n \geq 0$.

```
ind=0
DO i=1,n
  iv=iv+2
  a(iv)=...
ENDDO
print *,iv
```

$\Rightarrow$

```
DO i=1,n
  a(i*2) = ...
ENDDO
print *,n*2
```

Our algorithm handles this *zero trip problem* with new techniques developed in [6] known as *Symbolic Range Propagation*. Thanks to these techniques we were able to prove in all crucial cases of our application test suite that loops do not have zero trips. The notion of an *exact zero trip* is important in practice:

Definition

Given upper and lower loop bounds UB and LB, an **exact zero-trip** is executed when $UB = LB - 1$.

Intuitively, one would assume that $UB \geq LB$ needs to be true for induction variable substitution to be valid. In other words, if the loop upper bound is greater than or equal to the lower bound, then $i@n$ is correct. However, on closer inspection of the formulas it can be seen that the closed form expressions are still correct when $UB \geq LB - 1$.

As an example, consider the following loop which results from peeling the first iteration of loop 290 of OLDA_do300 in the Perfect Club Benchmark TRFD [15][7]:

```
      DO 291 K=I+1,N
       DO 281 L=1,K
        M = M + 1
281     XIJKL(M) = XKL(L)
291   CONTINUE
```

The last value $i@n_m^K$ of the induction variable m after loop K is:

$$i@n_m^K = \sum_{k'=i+1}^{n} k' = (n + n^2 - i - i^2)/2$$

When $I = N$ in the last iteration of an enclosing loop with index I (not shown), $i@n_m^K \equiv 0$, and as a result, the contribution of $i@n_m^K$ to $i@n_m^I$ is also 0. This preserves the semantics of the original untransformed K loop nest.

3.1.2 Wrap Around Variables

A *wrap around variable* is classically defined as a variable which takes on the value of an induction variable after one iteration of a loop [16]. There is at least one important case in the Perfect Club Benchmarks where a wrap around variable of this type occurs as the bound of a loop containing an induction site. This wrap around variable must be recognized as such in order to solve the induction at the level of the enclosing loop.

Loop 280 in OLDA_do300 contains an example of such a wrap around variable:

```
      LMIN=MJ
      LMAX=MI
      DO 290 MK=MI,MORB
       DO 280 ML=LMIN,LMAX
        MIJKL=MIJKL+1
280     XIJKL(MIJKL)=XKL(ML)
       LMIN=1
       LMAX=MK+1
290   CONTINUE
```

The variable $LMAX$ takes the value MI initially, and from then on takes the value of the 290 loop index MK. In addition to this wrap around variable which takes on the value of an induction variable, the lower bound of loop 280 also takes on a constant value after one iteration.

[7] The necessity of this peeling is discussed in the following section

The recognition of both of these wrap around variables can be accomplished by representing the program in SSA form and using *back substitution* [19] based on techniques discussed in [21].

Simple techniques such as these are sufficient to recognize the wrap around variable patterns that we found in our test suite. More sophisticated recognition techniques will be discussed in Section 5.

3.2 Reduction Recognition

The first step in the solution of parallel reductions is recognition of potential reductions – typically accumulation operations, such as `sum(x) = sum(x) + <expression>`. Following this, a second, data-dependence test pass analyzes these candidate reductions to determine if they are indeed reductions [5]. If the data-dependence test can prove independence (i.e., the accumulation operation uses different elements of the array `sum()` in each loop iteration), the candidate reduction flag is removed from this variable.

Finally, a transformation stage dependent on architectural considerations generates either *blocked*, *privatized*, or *expanded* reductions.

Recognition The algorithm for recognizing reductions searches for assignment statements of the form:

$$A(\alpha(i,j,k,\ldots)) = A(\alpha(i,j,k,\ldots)) \pm expression$$

where $i, j, k, \ldots$ are loop indices, A may be a multi-dimensional array, α and *expression* are any symbolic expressions, neither α nor *expression* contain a reference to A, A is not referenced elsewhere in the loop outside of definition-use pairs in other reduction statements, and α may be null (i.e., A is scalar).

The algorithm recognizes potential reductions which fall into the two classes of *histogram reductions* (where α is loop-variant) and *single address reductions* (where α is loop-invariant). The reduction recognition pass of Polaris is based on powerful pattern matching primitives that are part of the Polaris FORBOL environment [20].

Transformation 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. In practice this can lead to roundoff errors; however, in our experiments we have not found this to be a problem. In the event that roundoff does become a problem, our compiler includes a switch that allows the user to disable the transformation.

The first scheme, termed *blocked*, involves the insertion of synchronization primitives around each reduction statement, making the sum operation atomic. For example, consider the following code fragment:

```
DOALL I = 1, N
   ...
   begin critical section
      sum = sum + A(I)
   end critical section
   ...
EndDo
```

This is an elegant solution where the architecture provides fast synchronization operations. However, in the machines used in our experiments the synchronization overhead was high, and as a result we focussed our efforts on the following two methods.

In *privatized* reductions, shadow variables of the reduction variable that are private to each processor are created and used in the reduction statements instead of the original sum variable. The partial sums

accumulated by these variables are initialized to zero at loop start and summed up at the end of the loop. In this way, the loop may now be executed in a parallel doall fashion without the need for synchronization other than the sum across processors executed prior to loop exit. Furthermore, on non-uniform memory access architectures, locality of reference is improved because the partial sum variable is private to each processor. In our current implementation, the global sum at the loop end is done using synchronization primitives. An example of *privatized* reductions was discussed in Section 2.2.

The third scheme, termed *expanded* reductions 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. Similar to privatized reductions, the loop may be executed fully in parallel. This scheme does not need any synchronization in the parallel loop. Instead, 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 in Section 2.2. The following fragment extracted from the Perfect Club Benchmark code MDG exemplifies this solution method:

```
      SUBROUTINE INTERF(X, Y, Z, FX, FY, FZ, XM, YM, ZM, VIR)
      ...
      POINTER (PTR1,FX0)
      CPUVAR = MP_NUMTHREADS()
      PTR1 = MALLOC(24*CPUVAR+(-8)*NATOMS*CPUVAR+8*NATOMS*NMOL*CPUVAR)
      ...
C$DOACROSS LOCAL(I,TPINIT),SHARE(FX0)
      DO I = 1, CPUVAR, 1
       DO TPINIT = 1, 3+(-1)*NATOMS+NATOMS*NMOL, 1
        FX0(TPINIT, I) = 0.0
       ENDDO
       ...
      ENDDO
C$DOACROSS LOCAL(I,J,G110,G23,G45,GG0),SHARE(FX0)
CSRD$ LOOPLABEL 'INTERF_do1000'
      DO I = 1, NMOL1, 1
       PROCID = MP_MY_THREADNUM()+1
CSRD$ LOOPLABEL 'INTERF_do1100'
       DO J = 1, NMOL+(-1)*I, 1
        ...
 S1:     FX0(2+(-1)*NATOMS+I*NATOMS, PROCID) = FX0(2+(-1)*NATOMS+I*NATOM
     *S, PROCID)+GG0(11)+GG0(12)+G110+C1*G23
 S2:     FX0(2+(-1)*NATOMS+I*NATOMS+J*NATOMS, PROCID) = FX0(2+(-1)*NATOM
     *S+I*NATOMS+J*NATOMS, PROCID)+(-1)*GG0(13)+(-1)*GG0(14)+(-1)*G110+(
     *-1)*C1*G45
        ...
       ENDDO
      ENDDO
      DO I = 1, CPUVAR, 1
       DO TPINIT = 1, 3+(-1)*NATOMS+NATOMS*NMOL, 1
        FX(TPINIT) = FX(TPINIT)+FX0(TPINIT, I)
       ENDDO
       ...
      ENDDO
      CALL FREE(PTR1)
```

```
...
RETURN
END
```

This example is drawn from code generated by Polaris for the shared-memory multi-processor SGI Challenge. Note that the index in the first dimension of $FX0$ in statement S1 is loop invariant, whereas the same index varies in statement S2. Due to the combination of both variant and invariant indices, the reduction on $FX0$ has been classified as a *histogram reduction*. Finally, note the processor id *procid* used to index the second, expanded dimension of $FX0$ in both statements.

One issue in implementing methods two and three is to determine the size of the partial sum variables for histogram reductions. This is a non-trivial problem in symbolic analysis and is currently handled by the same techniques used in the zero-trip test, *Symbolic Range Propagation* [6]. The range accessed by each reduction statement is determined using interprocedural symbolic analysis, and the resulting ranges are merged to give an overall range. This merged range is then used to allocate the correct amount of memory for the expanded or privatized array. It is also used as the loop upper bound in the initialization and final reduction phases prior to and following the parallel loop.

4 Evaluation

In this section we will discuss two codes taken from the Perfect Club Benchmarks and one additional code which is a candidate for membership in the newly created HPSC/SPEC[8] suite.

From the Perfect Club Benchmarks, we have TRFD and MDG. TRFD is an algorithm simulating the computational aspects of a two-electron integral transformation. MDG is a molecular dynamic simulation of a flexible water molecule.

The candidate code from the HPSC/SPEC suite is TURB3D. Turb3d is a fluid dynamics code capable of solving turbulent fluid flow problems. The data set size is 64^3 elements.

Important in terms of overall program execution time, reductions occur in conjunction with inductions in two of the three codes under consideration:

Technique	MDG	Turb3d	TRFD
parallel reductions	6.3	4.9	
induction substitution	6.3		2.7

Table 1: Speedups over serial execution time on an 8 processor set on an SGI Challenge

4.1 Timing results on the SGI Challenge

The following three tables summarize the timing results for these three benchmarks.

These timings are wall-clock execution time averaged over five executions made at a maximum, non-degrading priority in timesharing mode. All parallel timing was conducted on an 8 processor processor set.

For comparison purposes, the three benchmarks under discussion were compiled and executed using the commercially available SGI product Power Fortran Accelerator, PFA [8].

Compilation commands were as follows: for the serial runs, "f77 -O2 -mips2"; for the Polaris parallel runs, "f77 -mp -O2 -mips2"; and for the PFA runs, "f77 -pfa keep -O2 -mips2"[9]. A second batch of

[8] The High Performance Steering Committee of the Standard Performance Evaluation Corporation

[9] The lower -O2 optimization was used for all compilations after I found that it produced faster code than the higher -O3 optimization level in several cases

	wallclock seconds	speedup
serial	194.03	1
SGI (PFA, default optimization)	177.20	1.10
SGI (PFA, aggressive optimization)	187.39	1.04
Polaris (with transformations)	30.41	6.38
Polaris (without reduction transformation)	1105.69	0.18
Polaris (without induction transformation)	1671.40	0.12

Table 2: Overall Program Timings and Speedup for MDG

	wallclock seconds	speedup
serial	21.15	1
SGI (PFA, default optimization)	95.79	0.22
SGI (PFA, aggressive optimization)	108.71	0.20
Polaris (with both transformations)	7.61	2.78
Polaris (without reduction transformation)	7.67	2.76
Polaris (without induction transformation)	226.08	0.09

Table 3: Overall Program Timings and Speedup for TRFD

	wallclock seconds	speedup
serial	156.72	1
SGI (PFA, default optimization)	45.84	3.42
SGI (PFA, aggressive optimization)	36.20	4.33
Polaris (with reduction transformation)	31.87	4.92
Polaris (without reduction transformation)	44.70	3.51

Table 4: Overall Program Timings and Speedup for TURB3D

runs was made using PFA with optimization switches "-ag=a -r=3" to provide a comparison with a commercial product using more aggressive optimizations[10].

4.2 Discussion of the Major Loops

The following table shows the total amount of wall-clock time in seconds spent in the major loops of these three codes. These results were obtained using a loop-by-loop instrumentation facility that is built into the Polaris compiler.

loop	serial	parallel	speedup
MDG-INTERF_do1000	175.192	22.920	7.644
MDG-POTENG_do2000	13.534	1.816	7.453
TRFD-OLDA_do100	13.820	3.310	4.175
TRFD-OLDA_do300	5.988	1.618	3.70
TURB3D-ENR_do2	13.780	2.038	6.76

Table 5: Loop timings for key loops in MDG, TRFD, and TURB3D

MDG-INTERF_do1000 This is the outermost loop of the subroutine INTERF, and consumes over 90% of the execution time. It is a triangular loop nest containing 18 sites of combined *histogram* and *single-address* reductions on three different arrays indexed by six different additive induction variables.

The near linear speedup achieved for this loop is attributable to the solution of the additive inductions in conjunction with reduction recognition. Polaris has applied *expanded reductions* for generating parallel code, as described in Section 3.2.

In order to show the importance of these two techniques separately, timings were made by turning off each optimization in turn. The results are displayed in Table 2. In both cases, when either parallel reduction solution or induction variable substitution is turned off, this loop is not parallelized and no speedup results.

It is interesting to note that the execution time for solving only reductions was 33% longer than that when solving only inductions. This is because several small trip count loops in do1000 containing reductions were parallelized, incurring a large degree of overhead. These inner loops are not parallelized when do1000 is parallelized. Nevertheless, heuristics to handle such situations better are under development.

MDG-POTENG_do2000 This loop accounts for about 2% of the execution time and becomes important once the loop INTERF_do1000 is parallel. It too has a very good speedup. Like INTERF_do1000, this loop contains a combination of six interdependent additive inductions, the solution of which is a prerequisite to doall parallelization.

The performance impacts of the two transformations individually is similar to those in loop INTERF_do1000.

TRFD-OLDA_do100 This is the first loop of three contained in the subroutine OLDA, and consumes over 65% of the execution time. This loop contains three induction variables as well as *histogram reductions* which occur at two different sites inside inner loops. Polaris succeeds in parallelizing this loop – in part due to the solution of these complex *triangular* inductions. Consequently, the inner loops can be executed serially and no further reduction processing is necessary.

The separate timings resulting from turning off each optimization in turn are displayed in Table 3. Although do100 does contain reductions, these only become important if the inductions are not solved.

[10] The code was actually preprocessed with the command "pfa -ag=a -r=3" and then compiled with the same f77 command as above

However, although these reductions are solved when induction solution is turned off, the given data set does not provide enough work for these parallel loops to offset the cost of parallel execution. A 10-fold slowdown would result.

TRFD-OLDA_do300 In OLDA_do300 the most complex *coupled, triangular* inductions occur. Responsible for 28% of the execution time of the benchmark, OLDA_do300 contains three induction variables, one *triangular*, one *doubly triangular*, and one *coupled, doubly triangular* induction variable. This loop also contains the wrap-around loop bounds discussed in Section 3.1.2. In addition, similar to OLDA_do100, two *histogram reductions* occur in this loop. Again, as in OLDA_do100, reduction operations occur in inner loops that can be executed serially after parallelizing OLDA_do300.

The speedup of these two loops is less than those in MDG-INTERF. One potential cause is the complexity of the closed-form expressions used to index arrays in inner loops. While not confirmed at this point, applying strength reduction and loop invariant hoisting to those loops which are *not* chosen to execute in parallel will likely reduce this overhead.

TURB3D-ENR_do2 The final loop under consideration occurs in subroutine ENR_do2, and accounts for about 8.8% of the execution time of the benchmark. It contains a *single-address reduction* in a quadruply nested loop. On the SGI Challenge machine, *single-address reductions* may be flagged with a directive, and are solved as *privatized reductions* by the back-end code generator. The reduction recognition pass in Polaris flagged this reduction, and the Polaris postpass generated the necessary SGI directive to allow the solution in a privatized doall fashion.

From the comparison with the timing made with parallel reductions turned off, it is clear that the solution of this reduction contributes significantly to the speedups achieved.

It is also interesting to note in Table 4 that this same reduction was solved by PFA using the more aggressive compilation options "-ag=a -r=3", which allow transformations based on associativity.

5 Related Work

Induction variable substitution and parallel reductions were recognized as two of four performance-critical techniques necessary for the parallelization of the Perfect Club Benchmarks [10]. Several related approaches to the solution of these two problems have been presented in the literature. At a high level, all these approaches fall under the category of symbolic analysis.

Symbolic Execution Haghighat and Polychronopoulos symbolically execute loops and use finite difference methods in conjunction with interpolation to determine the closed form for a given recurrence [12]. The scheme executes a loop and captures differences in values assumed by candidate induction variables from one iteration to the next. Differences of differences are taken as well – recursively until a constant is obtained at depth d. These differences are then interpolated to find the closed form of the induction variable, where the order of the derived polynomial is given by d.

The strength of this approach lies in the ability of the algorithm to handle a variety of both induction variables and *generalized induction expressions* under a variety of control flow conditions.

Abstract Interpretation Harrison and Ammarguellat's approach [1] is based on *abstract interpretation*, and represents syntactic constructs as maps from variables to expressions which encapsulate the state prior to entry to the construct. Termed an *abstract store*, this map summarizes the net effect of a single iteration of a given loop on each variable assigned in the loop.

Once the abstract stores have been composed across a loop body, the maps are unified with predefined templates that express various recurrence relations. A simple example of such a template is $X \mapsto X \bullet K$. An induction such as $m = m + 1$ would bind $X \rightarrow m, \bullet \rightarrow +$, and $K \rightarrow 1$. The template

contains a predefined closed form which is, in this case, $X \bullet (K \bullet' u)$, where u is the upper bound of the loop and $\bullet'$ repeats the + operation u times (equivalent to the $K * u$).

One of the strengths of this approach lie in its capability to recognize inductions which cross conditional control flow structures. However, at the time of writing no capability was present to handle nested loops, and it has been determined that no further work has been conducted in this direction [13].

A second feature of this approach is the unification of recurrences with predefined templates containing closed forms. The authors make the point that "This allows the scheme to be tailored to the recognition of a variety of recurrence relations, appropriate to the particular language or applications being compiled."

SSA Based Classification Wolfe *et al* base their approach on the *Static Single Assignment* representation of the program [11]. The algorithm breaks induction variable substitution down into two phases similar to the first two phases of our algorithm. During the recognition (and classification) stage, the SSA form of the graph is searched for patterns representing recurrences. A variant of Tarjan's algorithm is used to detect strongly connected components (SCCs) in the SSA graph. An SCC which carries a variable "around the loop" represents a *sequence variable*. In the base case, a *basic linear induction variable*, for example, is detected when four conditions are met: first, it occurs in an SCC with a single μ function; second, the SCC is composed of addition or subtraction of loop invariants and other linear variables (in conjunction with loads and stores); third, the induction variable occurs only once in the right hand side expression of the induction statement; and fourth, no ϕ functions occur in the SCC.

The solution method employed for basic linear induction variables is multiplication by the *basic loop counter* h_l, which has an initial value of 0 and is incremented by 1 each iteration. Other more sophisticated methods are employed for *polynomial* and *geometric* induction variables[11], varying from matrix inversion to symbolic execution of the loop body.

The strength of Wolfe *et al* lies in their classification scheme. As noted above, they have divided the types of sequence variables into several disjoint classes. The schemes employed to classify a given variable are based on the compilers' internal knowledge of classes of sequence variables matched against the SCC under investigation. This scheme also is able to recognize the wrap around variables discussed in Section 3.1.2.

As of the date of this writing, Wolfe *et al* are in the process of implementing the techniques necessary to solve and substitute inductions of the form discussed in Section 3.1 [22].

Our approach Our approach differs from these schemes in its emphasis on the complementing capabilities an idiom substitution pass needs to have in order to achieve substantial performance gains in real programs. While the related schemes describe sophisticated mechanisms to recognize induction variables, we have found simple recognition schemes to be sufficient in our test suite. Instead, we have found additional capabilities to be of primary importance: 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, and symbolically determining ranges of histogram reductions. Some of these capabilities are supported by other passes available in the Polaris compiler, and hence the integration of the idiom recognition phase with these passes was a crucial issue.

6 Conclusion

The goal of the Polaris project has been to parallelize real codes and demonstrate the parallelization technology needed to achieve significant speedups on architectures that provide a global address space. To that end we have focussed on two techniques which were identified as important in [10] – induction variable substitution and parallel reductions.

[11] The induction variable types we have termed *triangular* and *multiplicative* fall into these two classes

Our implementation confirms the findings of these early studies; namely, that idiom recognition techniques are crucial to good performance of real applications.

Furthermore 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, 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] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer, Boston, MA, 1988.

[3] 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.

[4] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The Next Generation in Parallelizing Compilers. *Proceedings of the Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York*, pages 10.1 – 10.18, August 1994.

[5] 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.

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

[7] 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.

[8] SGI Documentation. Fortran 77 Programmer's Guide and associated man pages. Technical report, Silicon Graphics, Inc., 1994.

[9] 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.

[10] 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.

[11] 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.*

[12] 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.

[13] Luddy Harrison. Personal communication with author, 1994.

[14] Jay Hoeflinger. Automatic Parallelization and Manual Improvements of the Perfect Club Program OCEAN for Cedar. Technical Report 1246, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., July 1992.

[15] Jay Hoeflinger. Automatic Parallelization and Manual Improvements of the Perfect Club Program TRFD for Cedar. Technical Report 1247, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., July 1992.

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

[17] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. *Supercomputing '91*, 1991.

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

[19] 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.

[20] 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.

[21] Michael Wolfe. Beyond induction variables. *Proceedings of ACM/SIGPLAN PLDI*, pages 162–174, June 1992.

[22] Michael Wolfe and Michael Gerlek. Personal communication with authors, 1994.