INDUCTION VARIABLE SUBSTITUTION AND REDUCTION RECOGNITION IN THE POLARIS PARALLELIZING COMPILER

BY

WILLIAM MORTON POTTENGER

B.S., University of Alaska, Fairbanks, 1989
B.A., Lehigh University, 1980

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ABSTRACT

The elimination of induction variables and the parallelization of reductions in FORTRAN codes is integral to performance improvement on shared-memory multi-processors ([11]). As part of the Polaris Project, compiler passes which do induction variable substitution and reduction recognition have been implemented and evaluated.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Rudi Eigenmann, for his guidance and suggestions. His support and insight have been invaluable throughout the process of conducting this research. I am also grateful to Jay Hoeflinger for his tireless assistance and to Jee Ku for his assistance with the reduction back end.

Finally, I'd like to thank my wife, Byunghee, and daughters, Jessica and Becky, for their continuing support and patience with their Daddy.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The parallelization of loops requires resolution of many types of data dependences ([5], [22], [2]). 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 which index arrays, 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 ([23]).

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 have termed these *coupled induction variables*), and two, when induction variables occur within *triangular* loop nests[1].

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

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

invariant indices. Based on work completed in [11], 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.

The following table outlines the slowdowns experienced when the above techniques are not implemented ([11], page 3):

Technique	ADM	ARC2D	BDNA	DYFESM	FLO52	MDG	MG3D	OCEAN	QCD	SPEC77	TRACK	TRFD
induction substitution								8.3[a]				12.7
parallel reductions	[b]		3.3	2.1	1.1	21	15.2			3.4		[c]

Table 1.1: Performance loss from disabling individual restructuring techniques in terms of factor increase of the best execution time

[a]This is a speedup cited from [10]

[b]ADM contains reductions in significant loops. Most of them can be parallelized using existing synchronization techniques without excessive overhead

[c]TRFD contains accumulation operations that would become important for parallelization if advanced induction variable substitution and array privatization were not available.

The results presented in the above table are based on experiments conducted on the the Alliant/FX80 and Cedar multi-processors. In these experiments, the Perfect Club Benchmark codes ([3]) were manually parallelized, and speedups for various transformations measured, including induction variable substitution and parallel reductions.

CHAPTER 2

PROBLEM PRESENTATION

2.1 What is Induction Variable Substitution?

Consider the following example code segment:

```
$K = 0$
Do $I = 1, N$
    $K = K + 1$
    $A(K) = 0$
EndDo
```

Due to loop-carried flow, anti, and output dependences on K, this loop cannot be executed in parallel. However, it is possible to automatically solve the recurrence relation represented by the induction on K as follows:

```
DOALL $I = 1, N$
    $K_{private} = I$
    $A(K_{private}) = 0$
EndDo
```

This loop can now be executed in parallel using the processor private variable $K_{private}$. This is the simplest case of an induction variable, and serves as a basis for the more complex inductions observed in real codes.

As a second example of an induction, consider the following fragment of code:

```
$K = 0$
Do $I = 1, N$
    Do $J = 1, I$
        $K = K + 1$
        $A(K) = 0$
    EndDo
EndDo
```

This code segment is termed a *triangular loop* due to the triangular iteration space formed by the indices I and J. This pattern is quite common in many codes, including forward and back substitution for triangular systems, Gaussian elimination, LU factorization, Cholesky factorization, and QR factorization. The induction on K is termed *triangular*.

The recurrence represented by the *triangular* induction on K has the following closed form solution:

```
DOALL $I = 1, N$
    DOALL $J = 1, I$
        $K_{private} = J + (I^2 - I)/2$
        $A(K_{private}) = 0$
    EndDo
EndDo
```

A second class of inductions important to the parallelization of the aforementioned benchmarks[1] 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
```

[1]E.g., in the Perfect Club Benchmark code TRFD, subroutine OLDA [19]

Here the induction variable $K2$ is dependent on the (triangular) induction variable $K1$. The closed form of this induction is:

```
$K1 = 0$
$K2 = 0$
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
```

The third and final class of inductions involve *multiplicative inductions*[2]. For example:

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

The closed form for this example *multiplicative* induction is:

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

Collectively, we have termed these three classes of inductions *Generalized Induction Variables*, or *GIVs*.

2.2 What is Reduction Recognition?

Consider the following example code segment:

[2]Important in the Perfect Club Benchmark code OCEAN [20]

```
Do $I = 1, N$
    ...
    $sum = sum + A(I)$
    ...
EndDo
```

Due to loop-carried flow, anti, and output dependences on sum, this loop cannot be executed in parallel (note that sum is not referenced elsewhere in the loop). However, it is possible to solve this reduction by a number of methods which take better advantage of parallel hardware. One such solution is:

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

In this case *begin critical section* and *end critical section* enable parallel execution of this loop. Note that in general this kind of solution is warranted only where there is enough other work in the loop to balance the added cost of parallelization and synchronization. While this is a very simple case, it serves as a basis for the more complex reductions observed in real codes.

Consider now the following example code where A may be either scalar or multidimensional, and α and β may be multivariate functions of the loop indice(s)[3]:

```
Do $I = 1, N$
    ...
    $A(\alpha(i, \ldots)) = A(\alpha(i, \ldots)) + \beta(i, \ldots)$
    ...
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*:

[3]For the sake of clarity all examples shown involve single dimensional arrays

```
DOALL $I = 1, N$
    begin preamble
        $A_{private}(lb_p : ub_p) = 0$
    end preamble
    ...
    $A_{private}(\alpha(i_p, \ldots)) = A_{private}(\alpha(i_p, \ldots)) + \beta(i_p, \ldots)$
    ...
    begin postamble
        begin critical section
            $A(lb_p : ub_p) = A(lb_p : ub_p) + A_{private}(lb_p : ub_p)$
        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_p to ub_p, which represents the slice of the iteration space assigned to processor P. The total size of these slices is determined symbolically based on the access pattern to the array A. Similarly, A is reduced across processors in the postamble. The notation i_p on the privatized reduction statement in the main loop body represents the range of the index i for a given processor P[4].

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[5]. On the other hand, on an architecture in which the memory is not distributed[6], *array expansion* removes the need for synchronization in the postamble:

```
$lb = min(\alpha(1 : n))$
$ub = max(\alpha(1 : n))$
$A_{expanded}(lb : ub, 1 : P_{max}) = 0$
DOALL $I = 1, N$
    ...
    $A_{expanded}(\alpha(i_p, \ldots), P_k) = A_{expanded}(\alpha(i_p, \ldots), P_k) + \beta(i_p, \ldots)$
    ...
EndDo
$A(lb : ub)) = A(lb : ub) + A_{expanded}(lb : ub, 1 : P_{max})$
```

[4]The example given below in Section 3.2.5 will clarify how these reductions work in practice
[5]e.g., the Convex Exemplar
[6]e.g., the SGI Challenge

Here P_k refers to the processor currently executing in a given iteration, P_{max} refers to the number of processors executing in parallel, and i_p refers to the iteration counter in the slice of the iteration space executing on processor P_k. The initialization and final reduction are represented in triplet notation, and both of these operations can be executed in parallel[7].

In summary, the two general classes of reductions under consideration are exemplified as follows:

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

In this case, α, β, δ, and γ may be either the same or different functions. In either case, if both α and δ are loop invariant, the reductions are termed *single address reductions.*

On the other hand, if either α or δ vary within the loop (as is the case here where α and δ are functions of the enclosing loop index), the combination of reductions on A forms a *histogram reduction*[8].

[7]The Polaris reduction backend does so for the initialization section
[8]This occurs in the Perfect Club Benchmark code MDG, which is analyzed in Section 3.2.4

CHAPTER 3

IMPLEMENTATION

The induction substitution and reduction recognition passes are implemented on a basic infrastructure provided by the Polaris programming environment ([7]). This environment will be described briefly for the sake of clarity.

Within Polaris, access to the internal representation (IR) 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. Further, transformations are never allowed to let the code enter a state in which it no longer has proper FORTRAN syntax. 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]).

The two passes discussed in the paper, along with Polaris, have been implemented in C ++. C ++ provides the necessary data-abstraction for manipulating programs represented in the Polaris IR. Through personal experience the author has found implementation using Polaris straightforward. Debugging posed a bit of a problem in that GNU C ++ was used for compilation, but no insurmountable difficulties were encountered.

3.1 Induction Variable Substitution

As noted above, induction variable substitution is a necessary prerequisite to privatization. As such, the induction pass is executed before privatization in the Polaris compiler.

3.1.1 Algorithm Overview

Induction variable substitution takes place in three distinct phases. First, we have a recognition phase, in which candidate induction variables are recognized. These candidates are then pruned to include the classes of induction variables of interest. In the second phase, the induction variables are topologically sorted and closed forms symbolically calculated. Finally, the uses of the inductions are substituted with the closed forms in the third and final phase.

3.1.2 Induction Variable Substitution Algorithm

At a high level, the induction substitution algorithm is as follows:

```
For each ProgramUnit
    Normalize wrap-around loop bounds
    For each DO loop in the ProgramUnit
        Recognize candidate induction variables
        Topologically sort and prune candidate list
        For each induction variable
            Symbolically solve recurrences
        For each use of each induction variable
            Substitute the closed-form solution of the recurrence
        EndFor
    EndFor
EndFor
```

Prior to calling the induction substitution pass, a prepass is executed which normalizes control flow (e.g., removing GOTO statements). The outermost loop then iterates over all ProgramUnits in the Program[1]. Next, loops are searched for wrap-around loop bounds[2]. At the next level, we iterate over all

[1]This includes such semantic units as subroutines, functions, block data, etc.
[2]This step is described in Section 3.1.6

do loops in the current ProgramUnit. It is at this level that phase one takes place. Initially, candidate induction variables of two types are recognized: multiplicative and additive. Together, these take the form:

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

The induction variable on the left, iv_i, may be coupled with one or more induction variables when the operation is {+} or {-}[3]. 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 the candidate to be rejected.

The next step is to topologically sort the list of successful candidates in order to determine the correct order of solution for coupled inductions. This sort may result in additional pruning should a recurrence requiring simultaneous solution occur[4]. This phase results in a directed acyclic graph (DAG) of the dependence relation between induction variables within one iteration of the loop[5]. Once this sort has been completed, initial values are determined, the closed forms are calculated, and substitution takes place.

At a high level, the calculation of the closed form of an induction variable involves a sweep through the loop which recurses upon encountering an inner loop. As the sweep proceeds through each loop, two expressions are formed to represent the closed form, one at the loop header and a second at the exit.

We will now turn to the details of the calculation of the closed forms. We will first treat additive inductions, then return to multiplicative inductions.

3.1.3 Additive Inductions

Definition 1

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

[4] e.g., i = i + j; j = j + i

[5] This approach ignores loop nesting, and thus could be optimized to allow a coupled recurrence across nests

Given a loop L with index $iter$, 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:

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

Here s iterates over the statements in loop L from $loop\,entry$ to $loop\,exit$[6].

Definition 2

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

$$i@i_{iv}^{L} \quad = \sum_{iter=loop\ init}^{iteration\,i-1\ by\ step} \psi_{iv}^{L}(iter)$$

Here $\psi_{iv}^{L}(iter)$ is summed over the iteration space of L from the lower bound to the $i-1$st iteration.

Definition 3

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

$$i@n_{iv}^{L} \quad = \sum_{iter=loop\ init}^{loop\ limit\ n\ by\ step} \psi_{iv}^{L}(iter)$$

Here we are summing $\psi_{iv}^{L}(iter)$ over the entire iteration space of L from the lower bound $loop\,init$ to the upper bound n by step.

[6]Note that ψ is defined $\psi \equiv 0$ for loops which have no (possibly nested) induction sites for iv

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[7].

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 inside these outer loops only as Phase 2 of the algorithm recursively returns 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 should be one of a statement-wise traversal of a given loop body, first summing increments, then multiplying across iteration spaces, all done recursively as necessary.

Let's trace the algorithm through an example. For the sake of simplicity, only examples with a stepsize of one will be used. However, any integer stepsize is allowable.

```
$M = 0$
Do $J = 1, U$
    Do $K = 1, J$
        $M = M + 1$
        $A(M) = 0$
    EndDo
EndDo
```

In this case, we have a triangular loop with a nesting depth of two. Each phase of the algorithm will now be considered in turn.

[7]The implementation of this sum function is discussed in Section 3.1.8

Phase 1 – Recognition

During the first phase, the anti-output-flow dependence on M is recognized. This is done very simply by pattern matching on assignments to scalar variables. Next, the induction is tested to determine if the right hand side increment varies by subtracting the left hand side from the right hand side. In the above case, M is classified as an additive induction with an invariant right hand side (the constant 1). Since no coupled inductions exist and flow control is limited to DO loops, M becomes a valid induction. The final step is to calculate ψ:

$$\psi_m^K(k) = \sum_{s=DOK}^{EndDo} increments\, to\, M = 1$$

This completes the recognition phase.

Phase 2 – Calculating the closed form

In this phase, processing begins with entry into the initial DO loop with index J[8]. Temporary values for $i@i$ and $i@n$ are formed within the context of this loop. Control immediately recurses on encountering the DO K loop. Since the limit of nesting has been reached, we now form the actual values of $i@i$ and $i@n$ in the context of the K loop:

$$i@i_m^K = \sum_{k'=1}^{k-1} 1 = k - 1$$

$$i@n_m^K = \sum_{k'=1}^{j} 1 = j$$

These expressions are now returned to the caller (DO J) where ψ, $i@i$, and $i@n$ are formed:

[8]In this thesis we assume, for the sake of simplicity, that all additive induction variables have an initial value of 0

$$\psi_m^J(j) = \sum_{s=DOJ}^{EndDo} increments\,to\,M = j$$

Note that ψ is the *index* of the current loop.

$$i@i_m^J = \sum_{j'=1}^{j-1} j' = j * (j-1)/2$$

$$i@n_m^J = \sum_{j'=1}^{u} j' = u * (u+1)/2$$

Phase 3 – Substitution

The substitution phase proceeds as follows:

```
For each Statement $S$ in Loop $L$
    Switch ($S.type$)
        case Do$_{L_{enc}}$: For each InductionVariable $iv$ in “$ivs\ in\ L$”
            If $iv$ is an $additive\ iv$
                Increment $RS_{iv}$ by $i@i_{iv}^{L_{enc}}$
            Else
                Multiply $RS_{iv}$ by $m@i_{iv}^{L_{enc}}$
        case EndDo$_{L_{enc}}$: For $iv$ in “$ivs\ in\ L$”
            If $iv$ is an $additive\ iv$
                Decrement $RS_{iv}$ by $\psi_{iv}^{L_{enc}}(iter)$ and by $i@i_{iv}^{L_{enc}}$
                Increment $RS_{iv}$ by $i@n_{iv}^{L_{enc}}$
            Else
                Divide $RS_{iv}$ by $m@i_{iv}^{L_{enc}}$ and by $\xi_{iv}^{L_{enc}}(iter)$
                Multiply $RS_{iv}$ by $m@n_{iv}^{L_{enc}}$
        case Assignment: For $iv$ in “$ivs\ in\ L$”
            If $S$ is an InductionVariableStmt
                If $iv$ is an $additive\ iv$
                    Increment $RS_{iv}$ by the current right hand side increment
                Else
                    Multiply $RS_{iv}$ by the current right hand side multiplier
            Else
                For each scalar Variable $V$ used in $S$
                    For $iv$ in “$ivs\ in\ L$”
                        If $V = iv$
                            For each use of $iv$ in $S$
                                Replace $iv$ with the Closed-form of $iv$
                    EndFor
                    Break
                  EndIf
                EndFor
              EndFor
            EndIf
        EndSwitch
      EndFor
```

In our example (repeated below for convenience), substitution proceeds by first iterating through the body of the J loop. A running sum RS is kept for each iv[9], and is updated according to the statement type encountered. The initial value of RS at the header of loop J is $j*(j-1)/2$. When the K loop header is reached, RS is incremented by $i@i_m^K$, resulting in $RS = (k-1)+(j*(j-1)/2)$. Next, the induction site $M = M+1$ is reached, and RS is updated with the right hand side increment +1. Thus, at the induction site, RS takes the value $k+(j*(j-1))/2$. Finally the use of M is encountered in the assignment statement to $A(M)$, and the current value of RS $(k+(j*(j-1))/2)$ is substituted for M.

```
$M = 0$
Do $J = 1, U$
    Do $K = 1, J$
        $M = M + 1$
        $A(M) = 0$
    EndDo
EndDo
```

The resulting code looks like this:

```
$M = 0$
DOALL $J = 1, U$
    DOALL $K = 1, J$
        $M_{private} = K + (J*(J-1))/2$
        $A(K + (J*(J-1))/2) = 0$
    EndDo
EndDo
```

[9]stored in the variable $*\_current\_value$ – see Section 3.1.7

The assignment to M now becomes unnecessary and the statement can be removed.

We are not quite finished because substitution now proceeds through the rest of the J loop. First the K EndDo is processed, and RS is updated with $i@n_m^K$. There are no other uses of M in the J loop, so upon encountering the J EndDo, RS is updated with $i@n_m^J$. This completes the substitution phase.

Let's turn our attention to multiplicative induction variables now.

3.1.4 Multiplicative Inductions

Similar definitions as those for additive induction variables hold for multiplicative induction variables:

Definition 1

Given a loop L with index $iter$, the product $\xi_{iv}^L\,(iter)$ of the multiplicative induction variable iv in the body of L is defined as:

$$\xi_{iv}^L\,(iter) = \prod_{s=loop\;entry}^{loop\;exit} multipliers\;of\;iv$$

Here s iterates over the statements in loop L from $loop\,entry$ to $loop\,exit$[10]:

Definition 2

The *multiplier at iteration* i of the multiplicative induction variable iv at the header of loop L is defined as:

$$m@i_{iv}^L \quad = \prod_{iter=loop\,init}^{iteration\,i-1\;by\;step} \xi_{iv}^L\,(iter)$$

Here the product of $\psi_{iv}^L\,(iter)$ is taken over the iteration space of L from the lower bound to the $i-1$st iteration.

[10]Note that ξ is defined $\xi \equiv 1$ for loops which have no (possibly nested) induction sites for iv

Definition 3

The *multiplier at n* of the multiplicative induction variable iv upon completion of loop L is defined as:

$$m@n_{iv}^{L} \quad = \prod_{iter=loop\,init}^{loop\,limit\,n\,by\,step} \xi_{iv}^{L}\,(iter)$$

Here we are taking the product of $\psi_{iv}^{L}\,(iter)$ over the entire iteration space of L from the lower bound $loop\,init$ to the upper bound n by step.

Consider the following example:

```
$K = 1$
Do $J = 1, M$
    $K = K * 2$
    $A(K) = 0$
EndDo
```

The following phases of the induction algorithm are executed:

Phase 1 – Recognition

During the first phase, the anti-output-flow dependence on K is recognized. This is done very simply by pattern matching on assignments to scalar variables[11]. Next, the induction is tested to determine if the right hand side increment varies by subtracting the left hand side from the right hand side. In the above case, this fails to identify K as an induction variable, so a further test is made to determine the nature of the operator. Once the operator has been recognized as multiplication, the multiplier is tested for invariance. In this case, 2 is invariant so K remains an active candidate. Since no coupled inductions exist and flow control is limited to DO loops, K becomes a valid induction. The final step is to calculate ξ:

[11] Using Polaris FORBOL primitives [26]

$$\xi_k^J(j) = \prod_{s=DOJ}^{EndDo} multipliers\ of\ K = 2$$

This completes the recognition phase.

Phase 2 – Calculating the closed form

In this phase, processing begins with entry into the initial DO loop with index J. Temporary values for $m@i$ and $m@n$ are formed within the context of this loop. Since the limit of nesting has been reached, we now form the actual values of $m@i$ and $m@n$ in the context of the J loop:

$$m@i_k^J = \prod_{j'=1}^{j-1} 2 = 2^{j-1}$$
$$m@n_k^J = \prod_{j'=1}^{m} 2 = 2^m$$

This completes phase 2 of the algorithm.

Phase 3 – Substitution

The substitution phase uses the same algorithm as that used for additive inductions, and proceeds by first iterating through the body of the J loop. The running sum RS has an initial value at the header of the J loop of 2^{j-1}. As control reaches the induction site, RS is multiplied by the right hand side multiplier 2. This results in the expression 2^j. When the use of K is encountered, K is substituted for the current value of $RS = 2^j$. Finally, control reaches the J EndDo, and RS is divided by ξ_k^J and $m@i_k^J$[12], and multiplied by $m@n_k^J$. This completes phase 3, yielding the final value $RS = 2^m$.

The resulting code looks like this:

[12]This results in $RS \equiv 1$

```
$K = 1$
DOALL $J = 1, M$
    $K_{private} = 2 * 2 * *(J - 1)$
    $A(2 * 2 * *(J - 1)) = 0$
EndDo
```

Before going on with more detailed examples of the induction algorithm, the implementation of the *zero-trip test* will be discussed.

3.1.5 Zero Trip Test

Zero trip loops pose a problem for induction substitution due to the potential for generating incorrect closed forms in the presence of loops with unknown symbolic bounds.

In a prototype implementation, the induction pass used the symbolic algebra package Maple^{TM} to solve recurrences. However, as has been pointed out in [16], there are serious drawbacks to both Mathematica^{TM} and Maple. Both of these packages are unable to handle the problem of zero-trip loops. For example, consider the following:

```
$K = 0$
READ *,$M$
Do $I = 1, N$
    Do $J = 1, M$
        $K = K + 1$
        $A(K) = 0$
    EndDo
EndDo
```

Without the accompanying context of the READ statement on M, both Maple and Mathematica will produce a closed form for the induction variable K which is incorrect if M is negative. It is therefore necessary to combine the functionality of a powerful symbolic algebra package with the semantics of the program. The induction substitution pass does this. As part of the pass, a powerful symbolic sum function works in conjunction with the recently developed *Symbolic Range Propagation* algorithm ([6]). The range propagation algorithm does a detailed analysis of the program semantics and determines relationships between variables at any point in the program. In the above example, the input dependence of M would be discovered by the algorithm and, in the absence of additional information such as user assertions, the resulting range of M would be determined to be unknown. This information would then be communicated to the induction pass and used by the zero-trip test to determine that the value of M is unknown, with the result that the recurrence on K would be solved only in the J loop.

Consider the following definition:

Definition

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

Normally, the intuitive understanding of loops leads one to the conclusion that $UB \geq LB \Leftrightarrow i@n$. In other words, if the loop upper bound is greater than or equal to the lower bound, $i@n$ is correct. However, this is not the case. To understand intuitively why this is so, take a look at the following example:

```
Parameter($UB = 10$)
$M = 0$
If ($P$) Then
    $U = 0$
Else
    $U = UB$
Do $J = 1, UB$
    Do $K = 1, U$
        $M = M + 1$
        $A(M) = 0$
    EndDo
    $Count = M$
EndDo
```

In this case, ψ, $i@i$, and $i@n$ for the K loop are:

$$\psi_m^K(k) = \sum_{s=DOK}^{EndDo} increments\, to\, M = 1$$

$$i@i_m^K = \sum_{k'=1}^{k-1} 1 = k - 1$$

$$i@n_m^K = \sum_{k'=1}^{u} 1 = u$$

And in the J loop:

$$\psi_m^J(j) = \sum_{s=DOJ}^{EndDo} increments\, to\, M = u$$

$$i@i_m^J = \sum_{j'=1}^{j-1} u = (j-1) * u$$

$$i@n_m^J = \sum_{j'=1}^{ub} u = u * ub$$

After induction solution the code becomes:

```
Parameter($UB = 10$)
$M = 0$
If ($P$) Then
    $U = 0$
Else
    $U = UB$
DOALL $J = 1, UB$
    DOALL $K = 1, U$
        $M_{private} = K + (J-1) * U$
        $A(K + (J-1) * U) = 0$
    EndDo
    $Count_{private} = J * U$
EndDo
```

The point to note is this: *the value of* $Count$ *will be correct no matter which control path is taken to define* U. In both cases, the value of $Count$ is identical to the value in the original code: if U is 0, $Count = J * U \equiv 0$, and if U is UB, $Count = J * U \equiv J * UB$. As a result the induction pass can relax the constraints when testing for zero trip loops. As we will see, this will come in handy later on.

3.1.6 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 ([21]). There is at least one important case in the Perfect Club Benchmark codes where a wrap around induction variable of this type occurs as the bound of a loop containing an induction site. This wrap around induction 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.

The recognition of both of these wrap around variables can be accomplished by representing the ProgramUnit in SSA form and using *back substitution* ([24]). For example, consider the SSA form of the above code fragment[13]:

```
LMIN_$1 = MJ_$0
LMAX_$1 = MI_$0
DO MK_$1 = MI, MORB, 1
   MIJKL_$1 = PHI(MIJKL_$0, MIJKL_$2)
   ML_$1 = PHI(ML_$0, ML_$2)
   LMIN_$2 = PHI(LMIN_$1, LMIN_$3)
   LMAX_$2 = PHI(LMAX_$1, LMAX_$3)
   DO ML_$2 = LMIN_$2, LMAX_$2, 1
      MIJKL_$2 = PHI(MIJKL_$1, MIJKL_$3)
      MIJKL_$3 = MIJKL_$2+1
```

[13]For reasons of brevity I assume some familiarity with the SSA form. For additional information, see [8]

```
280 LABEL 280
       ENDDO
       LMIN_$3 = 1
       LMAX_$3 = MK_$1+1
290 LABEL 290
    ENDDO
```

Let's trace the value of $LMAX$ through to it's use as the upper bound of the ML loop 280[14]. The first assignment of interest to $LMAX$ is $LMAX\_\$1 = MI\_\$0$. Next, $LMAX\_\$2 = \Phi(LMAX\_\$1,$ $LMAX\_\$3)$, so the initial value of $LMAX\_\$2$ (the ML loop upper bound), is $MI\_\$0$. $LMAX\_\$2$ takes it's next value from $LMAX\_\$3 = MK\_\$1 + 1$, which is the loop index of the MK loop plus 1. Within the MK loop, we can use *back substitution* to replace the term $LMAX\_\$3$ with $MK\_\$1 + 1$ in $\Phi(LMAX\_\$1, LMAX\_\$3)$. This results in a new Φ function $\Phi(LMAX\_\$3, MK\_\$1 + 1)$. Similarly, by forward substitution (for $LMAX\_\$1$), we get $\Phi(MI\_\$0, MK\_\$1 + 1)$. Since each back substitution corresponds to the execution of a single iteration of the loop, we conclude that after one iteration the value of $LMAX$ takes on the value of the MK loop index (plus a constant term), and thus loop ML is triangular with loop MK. In a similar way, it can be determined that $LMIN$ takes on the constant value 1 after one iteration. Thus, the entire loop nest can been reduced to a triangular entity simply by recognizing these relationships and peeling the first iteration of the MK loop[15].

We will now turn our attention to the C++ data structures employed in the induction variable substitution pass.

3.1.7 Data Structure Highlights

The use of object oriented features in Polaris in general was discussed briefly in the introduction to this section. In general the induction pass takes good advantage of the facility offered by such languages. Specifically, the data structure of the *InductionVariable* object logically conjoins both the control flow and data representation issues in a coherent manner, making both the understanding and enhancement of

[14] The SSA form presented here is taken from one of the Polaris internal representations
[15] An algorithm to accomplish this wrap around loop bound recognition is under development

the algorithm straightforward[16]. The InductionVariable object consists of two classes: *InductionVariable* and *increment_info*. The *increment_info* class tracks closed forms across multiple loops when inductions occur within multiply nested loops. The loop specific expressions ψ, $i@i$, $i@n$, ξ, $m@i$ and $m@n$ are stored here. The best way to think about this is to realize that an induction on a given variable can occur at multiple sites across multiply nested loops, and information specific to each loop must be maintained to properly solve the recurrence in all these different environments. Thus, multiple *increment_info* objects can exist for a single induction variable. In fact, these objects are stored in the list *_iv_info* in the *InductionVariable* object (see the declaration of *_iv_info* in the *InductionVariable* class below). Other key data members of *InductionVariable* include *_initial_value* and *_current_value*, both of which are used during the substitution phase. The private data members for each of these classes are displayed below:

```
class increment_info : public Listable {
    friend class InductionVariable;
    private:
        Statement *_enclosing_do_loop;
        Expression *_sum_of_incs;
        Expression *_prod_of_mults;
        Expression *_closed_form;
        Expression *_last_value;
        int _visited;
        RefList<increment_info> _enclosed_loops;
};
class InductionVariable : public Listable {
    private:
        Symbol *_symbol;
        InductionVarType _type;
        Expression *_initial_value;
        Expression *_current_value;
        Expression *_sum_of_incs;
        Expression *_prod_of_mults;
        Expression *_vvar_current_value;
        Expression *_outermost_use;
        List<increment_info> _iv_info;
        RefSet<Symbol> _vvarsyms;
        int _variant;
        int _solved;
};
```

[16]It took only about five hours to add the functionality to solve multiplicative induction variables on top of the existing pass

Although not part of the induction pass, the *Listable* class is worth mentioning due to its frequent (and invaluable) use. An essential part of the Polaris infrastructure, the *Listable* class provides, via inheritance, the basic framework upon which both the *InductionVariable* and *increment_info* classes are built.

3.1.8 Symbolic Sum Function

A symbolic sum function was implemented as part of the Polaris infrastructure. Based on *Bernoulli Numbers* (after Jakob Bernoulli), the sum function computes the sums of mth powers[17]:

Let $S_m(n)$ be defined as:

$$S_m(n) = 0^m + 1^m + \cdots + (n-1)^m = \sum_{k=0}^{n-1} k^m$$

The discovery of Bernoulli can be summarized as:

$$S_m(n) = \frac{1}{m+1} \sum_{k=0}^{m} \left(\begin{array}{c} m+1 \\ k \end{array} \right) B_k n^{m+1-k} \tag{3.1}$$

Where $\left(\begin{array}{c} m+1 \\ k \end{array} \right)$ is the combinatorial function *m+1 choose* k[18] $= \frac{(m-1)!}{k!(m-k-1)!}$.

To complete the above summation it is necessary to calculate the coefficients B_k. This is algorithmically possible using the following definition([14], page 270):

$$\sum_{j=0}^{m} \left(\begin{array}{c} m+1 \\ j \end{array} \right) B_j = \begin{cases} 1 & \text{if } m = 0 \\ 0 & \text{for } m > 0 \end{cases}$$

The Polaris *Summation* class implements an algorithmic solution to this implicit recurrence. Given the coefficients B_k, expressions are factored into monomial form and summed according to 3.1 above.

[17]Material drawn from [14], page 269

[18]Normally denoted $C(n, r)$

3.1.9 Induction Variable Substitution in TRFD

TRFD, one of the Perfect Club Benchmark codes, is a kernel simulating the computational aspects of a two-electron integral transformation. It's interest to us lies in the presence of some unusually complex induction variables.

The kernel of code we will be studying occurs in the subroutine OLDA, and has the following form:

```
NRS = (NUM * (NUM + 1))/2
NIJ = (MORB * (MORB + 1))/2
MIJ = 0
MIJKL = 0
MLEFT = NRS - NIJ
DO MI = 1, MORB, 1
   DO MJ = 1, MI, 1
      MIJ = MIJ+1
      LMIN = MJ
      LMAX = MI
      DO MK = MI, MORB, 1
         DO ML = LMIN, LMAX, 1
            MIJKL = MIJKL+1
         ENDDO
         LMIN = 1
         LMAX = MK+1
      ENDDO
      MIJKL = MIJKL+MIJ+MLEFT
   ENDDO
ENDDO
```

This code fragment has been extracted from OLDA_do300, a loop nest which accounts for 28% of the serial execution time of the benchmark[19]. As can be seen, the *triangular* induction variable MIJ is *coupled* to the *triangular* induction variable $MIJKL$[20].

There is an added complication in this loop which must be solved prior to the solution of the coupled inductions on MIJ and $MIJKL$. Namely, you will notice the bounds of the inner loop ML are loop variant wrap-around variables. The induction algorithm handles this as outlined above in Section 3.1.6.

After manually completing the transformation described above, the code becomes:

[19] measured on the SGI Challenge
[20] In fact, the induction site inside loop MK is doubly-triangular in nature

```
NRS = (NUM * (NUM + 1))/2
NIJ = (MORB * (MORB + 1))/2
MIJ = 0
MIJKL = 0
MLEFT = NRS - NIJ
DO MI = 1, MORB, 1
   DO MJ = 1, MI, 1
      MIJ = MIJ+1
      DO ML = MJ, MI, 1
         MIJKL = MIJKL+1
      ENDDO
      DO MK = MI+1, MORB, 1
         DO ML = 1, MK, 1
            MIJKL = MIJKL+1
         ENDDO
      ENDDO
      MIJKL = MIJKL+MIJ+MLEFT
   ENDDO
ENDDO
```

After performing this transformation, it turns out that the MK loop fails to satisfy the relation $MORB \geq MI+1$ in the last iteration of the MI loop. Fortunately this does not pose a problem, due to the fact that the MK loop executes an *exact zero-trip* on the last iteration (discussed in section 3.1.5). As we will see below, the closed form $i@n^{MK}_{mikjl}$ is:

$$i@n^{MK}_{mijkl} = \sum_{mk'=mi+1}^{morb} mk' = (morb + morb^2 - mi - mi^2)/2$$

When $MI = MORB$ (in the last iteration of the outermost MI loop), $i@n^{MK}_{mijkl} \equiv 0$, and as a result, the contribution of $i@n^{MK}_{mijkl}$ to $i@n^{MI}_{mijkl}$ is also 0. This preserves the semantics of the original untransformed MK loop.

We are now ready to proceed with an exact trace of the TRFD example through the three phases of the induction solution algorithm.

Phase 1

There are two induction variables in this example, MIJ and $MIJKL$. During the recognition phase, MIJ becomes a valid candidate with a constant right hand side increment. Likewise, $MIJKL$ is incremented by a loop invariant ($MLEFT$) and a candidate iv, MIJ[21].

In determining the sum of increments ψ for these five loops, it becomes immediately clear that at this point in the algorithm, ψ can only be calculated for loops in which induction sites occur. Thus we have:

$$\begin{aligned}
\psi^{MJ}_{mij}(mj) &= \sum_{s=DOMJ}^{EndDo} increments\ to\ MIJ &= 1\\
\psi^{MJ}_{mijkl}(mj) &= \sum_{s=DOMJ}^{EndDo} increments\ to\ MIJKL &= mleft\\
\psi^{ML_{MK}}_{mijkl}(ml_{mk}) &= \sum_{s=DOML_{MK}}^{EndDo} increments\ to\ MIJKL &= 1\\
\psi^{ML_{MJ}}_{mijkl}(ml_{mj}) &= \sum_{s=DOML_{MJ}}^{EndDo} increments\ to\ MIJKL &= 1
\end{aligned}$$

Note that ψ^{MJ}_{mijkl} is only a partial sum at this point. The complete value for ψ^{MJ}_{mijkl}, as well as the remainder of the values for ψ in the MK, MJ, and MI loops, must be determined during the calculation of the closed forms.

Lastly, since no control flow structures other than DO loops occur inside loop MI, we may proceed to Phase 2 of the algorithm.

[21]Note that the order in which the coupled induction variables occur is unimportant. The dependence between them will be determined and resolved (if possible) during Phase 2

Phase 2

The sorting stage is fairly straightforward[22]. The dependence relation between MIJ and $MIJKL$ is determined, and MIJ is solved first. Although values for $i@i_{mij}$ and $i@n_{mij}$ exist in each loop, they are of importance only in the MI loop, since this is the environment within which we solve the induction on MIJ, as well as reference it. The values for $i@i_{mij}$ and $i@n_{mij}$ are derived as follows:

$$\begin{aligned}
i@i_{mij}^{MJ} &= \sum_{mj'=1}^{mj-1} 1 = mj-1 \\
i@n_{mij}^{MJ} &= \sum_{mj'=1}^{mi} 1 = mi \\
i@i_{mij}^{MI} &= \sum_{mi'=1}^{mi-1} mi = (mi*(mi-1))/2 \\
i@n_{mij}^{MI} &= \sum_{mi'=1}^{morb} mi = (morb*(morb+1))/2
\end{aligned}$$

Likewise, the values for $i@i_{mijkl}$ and $i@n_{mijkl}$ are as follows:

The ML_{MJ} loop:

$$\begin{aligned}
i@i_{mijkl}^{ML_{MJ}} &= \sum_{ml'=mj}^{ml-1} 1 = ml-mj \\
i@n_{mijkl}^{ML_{MJ}} &= \sum_{ml'=mj}^{mi} 1 = 1+mi-mj
\end{aligned}$$

The ML_{MK} loop:

$$i@i_{mijkl}^{ML_{MK}} = \sum_{ml'=1}^{ml-1} 1 = ml-1$$

[22] In fact, the topological sort is done concurrently with the calculation of the closed forms

$$i@n^{ML_{MK}}_{mijkl} \;=\; \sum_{ml'=1}^{mk} 1 \;=\; mk$$

The MK loop:

$$\begin{aligned} i@i^{MK}_{mijkl} &= \sum_{mk'=mi+1}^{mk-1} mk' \\ &= (mk^2 - mi - mk - mi^2)/2 \\ i@n^{MK}_{mijkl} &= \sum_{mk'=mi+1}^{morb} mk' \\ &= (morb + morb^2 - mi - mi^2)/2 \end{aligned}$$

The MJ loop:

$$\begin{aligned} & i@i^{MJ}_{mijkl} = \\ & \quad \sum_{mj'=1}^{mj-1} (mj' + mleft + (mi^2 - mi)/2 + i@n^{MK}_{mijkl} + i@n^{ML_{MJ}}_{mijkl}) \\ & \quad = mj - 1 + (-morb - morb^2 + mj * morb + mj * morb^2)/2 - mleft + mj * mleft \end{aligned}$$

$$\begin{aligned} & i@n^{MJ}_{mijkl} = \\ & \quad \sum_{mj'=1}^{mi} (mj' + mleft + (mi^2 - mi)/2 + i@n^{MK}_{mijkl} + i@n^{ML_{MJ}}_{mijkl}) \\ & \quad = mi + (mi * morb + mi * morb^2)/2 + mi * mleft \end{aligned}$$

The MI loop:

$$i@i^{MI}_{mijkl} = \sum_{mi'=1}^{mi-1} \big(mi' + (mi' * morb + mi' * morb^2)/2 + mi' * mleft\big)$$

$$= ((morb*mi^2+mi^2*morb^2-mi*morb-mi*morb^2)/2+mi^2-mi+mleft*mi^2-mi*mleft)/2$$

$$i@n^{MI}_{mijkl} = \sum_{mi'=1}^{morb} \big(mi' + (mi' * morb + mi' * morb^2)/2 + mi' * mleft\big)$$

$$= (morb + (morb^2 + 2 * morb^3 + morb^4)/2 + morb^2 + morb * mleft + mleft * morb^2)/2$$

Phase 3

The third and final phase begins with RS^{MI}_{mij} equal to $(mi*(mi-1))/2$. Upon entering the MJ loop, RS is updated with $mj-1$ and becomes $mj-1+(mi*(mi-1))/2$. Next, we encounter the induction site $MIJ = MIJ+1$, and RS is updated to $mj+(mi*(mi-1))/2$, the value which is later substituted for MIJ on the right hand side of the $MIJKL$ induction site at the bottom of the MJ loop. Although this substitution does not take place at this point in the algorithm, when it does take place the $MIJKL$ induction site will become $MIJKL = MIJKL + mj + (mi*(mi-1))/2 + mleft$[23].

Continuing with $MIJKL$ now, we start out with an initial value for RS of:

$$\begin{aligned} RS^{MI}_{mijkl} &= ((morb*mi^2 + mi^2*morb^2 - mi*morb \\ &\quad - mi*morb^2)/2 + mi^2 - mi + mleft*mi^2 \\ &\quad - mi*mleft)/2 \end{aligned}$$

Upon encountering the MJ loop, this value is incremented by $mj-1+(-morb-morb^2+mj*morb+mj*morb^2)/2 - mleft + mj*mleft$. Next, we enter the ML_{MJ} loop, and RS is incremented by $ml-mj$. Now we have reached the first induction site, $MIJKL = MIJKL+1$ inside the ML_{MJ} loop, and RS becomes:

$$\begin{aligned} RS &= ((morb*mi^2 + mi^2*morb^2 - mi*morb \\ &\quad - mi*morb^2)/2 + mi^2 - mi + mleft*mi^2 \\ &\quad - mi*mleft)/2 + ml - 1 + (-morb - morb^2 + mj*morb \end{aligned}$$

[23]Note that the current value of RS_{mij} is carried through the loops ML_{MJ}, MK, and ML_{MK} without change, to this induction site

$$+ mj * morb^2)/2 - mleft + mj * mleft$$

Next the ML_{MJ} EndDo is reached, and RS is decremented by $i@i^{ML_{MJ}}_{mijkl}$, then incremented by $i@n^{ML_{MJ}}_{mijkl}$, and finally decremented by $\psi^{ML_{MJ}}_{mijkl}$. The resulting value is:

$$\begin{aligned} RS &= mi + ((morb * mi^2 + mi^2 * morb^2 - mi * morb \\ &\quad - mi * morb^2)/2 + mi^2 - mi - morb - morb^2 \\ &\quad + mj * morb + mj * morb^2 + mleft * mi^2 - mi * mleft)/2 \\ &\quad - mleft + mj * mleft \end{aligned}$$

The next two statements encountered are the MK and ML_{MK} loops. RS is incremented by $(mk^2 - mi - mk - mi^2)/2$ $(i@i^{MK}_{mijkl})$ and $ml - 1$ $(i@i^{ML_{MK}}_{mijkl})$. The resulting sum is:

$$\begin{aligned} RS &= ml - 1 + ((morb * mi^2 + mi^2 * morb^2 - mi * morb \\ &\quad - mi * morb^2)/2 + MK^2 - MK - morb - morb^2 \\ &\quad + mj * morb + mj * morb^2 + mleft * mi^2 - mi * mleft)/2 \\ &\quad - mleft + mj * mleft \end{aligned}$$

Control flow now reaches the second induction site and RS is incremented by 1. As control leaves the ML_{MK} and MK loop nests, RS is decremented by $\psi^{ML_{MK}}_{mijkl}$ (which is 1), as well as by $i@i^{ML_{MK}}_{mijkl}$ and $i@i^{MK}_{mijkl}$. Then RS is incremented by $(morb + morb^2 - mi - mi^2)/2$ $(i@n^{MK}_{mijkl})$ and mk $(i@n^{ML_{MK}}_{mijkl})$, yielding the value:

$$\begin{aligned} RS &= ((morb * mi^2 + mi^2 * morb^2 - mi * morb \\ &\quad - mi * morb^2)/2 + mj * morb + mj * morb^2 \\ &\quad + mleft * mi^2 - mi * mleft)/2 - mleft + mj * mleft \end{aligned}$$

Finally we encounter the remaining induction site in the MJ loop. At this site, RS is incremented by ψ^{MJ}_{mijkl}, which at this point in the processing has become $mj + (mi * (mi - 1))/2 + mleft$[24]. The resulting value is:

$$\begin{aligned} RS &= mj + ((morb * mi^2 + mi^2 * morb^2 - mi * morb \\ &\quad - mi * morb^2)/2 + mi^2 - mi + mj * morb + mj * morb^2 \\ &\quad + mleft * mi^2 - mi * mleft)/2 + mj * mleft \end{aligned}$$

As control flows out of the MJ loop, RS is decremented by ψ^{MJ}_{mijkl}, $i@i^{MJ}_{mijkl}$, $i@n^{ML_{MJ}}_{mijkl}$, and $i@n^{MK}_{mijkl}$. It is then incremented by $i@n^{MJ}_{mijkl}$ with the result:

$$\begin{aligned} RS &= (mi + (morb * mi^2 + mi^2 * morb^2 - mi * morb \\ &\quad - mi * morb^2)/2 + mi^2 + mi * morb + mi * morb^2 \\ &\quad + mleft * mi^2 - mi * mleft)/2 + mi * mleft \end{aligned}$$

The final update takes place upon encountering the MI EndDo, where RS is decremented by $i@n^{MJ}_{mijkl}$ and incremented by $i@n^{MI}_{mijkl}$ with the result:

[24] This is just the closed form of MIJ plus mleft

$$\begin{aligned} RS &= (morb + (morb^2 + morb^4)/2 + morb^2 + morb^3 \\ &\quad + mleft * morb + mleft * morb^2)/2 \end{aligned}$$

3.1.10 Induction Variable Substitution in OCEAN

The following is an excerpt from the Perfect Club Benchmark OCEAN[25], subroutine FTRVMT. A multiplicative induction occurs on the complex variable EXJ inside loop 109. However, due to the complexity of control flow within this loop, the iteration space must be partitioned prior to induction solution. An algorithm to accomplish this partitioning is under development. The unpartitioned code follows:

```
      SUBROUTINE FTRVMT (DATA,EX)
      ...
      DO 109 JL=1,I2K
         IF(JL-1) 102,102,104
  102    EXJ=(1.,0.)
         DO 103 JJ=JL,NPTS,I2KP
            DO 103 MM=1,MTRN
               JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
               H=DATA(JS)-DATA(JS+I2KS)
               DATA(JS)=DATA(JS)+DATA(JS+I2KS)
               DATA(JS+I2KS)=H
  103    CONTINUE
         GO TO 109
  104    IF(JL-JLI) 105,107,105
C
C INCREMENT JL-DEPENDENT EXPONENTIAL FACTOR
C
  105    EXJ=EXJ*EXK
         DO 106 JJ=JL,NPTS,I2KP
            DO 106 MM=1,MTRN
               JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
               H=DATA(JS)-DATA(JS+I2KS)
               DATA(JS)=DATA(JS)+DATA(JS+I2KS)
               DATA(JS+I2KS)=H*EXJ
```

[25] See [20]

```
  106    CONTINUE
         GO TO 109
  107    EXJ=CMPLX(0.,SGN1)
         DO 108 JJ=JL,NPTS,I2KP
            DO 108 MM=1,MTRN
               JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
               H=DATA(JS)-DATA(JS+I2KS)
               DATA(JS)=DATA(JS)+DATA(JS+I2KS)
               DATA(JS+I2KS)=CMPLX(-SGN1*HH(2),SGN1*HH(1))
  108    CONTINUE
  109 CONTINUE
      ...
      RETURN
      END
```

After manual partitioning, the code becomes:

```
      SUBROUTINE FTRVMT(DATA,EX)
      ...
      EXJ = (1.,0.)
      DO 103 JJ = 1, NPTS, I2KP
         DO 103 MM = 1, MTRN, 1
            JS = (JJ-1)*NSKIP+(MM-1)*MSKIP+1
            H = DATA(JS)-DATA(JS+I2KS)
            DATA(JS) = DATA(JS)+DATA(JS+I2KS)
            DATA(JS+I2KS) = H
  103 CONTINUE
      DO 109A JL = 2, JLI-1, 1
         EXJ = EXJ*EXK
         DO 106A JJ = JL, NPTS, I2KP
            DO 106A MM = 1, MTRN, 1
               JS = (JJ-1)*NSKIP+(MM-1)*MSKIP+1
               H = DATA(JS)-DATA(JS+I2KS)
               DATA(JS) = DATA(JS)+DATA(JS+I2KS)
               DATA(JS+I2KS) = H*EXJ
 106A    CONTINUE
 109A CONTINUE
      IF (JLI.NE.1) then
         EXJ = CMPLX(0.,SGN1)
         DO 108 JJ = JLI, NPTS, I2KP
            DO 108 MM = 1, MTRN, 1
               JS = (JJ-1)*NSKIP+(MM-1)*MSKIP+1
               H = DATA(JS)-DATA(JS+I2KS)
               DATA(JS) = DATA(JS)+DATA(JS+I2KS)
               DATA(JS+I2KS) = CMPLX(-SGN1*HH(2),SGN1*HH(1))
  108    CONTINUE
      ENDIF
      DO 109B JL = JLI+1, I2K, 1
         EXJ = EXJ*EXK
         DO 106B JJ = JL, NPTS, I2KP
```

```
            DO 106B MM = 1, MTRN, 1
               JS = (JJ-1)*NSKIP+(MM-1)*MSKIP+1
               H = DATA(JS)-DATA(JS+I2KS)
               DATA(JS) = DATA(JS)+DATA(JS+I2KS)
               DATA(JS+I2KS) = H*EXJ
 106B    CONTINUE
 109B CONTINUE
      ...
      RETURN
      END
```

Essentially what we've done is to peel the first iteration, execute the next $JLI-2$ iterations (in loop 109A), peel the JLIth iteration, and then execute the final $I2K-JLI$ iterations (loop 109B).

Let's trace the induction substitution algorithm through the three phases of solution for this loop.

Phase 1 – Recognition

Recognition of the induction variable EXJ proceeds very much like that for K is the previous example. EXJ is recognized as a multiplicative induction variable within the context of the two JL loops (hereafter referred to as JL_1 and JL_2). Again similar to the previous example, EXK is invariant within both JL loops, and since no coupled inductions exist EXJ becomes a valid induction. The final step is to calculate ξ[26]:

$$\xi_{exj}^{JL_1}(jl_1) = \prod_{s=DOJL_1}^{EndDo} multipliers\ of\ EXJ = exk$$

$$\xi_{exj}^{JJ_{JL_1}}(jj_{jl_1}) = \prod_{s=DOJJ}^{EndDo} multipliers\ of\ EXJ \equiv 1$$

$$\xi_{exj}^{MM_{JL_1}}(mm_{jl_1}) = \prod_{s=DOMM}^{EndDo} multipliers\ of\ EXJ \equiv 1$$

$$\xi_{exj}^{JL_2}(jl_2) = \prod_{s=DOJL_2}^{EndDo} multipliers\ of\ EXJ = exk$$

[26]Note that $\xi_{exj}^{JJ_{JL}}$ and $\xi_{exj}^{MM_{JL}}$ are by definition 1 for both loops because no induction site exists inside either of these loops (even though EXJ *is* used)

$$\xi_{exj}^{JJ_{JL_2}}(jj_{jl_2}) = \prod_{s=DOJJ}^{EndDo} multipliers\ of\ EXJ \equiv 1$$

$$\xi_{exj}^{MM_{JL_2}}(mm_{jl_2}) = \prod_{s=DOMM}^{EndDo} multipliers\ of\ EXJ \equiv 1$$

This completes the recognition phase.

Phase 2 – Calculating the closed form

In this phase, processing begins with entry into the initial DO loop with index JL_1. However, the issue of initial values must be dealt with explicitly. EXJ is a complex variable, and like additive induction variables, it's initial value must be a appear in the expression for the closed form[27]. In this example, EXJ has an initial value of CMPLX$(1.0, 0.0)$ outside the JL_1 loop. If the initial value for any induction variable can be determined as a constant (or loop invariant), that value will be used in the closed form. If however, the initial value cannot be determined statically, a temporary is created outside the parallel loop and assigned the value of the induction variable at that point. In this example, a temporary $EXJ0$ is created and the assignment $EXJ0 = EXJ$ placed just outside the JL_1 loop header[28].

Continuing with Phase 2 now, temporary values for $m@i$ and $m@n$ are formed within the context of the JL_1 loop. Since the limit of nesting has been reached with respect to induction sites for EXJ[29], we now form the actual values of $m@i$ and $m@n$ in the context of this first loop. Processing then continues with the JL_2 loop, resulting in the following four equations:

$$m@i_{exj}^{JL_1} = \prod_{jl'_1=2}^{jl_1-1} exk = exk^{jl-2}$$

[27]We sidestepped this issue when dealing with additive inductions simply by assuming initial values of 0

[28]The induction solution pass is relatively simple when it comes to determining initial values – if control flow diverges while searching for an initial value, the search ends and a temporary variable is created and assigned at the header of the parallel loop

[29]For the sake of consistency, $m@i_{exj}$ and $m@n_{exj} in \left\{JJ_{JL_1}, MM_{JL_1}, JJ_{JL_2}, MM_{JL_2}\right\}$ are all defined per their proper definitions with $\xi_{exj} \equiv 1$

$$m@n_{exj}^{JL_1} = \prod_{jl'_1=2}^{jli-1} exk = exk^{jli-2}$$

$$m@i_{exj}^{JL_2} = \prod_{jl'_2=jli+1}^{jl_2-1} exk = exk^{jl-(jli+1)}$$

$$m@n_{exj}^{JL_2} = \prod_{jl'_2=jli+1}^{i2k} exk = exk^{i2k-jli}$$

This completes phase 2 of the algorithm.

Phase 3 – Substitution

The substitution phase begins in loop JL_1 with $RS = m@i_{exj}^{JL_1} = exk^{jl-2}$. As control reaches the induction statement $EXJ = EXJ * EXK$, RS is multiplied by both EXK and the initial value $EXJ0$ with the result that $RS = exj0 * exk^{jl-1}$. Control flow now enters the JJ^{JL_1} loop, and the value of RS remains unchanged ($m@i_{exj}^{JJ_{JL_1}} \equiv 1$). Likewise, upon entry to the MM^{JL_1} loop, RS remains unchanged.

We next encounter the use of EXJ in the statement $DATA(JS + I2KS) = H * EXJ$, and EXJ is substituted by RS yielding $DATA(JS + I2KS) = H * EXJ0 * EXK^{JL-1}$. As control leaves the JJ and MM inner loops, RS is divided by $m@i_{exj}^{MM_{JL_1}}$, $m@i_{exj}^{JJ_{JL_1}}$, ξ_{exj} (for both loops), and multiplied by $m@n_{exj}^{MM_{JL_1}}$ and $m@n_{exj}^{JJ_{JL_1}}$, yielding $RS = ((((((RS/1)/1)/1)/1) * 1) * 1) = RS$[30].

Control has now reached the JL_1 EndDo, and RS is divided by $m@i_{exj}^{JL_1} = exk^{jl-2} * \xi_{exj}^{JL_1} = exk$ (with the result $RS = 1$), and multiplied by $m@n_{exj}^{JL_1} = exk^{jli-2}$. The *last value* of EXJ in the JL_1 loop is thus $exj0 * exk^{jli-2}$.

Processing for the JL_2 loop is almost identical, the only difference being in the bounds of the loop.

This completes Phase 3 of the algorithm, and we will now turn our attention to Reduction Recognition.

[30] In fact the implementation doesn't actually do the division or multiplication in this case. Instead, upon entry to and exit from a loop, the values for $i@i$, $m@i$, $i@n$, $m@n$, ψ, and ξ are first checked, and only if they exist (are non-NULL), is the operation completed

3.2 Reduction Recognition

As the title of this section suggests, solving reductions in parallel is handled in more than one step. It's should be mentioned that Polaris is, in general, a tool for recognizing parallelism. Parallel reductions are solved within this framework, and as a result the work described in this section blends naturally with Polaris' overall goal. Nonetheless, the implementation of parallel reductions is of critical importance, and will be discussed below in Section 3.2.4.

3.2.1 Algorithm Overview

The first step in the solution of parallel reductions is recognition of potential reductions. Following this, a second, data-dependence test pass analyzes these candidate reductions to determine if they are indeed reductions [5]. In the case where the data-dependence test fails to determine independence, the algorithm conservatively assumes that a reduction actually exists so that, in effect, we are still able to parallelize the loop.

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

$$A(\alpha(i,j,k,...)) = A(\alpha(i,j,k,...)) + \beta(l,m,n,...)$$

where A may be a multi-dimensional array, α and β may be multivariate, neither α nor β 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).

We found a pattern-matching approach to be of sufficient power to recognize commonly occurring, performance critical reductions in our test program suite. The reduction recognition pass of Polaris is thus based on powerful pattern-matching primitives that are part of the Polaris FORBOL environment[31]. As a rule we assume that the + operation is associative, and we have not run into any numerical problems to date.

[31] Discussed further in Data Structure Highlights Section 3.2.3

3.2.2 Outline of Recognition Algorithm

At a high level the algorithm is as follows:

```
For each ProgramUnit in the Program
    Normalize loops
    For each do loop in the ProgramUnit
        For each statement in the loop body
            Match (potentially multiple) definition-use pairs of variables of the form:
                $A(\alpha(i, j, k, ...)) = A(\alpha(i, j, k, ...)) + \beta(l, m, n, ...)$
                (This includes possible definition-use pairs based on equivalences)
            Insure that neither A nor its equivalences are referenced in either $\alpha$ or $\beta$
        EndFor
        For each successful candidate reduction statement
            Eliminate any variable (or its equivalence) that is defined or used elsewhere
            in the loop in a statement which is not itself a candidate reduction statement
        EndFor
    EndFor
EndFor
```

As is clear from the algorithm above, reduction recognition iterates through each program unit in the program[32]. For each ProgramUnit, the algorithm normalizes loops to have a lower bound of 1. All references to the loop index within a given loop are adjusted accordingly[33].

Next we iterate through each do loop in a given ProgramUnit and each statement in the loop. The performance critical reductions identified in [11] are additive reductions. As a result, at this stage of the algorithm we look only at assignment statements of the form outlined in the algorithm above.

Having identified a list of all assignment statements of this form, the candidate list is then pruned to exclude statements for which a singular definition or use of the reduction variable occurs. For example, the array B in the following code would be removed from the candidate list:

```
K = constant
Do $I = 1, N$
    . . .
    $B(K, \ldots) = B(K, \ldots) + \beta(i, \ldots)$
    . . .
```

[32] In fact ProgramUnit is a Polaris C ++ class representing a FORTRAN semantic unit (e.g., a subroutine)

[33] This normalization proved necessary due to the interaction between the reduction recognition pass and the data-dependence test (ddtest) pass

```
        $\ldots = B(I, \ldots)$
        $\ldots$
    EndDo
```

Finally, the successful candidate list of reductions is attached to the loop header of each enclosing loop. This is done so that when a parallel loop is chosen[34] at code generation, the reduction information is available in the proper environment.

The reduction on A in the algorithm above depicts A as a single dimensioned array indexed by the possibly multi-variate function α. However, the algorithm supports reductions on arrays of multiple dimensions – the only constraint is that no more than one dimension may vary within a given loop. This was done, however, to ease implementation, and the algorithm could be extended to handle variation in multiple dimensions as well.

To recap, the algorithm above recognizes potential reductions which fall into the two classes of *histogram reductions* and *single address reductions*. Assertions are created for each reduction variable, and the ddtest pass then processes these statements to determine their dependence relation. If ddtest can prove that there are no loop-carried dependences involving the reduction variable, the assertion is deleted; otherwise, it is left intact.

3.2.3 Data Structure Highlights

The reduction recognition pass in Polaris primarily makes use of the class Wildcard[35]. For example, the pattern used to match definition-use pairs in the form:

$$A(\alpha(i,j,k,...)) = A(\alpha(i,j,k,...)) + \beta(l,m,n,...)$$

is implemented as

[34] Possibly from among several alternatives
[35] Part of the FORBOL environment mentioned above [26]

$$wildcard\_not(wildcard\_contains(id(symbol))).$$

This translates to the creation of a Wildcard object which does not contain a reference to $id(symbol)$, where $id(symbol)$ returns a reference to the array name A shown above[36]. All structure matching is then done using these objects.

For further detail on the implementation of the WildCard class structure, an excellent source can be found in [26], which treats the FORBOL environment in Polaris.

There is much of interest and importance to discuss in the reduction backend, and this is the topic of the next section.

3.2.4 Code Generation Overview

As noted in the introduction to this section, reduction recognition is only the first step. The implementation of parallel reductions is architecture specific, and this section deals with many of these issues.

In the reduction backend, three different transformations are possible. They are termed *blocked*, *privatized*, and *expanded*. Currently the user determines which transformation to use on a program-global basis, but eventually we hope to have the compiler determine the most efficient transformation on a loop-by-loop basis.

The first solution, termed *blocked*, involves the insertion of synchronization primitives around each reduction statement. This method is desirable simply because it always works[37]. However, the overhead is not insignificant.

In *privatized* reductions, a variable that is private to each processor is created. All reductions are summed into this new private variable, be it a scalar or an array. This method requires initialization in a preamble to the parallel loop, and a global reduction (across processors) in the corresponding postamble. The advantages are twofold: one, the loop may now be executed in a parallel doall fashion

[36]When seen in more detail, this statement expands to: new WildCardNot (new WildCardContains (id(symbol)))
[37]Modulo numerical considerations

without the need for synchronization (modulo the reduction in the postamble); and two, on logically shared, physically distributed memory (LSPDM) architectures, locality of reference is improved[38]. In our current implementation, the global reduction in the postamble is done using synchronization primitives.

The third category, termed *expanded*, uses a global array that has an additional dimension equal to the number of processors executing in parallel. All reduction variables (again, be they scalar or multi-dimensional in nature), are replaced by references to this new, global array. Index expressions in the original N dimensions remain unchanged; however, the final, newly created dimension is indexed by the processor *id* of the processor currently executing a given iteration. Similar to privatized reductions, no dependences remain, and the loop may be executed entirely in parallel. The advantages in expansion lie in the lack of a need for synchronization. Although similar to privatization in that initialization and global reductions are necessary prior to and following execution of the parallel loop, unlike privatization, both of these operations can be done completely in parallel [39].

Let's now consider an example.

3.2.5 Parallel Reductions in MDG

Consider the following excerpt from the Perfect Club Benchmark code MDG:

```
      SUBROUTINE INTERF(X, Y, Z, FX, FY, FZ, XM, YM, ZM, VIR)
      ...
      IW1 = 1
      IWO = 2
      IW2 = 3
CSRD$ LOOPLABEL 'INTERF_do1000'
CSRD$ ASSERT NATOMS.GE.0
      DO I = 1, NMOL1, 1
       JW1 = IW1
       JWO = IWO
       JW2 = IW2
CSRD$ LOOPLABEL 'INTERF_do1100'
       DO J = I+1, NMOL, 1
        JW1 = JW1+NATOMS
        JWO = JWO+NATOMS
```

[38]E.g., the Convex Exemplar

[39]This point will further discussed below

```
          JW2 = JW2+NATOMS
          IF ((-9)+KC.NE.0) THEN
           ...
           G110 = GG(10)+GG(1)*C1
           G23 = GG(2)+GG(3)
           G45 = GG(4)+GG(5)
           FX(IW0) = FX(IW0)+G110+GG(11)+GG(12)+C1*G23
           FX(JW0) = (((FX(JW0)-G110)-GG(13))-GG(14))-C1*G45
           TT1 = GG(1)*C2
           TT = G23*C2+TT1
           FX(IW1) = FX(IW1)+GG(6)+GG(7)+GG(13)+TT+GG(4)
           FX(IW2) = FX(IW2)+GG(8)+GG(9)+GG(14)+TT+GG(5)
           TT = G45*C2+TT1
           FX(JW1) = ((((FX(JW1)-GG(6))-GG(8))-GG(11))-TT)-GG(2)
           FX(JW2) = ((((FX(JW2)-GG(7))-GG(9))-GG(12))-TT)-GG(3)
           ...
          ENDIF
   1100   CONTINUE
         ENDDO
         IW1 = IW1+NATOMS
         IW0 = IW0+NATOMS
         IW2 = IW2+NATOMS
   1000  CONTINUE
         ...
        ENDDO
        RETURN
        END
```

MDG-INTERF_do1000 contains a series of reductions on the three arrays FX, FY, and FZ[40]. After induction solution and loop normalization, the above code becomes:

```
        SUBROUTINE INTERF(X, Y, Z, FX, FY, FZ, XM, YM, ZM, VIR)
        ...
        IW1 = 1
        IW0 = 2
        IW2 = 3
  CSRD$ LOOPLABEL 'INTERF_do1000'
        DO I = 1, NMOL1, 1
         JW1 = 1+(-1)*NATOMS+I*NATOMS
         JW0 = 2+(-1)*NATOMS+I*NATOMS
         JW2 = 3+(-1)*NATOMS+I*NATOMS
  CSRD$ LOOPLABEL 'INTERF_do1100'
         DO J = 1, NMOL+(-1)*I, 1
          JW1 = 1+(-1)*NATOMS+I*NATOMS+J*NATOMS
          JW0 = 2+(-1)*NATOMS+I*NATOMS+J*NATOMS
          JW2 = 3+(-1)*NATOMS+I*NATOMS+J*NATOMS
```

[40]The pattern exhibited on the array FX is repeated twice more for FY and FZ

```
         ...
         IF ((-9)+KC.NE.0) THEN
          ...
          FX(2+(-1)*NATOMS+I*NATOMS) = FX(2+(-1)*NATOMS+I*NATOMS)+GG(11)+
     *GG(12)+G110+C1*G23
          FX(2+(-1)*NATOMS+I*NATOMS+J*NATOMS) = FX(2+(-1)*NATOMS+I*NATOMS
     *+J*NATOMS)+(-1)*GG(13)+(-1)*GG(14)+(-1)*G110+(-1)*C1*G45
          TT1 = GG(1)*C2
          TT = TT1+C2*G23
          FX(1+(-1)*NATOMS+I*NATOMS) = FX(1+(-1)*NATOMS+I*NATOMS)+GG(4)+G
     *G(6)+GG(7)+GG(13)+TT
          FX(3+(-1)*NATOMS+I*NATOMS) = FX(3+(-1)*NATOMS+I*NATOMS)+GG(5)+G
     *G(8)+GG(9)+GG(14)+TT
          TT = TT1+C2*G45
          FX(1+(-1)*NATOMS+I*NATOMS+J*NATOMS) = FX(1+(-1)*NATOMS+I*NATOMS
     *+J*NATOMS)+(-1)*GG(2)+(-1)*GG(6)+(-1)*GG(8)+(-1)*GG(11)+(-1)*TT
          FX(3+(-1)*NATOMS+I*NATOMS+J*NATOMS) = FX(3+(-1)*NATOMS+I*NATOMS
     *+J*NATOMS)+(-1)*GG(3)+(-1)*GG(7)+(-1)*GG(9)+(-1)*GG(12)+(-1)*TT
          ...
         ENDIF
 1100    CONTINUE
        ENDDO
        IW1 = 1+I*NATOMS
        IWO = 2+I*NATOMS
        IW2 = 3+I*NATOMS
 1000   CONTINUE
       ENDDO
       ...
       RETURN
       END
```

As discussed in Section 3.1, the cross-iteration dependences resulting from the induction variables have been removed.

The following code is generated when using calls to the custom locking routine TAS to solve the histogram reduction on the array FX[41]:

```
       SUBROUTINE INTERF(X, Y, Z, FX, FY, FZ, XM, YM, ZM, VIR)
       ...
       CALL TAS((-1), 3+(-1)*NATOMS+NATOMS*NMOL)
C$DOACROSS LOCAL(GG,G110,K,J,IO,TT1,YL,TT,XL,ZL,RS,RL,G23,G45,KC,FTEMP,
C$& FF,I),SHARE(NMOL1,NMOL,XM,X,NATOMS,BOXH,BOXL,YM,Y,ZM,Z,CUT2,REF4,
C$& QQ4,REF2,QQ2,REF1,QQ,B1,AB1,B2,AB2,B4,AB4,B3,AB3,C1,FX,C2,FY,FZ),
C$& REDUCTION(VIR)
```

[41]This code example is targeted for the SGI Challenge at NCSA. The DOACROSS directive on the Challenge series subsumes the DOALL model of parallel execution

```
CSRD$ LOOPLABEL 'INTERF_do1000'
      DO I = 1, NMOL1, 1
CSRD$ LOOPLABEL 'INTERF_do1100'
       DO J = 1, NMOL+(-1)*I, 1
        ...
        IF ((-9)+KC.NE.0) THEN
         ...
         G110 = GG(10)+GG(1)*C1
         G23 = GG(2)+GG(3)
         G45 = GG(4)+GG(5)
         CALL TAS(1, 2+(-1)*NATOMS+I*NATOMS)
         FX(2+(-1)*NATOMS+I*NATOMS) = FX(2+(-1)*NATOMS+I*NATOMS)+GG(11)+
     *GG(12)+G110+C1*G23
         CALL TAS(0, 2+(-1)*NATOMS+I*NATOMS)
         CALL TAS(1, 2+(-1)*NATOMS+I*NATOMS+J*NATOMS)
         FX(2+(-1)*NATOMS+I*NATOMS+J*NATOMS) = FX(2+(-1)*NATOMS+I*NATOMS
     *+J*NATOMS)+(-1)*GG(13)+(-1)*GG(14)+(-1)*C1*G45+(-1)*G110
         CALL TAS(0, 2+(-1)*NATOMS+I*NATOMS+J*NATOMS)
         TT1 = GG(1)*C2
         TT = TT1+C2*G23
         CALL TAS(1, 1+(-1)*NATOMS+I*NATOMS)
         FX(1+(-1)*NATOMS+I*NATOMS) = FX(1+(-1)*NATOMS+I*NATOMS)+GG(4)+G
     *G(6)+GG(7)+GG(13)+TT
         CALL TAS(0, 1+(-1)*NATOMS+I*NATOMS)
         CALL TAS(1, 3+(-1)*NATOMS+I*NATOMS)
         FX(3+(-1)*NATOMS+I*NATOMS) = FX(3+(-1)*NATOMS+I*NATOMS)+GG(5)+G
     *G(8)+GG(9)+GG(14)+TT
         CALL TAS(0, 3+(-1)*NATOMS+I*NATOMS)
         TT = TT1+C2*G45
         CALL TAS(1, 1+(-1)*NATOMS+I*NATOMS+J*NATOMS)
         FX(1+(-1)*NATOMS+I*NATOMS+J*NATOMS) = FX(1+(-1)*NATOMS+I*NATOMS
     *+J*NATOMS)+(-1)*GG(2)+(-1)*GG(6)+(-1)*GG(8)+(-1)*GG(11)+(-1)*TT
         CALL TAS(0, 1+(-1)*NATOMS+I*NATOMS+J*NATOMS)
         CALL TAS(1, 3+(-1)*NATOMS+I*NATOMS+J*NATOMS)
         FX(3+(-1)*NATOMS+I*NATOMS+J*NATOMS) = FX(3+(-1)*NATOMS+I*NATOMS
     *+J*NATOMS)+(-1)*GG(3)+(-1)*GG(7)+(-1)*GG(9)+(-1)*GG(12)+(-1)*TT
         CALL TAS(0, 3+(-1)*NATOMS+I*NATOMS+J*NATOMS)
         ...
        ENDIF
 1100   CONTINUE
       ENDDO
 1000  CONTINUE
      ENDDO
      ...
      RETURN
      END
```

The synchronization primitives used above are a custom implementation for the SGI Challenge series. Suffice it to say that using a vector of locks provided significant speedup over a single lock implementation due to reduction in contention[42].

If, after solving the inductions, either privatization or expansion is used to implement the solution, the index expressions are analyzed to determine the nature of the reduction. In this example, JWO, JW1, and JW2 vary in the innermost (J) loop, so the reduction on FX is recognized as a histogram reduction. The ranges of the six distinct index expressions[43] are merged and the required allocation of scratch memory space made according to the solution method employed.

In the case of privatized reductions, processor private memory is allocated on LSPDM architectures (e.g, the Convex Exemplar). However, the implementation and code generation for such an architecture is still underway at the time of writing, so the concepts will be exemplified using a logically shared, physically shared memory (LSPSM) architecture[44]:

```
      SUBROUTINE INTERF(X, Y, Z, FX, FY, FZ, XM, YM, ZM, VIR)
      ...
      ALLOCATE (FX0(1:3+(-1)*NATOMS+NATOMS*NMOL, 1:CPU_COUNT_()))
      ...
C$DIR FORCE_PARALLEL
      DO PROCIN = CPUVAR, 1, -1
C Loop prologue -- Initialization
       VIR0 = 0.0
       DO TPINIT = 1, 3+(-1)*NATOMS+NATOMS*NMOL, 1
        FX0(TPINIT, GETCID()+1) = 0.0
       ENDDO
       ...
       TPSIGN = ISIGN(1, 1)
       II1 = TPSIGN+((-1)+NMOL1)/CPUVAR
       II2 = 1+(-1)*II1+II1*PROCIN
CSRD$ LOOPLABEL 'INTERF_do1000'
       DO I = II2, MIN0(TPSIGN*(II2+(II1-TPSIGN)), TPSIGN*NMOL1)/TPSIGN,1
CSRD$ LOOPLABEL 'INTERF_do1100'
        DO J = 1, NMOL+(-1)*I, 1
          ...
         IF (KC+(-9).NE.0) THEN
          ...
```

[42]Speedups will be analyzed in the following section 4 on Evaluation
[43]IWO, IW1, IW2, JWO, JW1, JW2
[44]This code example is targeted for the Convex C3880 at NCSA

```
          G110 = GG(10)+GG(1)*C1
          G23 = GG(2)+GG(3)
          G45 = GG(4)+GG(5)
          FXO(2+(-1)*NATOMS+I*NATOMS, GETCID()+1) = FXO(2+(-1)*NATOMS+
     *I*NATOMS, GETCID()+1)+GG(11)+GG(12)+G110+C1*G23
          FXO(2+(-1)*NATOMS+I*NATOMS+J*NATOMS, GETCID()+1) = FXO(2+(-1
     *)*NATOMS+I*NATOMS+J*NATOMS, GETCID()+1)+(-1)*GG(13)+(-1)*GG(14)+(-
     *1)*C1*G45+(-1)*G110
          TT1 = GG(1)*C2
          TT = TT1+C2*G23
          FXO(1+(-1)*NATOMS+I*NATOMS, GETCID()+1) = FXO(1+(-1)*NATOMS+
     *I*NATOMS, GETCID()+1)+GG(4)+GG(6)+GG(7)+GG(13)+TT
          FXO(3+(-1)*NATOMS+I*NATOMS, GETCID()+1) = FXO(3+(-1)*NATOMS+
     *I*NATOMS, GETCID()+1)+GG(5)+GG(8)+GG(9)+GG(14)+TT
          TT = TT1+C2*G45
          FXO(1+(-1)*NATOMS+I*NATOMS+J*NATOMS, GETCID()+1) = FXO(1+(-1
     *)*NATOMS+I*NATOMS+J*NATOMS, GETCID()+1)+(-1)*GG(2)+(-1)*GG(6)+(-1)
     **GG(8)+(-1)*GG(11)+(-1)*TT
          FXO(3+(-1)*NATOMS+I*NATOMS+J*NATOMS, GETCID()+1) = FXO(3+(-1
     *)*NATOMS+I*NATOMS+J*NATOMS, GETCID()+1)+(-1)*GG(3)+(-1)*GG(7)+(-1)
     **GG(9)+(-1)*GG(12)+(-1)*TT
          ...
         ENDIF
 1100    CONTINUE
        ENDDO
 1000   CONTINUE
       ENDDO
       ...
C Loop epilogue -- Final reduction across processors
C$DIR FORCE_PARALLEL
       DO TPINIT = 3+(-1)*NATOMS+NATOMS*NMOL, 1, -1
        CALL LOCK(LKBYTE(TPINIT))
        FX(TPINIT) = FX(TPINIT)+FXO(TPINIT, GETCID()+1)
        CALL LOCK(LKBYTE(TPINIT))
       ENDDO
       ...
      ENDDO
      DEALLOCATE (FXO)
      ...
      RETURN
      END
```

Privatization has been accomplished by stripmining the original parallel loop and creating a new, enclosing parallel loop. This is necessary on architectures such as the Convex C3880 and SGI Challenge which do not support loop prologues and epilogues[45].

[45] There is a significant effect on performance which can be alleviated by using *expanded* reductions. In fact, the observant reader will notice that the solution actually employs expanded arrays due to the LSPS nature of the architecture

Initially, private storage must be allocated. This is accomplished through the ALLOCATE intrinsic[46]. Upon entering the parallel loop, the preamble is executed. In this example, the preamble need only initialize the private array FX0[47]. In the body of the loop, accesses are made to FX0 using the *id* of the processor currently executing a given iteration[48].

Finally, the actual array FX is updated in the postamble using Convex C3880 synchronization primitives inside a parallel loop[49]. Following this, the private storage is deallocated outside the parallel loop.

The implementation of *expanded* reductions is exemplified by the following extract from code generated for a 32 processor SGI Challenge:

```
      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$DOACROSS LOCAL(I,TPINIT),SHARE(CPUVAR,NATOMS,NMOL,FX0,FY0,FZ0)
      DO I = 1, CPUVAR, 1
       DO TPINIT = 1, 3+(-1)*NATOMS+NATOMS*NMOL, 1
        FX0(TPINIT, I) = 0.0
       ENDDO
       ...
      ENDDO
C$DOACROSS LOCAL(G110,K,J,I0,TT1,TT,G23,G45,KC,FTEMP,FF0,RL0,RS0,ZL0,
C$& XL0,YL0,GG0,I),SHARE(NMOL1,PROCID,NMOL,XM,X,NATOMS,BOXH,BOXL,YM,Y,
C$& ZM,Z,CUT2,REF4,QQ4,REF2,QQ2,REF1,QQ,B1,AB1,B2,AB2,B3,AB3,B4,AB4,C1,
C$& FX0,C2,FY0,FZ0),REDUCTION(VIR)
CSRD$ LOOPLABEL 'INTERF_do1000'
      DO I = 1, NMOL1, 1
       PROCID = MP_MY_THREADNUM()+1
CSRD$ LOOPLABEL 'INTERF_do1100'
       DO J = 1, NMOL+(-1)*I, 1
        ...
        IF (KC+(-9).NE.0) THEN
```

[46]This intrinsic is supported both by Polaris and the Convex FORTRAN compiler. On LSPDM architectures such as the Convex Exemplar this allocation will be done in processor local memory

[47]Note that this is overkill in the sense that we have allocated enough storage in the array FX0 to cover *all* accesses by all processors

[48]While GETCID() on the C3880 does incur the overhead of a function call, on other architectures (such as the SGI Challenge), the id is "globally" accessible

[49]Again, this is overkill because we don't actually use all the locations in FX0, so a simple optimization would be to guard the update with a conditional test for FX0(TPINIT,GETCID()+1) != 0

```
          ...
          G110 = GGO(10)+GGO(1)*C1
          G23 = GGO(2)+GGO(3)
          G45 = GGO(4)+GGO(5)
          FX0(2+(-1)*NATOMS+I*NATOMS, PROCID) = FX0(2+(-1)*NATOMS+I*NATOM
      *S, PROCID)+GGO(11)+GGO(12)+G110+C1*G23
          FX0(2+(-1)*NATOMS+I*NATOMS+J*NATOMS, PROCID) = FX0(2+(-1)*NATOM
      *S+I*NATOMS+J*NATOMS, PROCID)+(-1)*GGO(13)+(-1)*GGO(14)+(-1)*G110+(
      *-1)*C1*G45
          TT1 = GGO(1)*C2
          TT = TT1+C2*G23
          FX0(1+(-1)*NATOMS+I*NATOMS, PROCID) = FX0(1+(-1)*NATOMS+I*NATOM
      *S, PROCID)+GGO(4)+GGO(6)+GGO(7)+GGO(13)+TT
          FX0(3+(-1)*NATOMS+I*NATOMS, PROCID) = FX0(3+(-1)*NATOMS+I*NATOM
      *S, PROCID)+GGO(5)+GGO(8)+GGO(9)+GGO(14)+TT
          TT = TT1+C2*G45
          FX0(1+(-1)*NATOMS+I*NATOMS+J*NATOMS, PROCID) = FX0(1+(-1)*NATOM
      *S+I*NATOMS+J*NATOMS, PROCID)+(-1)*GGO(2)+(-1)*GGO(6)+(-1)*GGO(8)+(
      *-1)*GGO(11)+(-1)*TT
          FX0(3+(-1)*NATOMS+I*NATOMS+J*NATOMS, PROCID) = FX0(3+(-1)*NATOM
      *S+I*NATOMS+J*NATOMS, PROCID)+(-1)*GGO(3)+(-1)*GGO(7)+(-1)*GGO(9)+(
      *-1)*GGO(12)+(-1)*TT
          ...
         ENDIF
 1100    CONTINUE
        ENDDO
 1000   CONTINUE
       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
```

As mentioned above, the initialization section is executed in parallel[50]. In the original (now parallel) loop, accesses to FX are now made to a "private" section of the global FX0. The resulting "private" sums are then summed across processors in the sequential loop which follows[51].

[50]This could be further optimized by merging the three inner loops (not shown)
[51]As noted previously, this nest also can be executed in parallel if the loops are interchanged

A few words are in order as to how the ranges of the six reductions are determined and merged. This is a non-trivial problem in symbolic analysis, and is currently handled by new techniques developed in [6] known as *Symbolic Range Propagation*[52]. The range of each reduction statement is determined using interprocedural symbolic analysis, and the resulting ranges are merged to give an overall range of $1 : 3 + (-1) * NATOMS + NATOMS * NMOL$ for the six statements. This merged range is then used to allocate the correct amount of memory for the expanded[53] array FX0. It is also used as the loop upper bound in the initialization and final reduction phases prior to and following the parallel loop.

[52] Discussed earlier in section 3.1.5
[53] or privatized

CHAPTER 4

EVALUATION

This section will discuss two codes taken from the Perfect Club Benchmark suite and one additional code which is a candidate for membership in the newly created HPSC/SPEC (High Performance Steering Committee SPEC) suite.

I will briefly introduce each of these codes. From the Perfect Club Benchmark suite, we have TRFD, and MDG. TRFD is a kernel 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 of up to 256^3 double precision (64 bit) elements in size[1].

Important[2] 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 4.1: Speedups over serial execution time on an 8 processor set on an SGI Challenge

[1]The problem size used for all timings reported here is 64^3
[2]Important in terms of overall program execution time

4.1 Timing results on the SGI Challenge

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

	wallclock seconds	speedup
serial	194.029	1
SGI (PFA, default optimization)	98.658	1.967
SGI (PFA, aggressive optimization)	187.394	1.035
Polaris (with transformations)	30.408	6.380
Polaris (without reduction transformation)	1105.690	(-5.698)
Polaris (without induction transformation)	1671.404	(-8.614)

Table 4.2: Overall Program Timings and Speedup for MDG

	wallclock seconds	speedup
serial	21.15	1
SGI (PFA, default optimization)	95.786	(-4.529)
SGI (PFA, aggressive optimization)	108.712	(-5.140)
Polaris (with both transformations)	7.612	2.778
Polaris (without reduction transformation)	7.670	2.757
Polaris (without induction transformation)	226.078	(-10.69)

Table 4.3: Overall Program Timings and Speedup for TRFD

	wallclock seconds	speedup
serial	156.716	1
SGI (PFA, default optimization)	45.842	3.419
SGI (PFA, aggressive optimization)	36.204	4.328
Polaris (with reduction transformation)	31.872	4.917
Polaris (without reduction transformation)	44.698	3.506

Table 4.4: Overall Program Timings and Speedup for TURB3D

The above timings are wall-clock execution time averaged over five executions made in “flash” mode[3,4]. Parallel processes are executed on a *processor set* consisting of 2, 4 or 8 processors[5], and

[3]Flash mode sets the master processes' priority to a non-degrading maximum which is inherited by all spawned (slave) processes [9]

[4]Timing was performed by the *timex* command which reports wall-clock, user, and system times to 1/100 second

[5]For background on processor sets see [15]

all processes were set to spin-wait (rather than block) while waiting for a parallel section to begin execution[6]. In the above tests all parallel timing was conducted on an 8-processor processor set with the exception of the single processor times. These were obtained by running on a two-processor processor set and disabling migration between processors (i.e., executing on only a single processor)[7].

For comparison purposes, the three benchmarks under discussion were compiled and executed using the commercially available SGI product Power Fortran Accelerator, PFA (see [9] for detail).

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"[8]. A second batch of runs was made using PFA with optimization switches "-ag=a -r=3" to provide a comparison with a commercial product using more aggressive optimizations[9].

4.2 Discussion of the Major Loops

The table below shows the total amount of wall-clock time in seconds spent in the major loops of these three codes:

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

Instrumentation for the results displayed in Table 4.5 was performed by the Polaris utility p-instrument, which instruments on a loop-by-loop basis by enclosing outermost parallel loops. The timing

[6]This eliminates startup overhead which would otherwise be incurred by the master on behalf of the slaves

[7]This had the desired effect of maximizing cache utilization during the single processor runs

[8]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

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

for the loop-by-loop results was done by *etime*[10], called from an interval library developed at CSRD and ported to the Challenge.

4.2.1 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 special handling required for these reductions is discussed in Section 3.2.4. However, it should be noted that the near linear speedup achieved for this loop is attributable to the solution of the additive inductions (Section 3.1) in conjunction with reduction recognition and code generation. These results are based on *expanded reductions*, and as expected tests made using a vector of locks around each reduction site resulted in a marked decrease in speedup[11].

To understand the impact of these two transformations, separate timings were made by turning off each optimization in turn (these timings are reported in Table 4.2). In this case, when either parallel reduction solution or induction variable substitution is turned off, this loop is not parallelized. The timing results show the devastating effects which result.

As an aside, it's interesting to note that the execution time for solving only reductions was 33% longer than that when solving only inductions. The cause for this is found in the fact that several small trip count loops in do1000 which contain reductions were parallelized, incurring a large degree of overhead[12]. Heuristics to handle this situation better are under development.

[10] accurate to 1/100 of a second [9]
[11] On the order of 4, versus the 6.3 reported above
[12] These inner loops are of course not parallelized when do1000 is parallelized

4.2.2 MDG-POTENG_do2000

This loop accounts for less than 2% of the execution time, however it too has a very good speedup. Like INTERF_do1000, this loop also contains a battery of six interdependent additive inductions, the solution of which is a prerequisite to DOALL parallelization.

Similar observations hold in terms of the speedups when not using the two transformations as outlined above for INTERF_do1000.

4.2.3 TRFD-OLDA_do100

This is the first loop of three contained in the subroutine OLDA, and consumes over 65% of the execution time. Discussed in Section 3.1, this loop contains three induction variables and, in addition, *histogram reductions* occur at two different sites. Due to the solution of these complex *triangular* inductions, the reductions are solvable using *array privatization* ([23]) and no further reduction processing is necessary for DOALL parallelism in this loop[13].

The separate timings resulting from turning off each optimization in turn hold no surprises; although do100 does contain reductions, as noted in Table 1.1 in Section 1, these only become important if the inductions are not solved. However, although these reductions are solved when induction solution is turned off, as demonstrated by the results in Table 4.3 there is not enough work contained in these parallel loops to offset the cost of parallel execution.

4.2.4 TRFD-OLDA_do300

In some sense OLDA_do300 is the flagship of the induction substitution pass, because it is here that 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[14]. In addition, similar to OLDA_do100, two *histogram*

[13] Because the inner loops containing the reductions are now executed serially inside the enclosing parallel loop
[14] Coupled to a *triangular* induction variable, which makes the closed form a quartic

reductions occur in this loop. Like OLDA_do100, induction variable solution paves the way for array privatization, and not only is the loop executed with DOALL parallelism, but as was the case with OLDA_do100, no special reduction processing is necessary.

The speedup of these two loops is not as impressive as 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.

4.2.5 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, *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.

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

It is also interesting to note that this same reduction was solved by PFA using the more aggressive compilation options "-ag=a -r=3", which allow transformations based on associativity. The resulting speedups for PFA as reported in Table 4.4 reflect this fact, as well as the fact that the performance of Polaris was slightly better. While not confirmed at this time, the reason for this result is most likely due to Polaris' greater success in identifying parallel loops.

CHAPTER 5

RELATED WORK

In [11], induction variable substitution and parallel reductions were recognized as two of four performance-critical techniques necessary for the parallelization of the Perfect Club Benchmark codes. This section deals with several alternative approaches to the solution of these two problems.

At a high level, all these approaches fall under the category of symbolic analysis. Harrison and Ammarguellat use abstract interpretation to map each variable assigned in a loop to a symbolic form (expression), and match these against template patterns which represent commonly occurring recurrences ([1]). Haghighat and Polychronopoulos symbolically execute loops and use finite difference methods in conjunction with interpolation to determine the closed form for a given recurrence ([17]). Wolfe *et al* derive relations between variables by matching against cycles in the SSA graph, and then use matrix inversion (among other methods) to determine the closed forms ([13]).

5.1 Symbolic Execution

While we have logically separated recognition and closed-form solution, Haghighat's approach unifies these two phases of the algorithm. The methods employed first perform preprocessing, which determines an *abstract model* for each reducible loop in the flowgraph ([17]). Based on this model, a loop is symbolically executed and recurrence relations determined for families of induction variables. These relations are then solved using a finite differencing scheme. The differencing scheme is based on a user-determinable heuristic which sets an upper bound on the depth of the difference table, thereby limiting the degree of the interpolating polynomial[1]. Using Newton's forward formula, the difference table is used for interpolation and the characteristic function (i.e., the closed form) is determined.

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.

5.2 Abstract Interpretation

Harrison and Ammarguellat's approach 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[2].

Once the abstract stores have been composed across a loop body, the maps are unified with user-defined templates which 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 ([18]). Thus, although it appears that the framework is capable of supporting the solution of inductions in multiple loops, it has not been implemented.

A second feature of [1] 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

[1]The default value is three

[2]This stage is very similar in function to the Recognition phase described in Section 3.1.3

variety of recurrence relations, appropriate to the particular language or applications being compiled." ([1], page 10).

5.3 SSA Based Classification

Wolfe *et al* base their approach on the *Static Single Assignment* representation of the program. The algorithm breaks induction variable substitution down into two phases similar to the first two phases of our algorithm[3]. During the recognition (and classification) stage, the SSA form of the graph is searched for patterns representing recurrences[4]. 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[5].

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[6], varying from matrix inversion to symbolic execution of the loop body[7].

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 *a priori* knowledge of classes of sequence variables matched against the SCC under

[3]The current implementation of their work does not do substitution [27]

[4]This operates in a manner similar to that we have implemented for the detection of wrap-around loop bounds in Section 3.1.6

[5]μ and ϕ functions are distinguished in a way similar to [25]. μ is a special case of ϕ that has an arity of two with reaching definitions coming from outside and inside the loop respectively

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

[7]Used in the solution of non-constant *periodic sequences*

investigation. This is the technique we have employed to recognize wrap-around loop bounds (discussed in section 3.1.6).

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 3.1.9 ([27]).

CHAPTER 6

CONCLUSION

The goal of the Polaris project has been to parallelize real codes and achieve real speedups on shared memory architectures. To that end we have focussed on two techniques which were clearly identified as important in [11] – induction variable substitution and parallel reductions.

The approaches discussed in the Section 5 solve recurrence relations in one degree or another, but none of the reported work has demonstrated actual speedups on real benchmark codes. It is difficult to determine the actual benefit of a transformation without the ability to conduct empirical tests. Thanks to the integration of the *Range Test*, *array privatization*, the *internal representation*, and many other tools and utilities, with *induction variable substitution* and *reduction recognition* in Polaris, it has become possible to measure performance on *real* parallel machines ([4],[5],[12],[25]).

Although the results for only three codes have been presented above, the evaluation of the benefit of these two transformations is being measured on a test suite which includes candidate codes from the High Performance SPEC Benchmark Suite, two Grand Challenge codes, and six codes from the Perfect Club Benchmark suite. Thanks to the near linear speedups achieved in MDG and significant speedups in other codes, there is reason to hope that the analysis of the remaining benchmarks will bear similar results.

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.

[6] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. Technical Report 1381, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, 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] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. *Proceedings of ACM Symposium on the Priniciples of Programming Languages*, 1989.

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

[10] 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.

[11] 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.

[12] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. Technical Report 1317, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. and Dev, October 1993.

[13] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driver ssa form. *To appear in TOPLAS*, August, 1994.

[14] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, New York, 1990.

[15] Anoop Gupta, Andrew Tucker, and Luis Stevens. Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach. Technical Report CSL-TR-91-475A, Stanford University, July 1991.

[16] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs. *Proceedings of the Sixth Annual Languages and Compilers for Parallelism Workshop, Portland, Oregon*, August 1993.

[17] 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.

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

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] Peng Tu and David Padua. Efficient Building and Placing of Gating Functions. Technical Report 1389, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., November 1994.

[26] 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.

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