Detecting Induction Variables using SSA Form

Michael P. Gerlek

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 93-014

June 1993

Detecting Induction Variables using SSA Form*

Michael P. Gerlek

Department of Computer Science and Engineering

Oregon Graduate Institute

7 May 1993

revised 7 June 1993

Abstract

The detection of induction variables for the purpose of strength reduction is a well-known compiler optimization. Traditionally, however, this optimization has been restricted to simple linear expressions. In this paper we present the use of a fast and efficient algorithm based on the Static Single Assignment (SSA) form which detects linear induction expressions as well as several types of more complex, non-linear expressions. We present some details of the implementation of SSA-based induction variable analysis in our Nascent Fortran compiler, and present experimental results showing that this extended induction expression classification scheme can enhance data dependence analysis as well as provide more opportunities for strength reduction.

*OGI-CSE technical report 93-014

1 Introduction

To achieve high performance on modern pipelined and parallel architectures, compilers must take advantage of advanced optimization techniques such as vectorization, loop transformation, and loop distribution. Data dependence analysis is required to determine the validity of these optimizations: specifically, array subscript expressions within loops must be analyzed and classified to determine the dependence relationships that may exist. Often these subscripts are simply linear induction variables (e.g. a loop counter), but they may be of more complex forms involving expressions of mutually defined induction variables, polynomial sequences, periodic sequences, and so on.

The detection of induction variables (IVs) for the purpose of strength reduction is a well-known compiler optimization. Traditionally, however, this optimization has been restricted to relatively simple linear expressions detected by pattern matching techniques on the intermediate representation of the program. In [Wol92], Wolfe presented a new technique to detect IVs based on the Static Single Assignment (SSA) form of the program. This new algorithm detects all linear induction expressions as well as several types of more complex, non-linear expressions in one unified framework.

We have implemented SSA-based induction variable analysis in our Nascent Fortran compiler. In this paper we will present some of our experiences in implementing the algorithm for full analysis; in particular, it is clear that a compiler requires several features in order to detect some of the complex induction expressions encountered "in the wild" and addressed in the literature. We also present some experimental results showing the classifications of subscript expressions in typical scientific Fortran applications, as determined by our analysis. This data agrees with previous reported results and provides useful information for data dependence solvers. Finally, although our focus is on determining induction expressions for data dependence solvers in our compiler, SSA-based analysis can also be used for strength reduction.

The rest of this paper is divided as follows. In section two, we introduce induction expressions in more detail and then describe SSA form to show how it is used to find

induction expressions. In the third section, we present an example in some detail and point out key extensions to the algorithm originally described. In the fourth section, we present some experimental results showing the frequencies of various types of subscript expressions – constants, linear induction expressions, etc – in common scientific codes. In section five we discuss related work and then present some final conclusions in section six.

2 Induction Variables and SSA Form

2.1 Induction Variables

Intuitively, a *basic induction variable* is a variable that is assigned in a loop and incremented by a constant amount on every iteration [ASU86]. More generally, an induction variable can be defined in terms of itself and some linear combination of constants and other induction variables.

Consider the following loop:

```
        i := j := k := l := 1
L1:     loop
            i := i + 3
            j := k + n
            k := j + 1
            m := t + 4*i
            A[m+1] := ...B[m] ...
        endloop
```

The definition of `i` in loop `L1` is a linear induction variable: its starting value is 4, and on each subsequent iteration it is incremented by 3. The variables `j` and `k` are *mutual* induction variables [ACK81], since they are defined in terms of each other: `j` and `k` have initial values of `n+1` and `n+2`, respectively, and both are incremented by the loop-invariant value `n+1` on each subsequent iteration. The variable `m` is not incremented on each iteration but since it is assigned a linear function of another induction variable, `i`, and a loop invariant value, `t`, it is a *derived* induction variable. This implies the subscript expressions of `A` and `B` are *induction expressions.*

We will represent linear induction variables using the notation $c + ah_L$, where h is the *basic loop counter* for loop number L, i.e. on the first iteration h has value 0 and increments by 1 one thereafter, and c is the initial value of the variable. In the above example, `i` $= 4 + 3h_1$ and `k` $= 2 +$ `n` $+ (\mathtt{n} + 1)h_1$. We will omit the loop number where the meaning is clear.

2.2 Static Single Assignment Form

Our method for discovering induction expressions relies on using the Static Single Assignment (SSA) form as an intermediate representation of programs [CFR+91]. When converted to SSA form, a program has the property that each use of a variable has exactly one reaching definition.

A program in SSA form is represented by renaming variables at their definition points so that each variable will have a unique name when it has a different value. A merge operator, called a ϕ-function, is inserted for a variable at a merge point in the control flow graph (CFG) if a variable has more than one distinct reaching definition at that point. The arguments to the function correspond to the reaching definitions from each predecessor of the merge point. We represent the modified program by labeling each definition of a variable with a unique subscript and each use with the appropriate subscript for the reaching definition. In figure 1, we show a very simple program before and after conversion to SSA form.

The *SSA graph* of a program is a directed graph with edges from each vertex, corresponding to operations in our intermediate representation, to each of its arguments. In our representation, the usual arithmetic operators have one or two arguments. For *fetch* operators, two arguments are required: one for the name of the variable and one for the reaching definition. The latter edge, an *ssalink*, is the result of the conversion to SSA form: the sink of an ssalink is a definition point, either a store operator or a ϕ-function. *Stores* also require two arguments: the name of the variable and the expression to be stored. ϕ-functions have one edge, represented by an ssalink, to each reaching definition at that point. In this imple-

```
      j := 1                          j1 := 1
L2:   loop                       L2: loop
          j := j + 3                      j2 :=φ(j1, j3)
      endloop                             j3 := j2 + 3
                                      endloop
```

Figure 1: Sample program

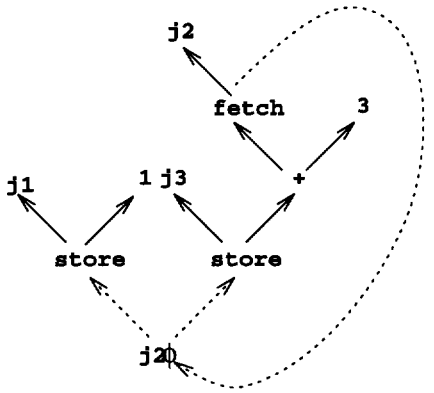


Figure 2: SSA graph of sample program

mentation we consider only integer valued expressions and do not consider array references. Figure 2 shows the SSA graph for the example program (ssalinks are represented by dashed lines).

2.3 Finding Linear Induction Expressions using SSA

In [Wol92] a technique is presented for discovering induction expressions using the SSA graph based on the following observations. By definition, IVs will occur only when there is an assignment to a variable within a loop. In such a case, there will be a *loop header ϕ-function* to merge the two reaching definitions of the variable, the initial value of the variable (from above the loop) and the subsequent value (from within the body of the loop). Also by definition, the assignment to the IV within the body of the loop uses the previous value of the variable, so the fetch in our representation will have an ssalink to the loop header

ϕ-function. Therefore, if there is a cycle in the SSA graph, then the cycle may define an induction variable.

Our algorithm for finding induction variables, then, relies on discovering cycles in the SSA graph. We use Tarjan's algorithm [Tar72] to find strongly connected regions (SCRs), which allows the compiler to perform a depth-first search of the graph, visiting each SCR only when all its descendants have been visited. When an SCR is visited, it is classified as an induction expression, a loop invariant, or some other type of expression, based on the classifications of its child SCRs.

For linear induction variables, the operations within the cycle may consist of fetches, stores, and addition of loop invariant values or other linear variables. Subtraction is also allowed, provided the right operand is not in the cycle (i.e. `i := k - i` is not linear). There must be one ϕ-function at the loop header and no other ϕ-functions. As we will see below, other combinations of operations in the cycle will yield other types of induction expressions.

As an example, consider the graph from above. The search procedure finds the cycle involving `j` and classifies it as a linear induction variable based on the operations within the cycle. By traversing the cycle in reverse starting at the loop header ϕ, the compiler can determine the initial value for each operator and accumulate the total increment in the cycle. The ϕ and the fetch operators have an initial value of 1 inherited from outside the loop, the add operator then assumes the value of $1 + 3 = 4$, and $\mathtt{j}_3$ then has an initial value of 4. The total increment is 3. The linear induction expression for the ϕ-function is $1 + 3h_2$ and the store $\mathtt{j}_3 = 4 + 3h_2$. Had there been a use of `j` below the assignment to $\mathtt{j}_3$, the linear expression $3 + 4h_2$ would be propagated down the ssalink to the fetch of $\mathtt{j}_3$.

2.4 Other Types of Induction Expressions

By construction, every SCR in an SSA graph will have at least one ϕ-function at the loop header[1]. If the SCR contains operations for the addition and subtraction of invariant values as well, a linear induction variable may be described. If these constraints are relaxed, more

[1]We consider only reducible flow graphs.

complex sequences may be classified [Wol92]. We describe these briefly:

- *Polynomial* IVs can arise by incrementing an IV in a loop by another (linear or polynomial) induction variable and may be represented by an expression polynomial in h.
- *Geometric* IVs are produced by assignments of the form `i := i*k`. They are represented by an equation with a term of the form b^h where h is the basic loop counter and b is the multiplicative constant. Expressions of the form `j := c - j` (where `c` is loop invariant) also describe geometric sequences since `c - j` $\equiv$ `-1*j + c`.
- *Wrap-around* variables are variables in loops that take on some value determined from outside the loop on the first n iterations and on all subsequent iterations take on a value that can be represented by an induction expression. Wrap-around expressions occur naturally in programs at the loop header ϕ-functions of derived induction variables.
- *Periodic* variables take on the values in a sequence of n values for the first n iterations of the loop, and on iteration $n + 1$ the sequence repeats; only periodic induction variables have more than one loop header ϕ. When the *period* of the sequence is 2, the periodic value alternates between two values, producing a *flip-flop* variable[2].
- *Monotonic* variables may occur when a variable within a loop is conditionally incremented. If the cumulative effect of the body of the loop is to increment the variable by a constant, the variable may be classified as monotonically *increasing*; if the compiler can determine the variable is always incremented (as in the case where a variable is incremented on both branches of an if-test), the variable may be classified as *strictly* monotonically increasing. Monotonically *decreasing* variables are defined analogously. Only monotonic induction variables have non-loop header ϕ-functions.

3 Implementation Issues and Extensions

To perform SSA-based induction variable analysis on non-trivial loops, several important points must be addressed. For example, the compiler needs to be able to operate on arbitrary symbolic expressions, inter-loop SSA edges must be gated to prevent inter-loop cycles, the compiler must be able to determine tripcounts, and it must make some use of derived assertions to produce the best results. In addition, some of the rules originally given for nonlinear expressions may be extended. In this section, we present a typical loop nest and describe in some detail how our algorithm identifies and classifies the induction expressions within it. We do this for three reasons: one, to provide a full example of the implementation

[2]Flip-flop variables may also be expressed as geometric sequences, using -1 as the base.

of our technique; two, to point out areas we addressed that were not detailed in the original paper; and three, to show our scheme handles real-world examples other researchers have addressed as important.

3.1 An Example from the Benchmarks

Several researchers studying parallelizing compilers have pointed out several specific types of induction variable forms that, while relatively infrequent, can lead to significant parallelism if the subscript expressions can be classified [EHLP92, HP92]. A typical example is a triangular loop containing a polynomial induction variable, similar to the program shown in figure 3 in both Fortran and in SSA form.

To parallelize this loop, the compiler must determine if at any iteration of the `i` and `j` loops, the references to `A(j+1)` and `A(k)` refer to the same element – this is the essence of dependence analysis. It is important, therefore, to be able to characterize `j+1` and `k` as functions of the loop indexes.

The SSA form of this program is also shown in figure 3. Note that because of a technicality of our representation of Fortran DO loops, the loop index variable provides the reaching definition of the variable and not the loop-header ϕ-function; our compiler handles loops defined using traditional gotos and if-tests as well, but our treatment of DO loops makes the presentation simpler.

Our SSA form also includes η-functions, gating the exit values of variables assigned within the loops; this will be explained below. We begin our example by considering only the inner loop.

3.2 The Inner Loop

When considering a loop nest, we must consider any reference from outside of the loop as invariant because we wish to define induction variables in the context of the current loop only. Figure 4 shows the inner loop of the example program.

We may apply Tarjan's algorithm to the nodes of the SSA graph of the inner loop in

```
L3:   do i = 1, n
L4:      do j = 1, i
            A(j+1) = ...
            k = k + 1
         enddo
         ...= A(k)
      enddo
```

```
L3:   do
         i2 = φ(i0, i1)
         i1 = dosequence(1, n0)
         j1 = φ(j0, j3)
         k1 = φ(k0, k4)
L4:      do
            j3 = φ(j1, j2)
            j2 = dosequence(1, i1)
            k2 = φ(k1, k3)
            A(j2+1) = ...
            k3 = k2 + 1
         enddo
         j3 = η(j2)
         k4 = η(k2)
         ...= A(k4)
      enddo
      i3 = η(i1)
      j5 = η(j2)
      k5 = η(k4)
```

Figure 3: Triangular loop nest

```
L4:   do
         j3 = φ(j1, j2)
         j2 = dosequence(1, i1)
         k2 = φ(k1, k3)
         A(j2 + 1) = ...
         k3 = k2 + 1
      enddo
```

Figure 4: The inner loop

any order we choose. Assuming we identify the only nontrivial SCR in the graph first, we identify k as a linear induction variable. In particular, $\mathtt{k}_2 = \mathtt{k}_1 + h_4$ and $\mathtt{k}_3 = 1 + \mathtt{k}_2 + h_4$, where $\mathtt{k}_1$ is loop invariant symbolic expression. The loop index, $\mathtt{j}_2$, is not in an SCR, but DO loop indexes are always linear induction variables – in this case $\mathtt{j}_2 = 1 + h_4$. The ϕ-function at $\mathtt{j}_3$ is actually a wrap-around variable: on the first iteration, it has value $\mathtt{j}_1$ and on each subsequent iteration it takes the value of $\mathtt{j}_2$. The subscript expression for `A(j+1)` is a linear expression, since it is a use of $\mathtt{j}_2$.

3.3 Tripcounts

For practical purposes, we are limited in our ability to determine the tripcounts of loops. The tripcount of Fortran DO loops can be determined at compile-time, although the result may be symbolic and need not be constant: by definition, the Fortran loop `do i=ia,ib,ic` will have a tripcount of `imax(0,(ib-ia+ic)/ic)`. The tripcount of the example inner loop therefore will be $tc_4 = \mathtt{imax}(0, (\mathtt{i}_1 + 1 - 1)/1)$ which is simplified to $tc_4 = \mathtt{imax}(0, \mathtt{i}_1)$ in the internal representation in the compiler.

Because the tripcount is an important factor for solving nested loops, it is advantageous to simplify this expression as much as possible: in some cases our compiler can remove the `imax` expression from the tripcount. The value of $\mathtt{i}_1$ is not known at this point in the analysis, but it is represented internally by a fetch of $\mathtt{i}_1$. If we follow the ssalink of this fetch to its reaching definition, the compiler discovers $\mathtt{i}_1$ is a DO loop index of an outer loop and quick inspection reveals this outer DO loop has a lower bound of 1. The compiler can then assume that if the inner loop has executed, the value of $\mathtt{i}_1$ must be at least 1. The `imax` operator may then be optimized away, setting the tripcount of the inner loop to be $tc_4 = \mathtt{i}_1$. If the target of the ssalink had been a store, our compiler would attempt to find the lower bound for the expression on the right-hand side. No lower bound information is returned if a cycle or a ϕ is encountered.

At present, our compiler treats this lower bound technique as a special case for reducing `imax` expressions, but it is evident that what is actually happening is a demand-driven

forward-substitution of symbolic values. Our technique for finding induction variables inherently performs a style of propagation of symbolic values in loop bodies (and beyond, via η-functions). Although not addressed in the original paper, a demand-driven walk of the SSA graph seems to provide a convenient method of constant propagation. As values from outside the loop are treated as invariants, our current approach will not unify constant propagation and induction variable classification. Our group is currently investigating modifications to the standard SSA form to support aggressive constant propagation [WZ91] and we hope to study the merging of these techniques.

Our studies show roughly three-quarters of the loops in scientific Fortran codes are *DO* loops. For more general loops our strategy to determine the tripcount is to examine the condition controlling whether or not the loop's CFG exit edge is taken. If an induction expression for that condition may be found (e.g. `i<n` may be treated as `i-n`), the number of times the loop executes may be determined. For loops with multiple exits, we do not currently attempt to determine a tripcount. Obviously tripcounts for loops with simple exit conditions like `if(i<n)` where `i` is not an induction expression cannot be determined either. Our compiler does not currently consider complex exit expressions such as `if(i<n OR j<m)`.

3.4 Exit Value Expressions and η-functions

Having classified all expressions in the inner loop and determined its tripcount, we proceed to the next loop level. Here the SCRs for variables `j` and `k` span loop boundaries, however. We wish to consider the effects of the inner loop on these variables as fixed and for this reason we *gate* the *exit value* of each variable assigned within the loop and restrict the walk of the SSA graph from passing through these gates. We are essentially collapsing the effect of the loop body into this exit value gate expression. A need for exit values was pointed out in Wolfe's original paper; in this section we describe our method.

We add a new operator to the traditional SSA-graph, the *η-function*, which is placed in a position immediately after the body of the loop. Ballance et al introduced η-functions

in their Gated Single Assignment (GSA) form with loop predicate information to determine under what conditions the value being gated would be used [BMO90]. Here we adopt the η-function but use it simply as a convenient placeholder for the exit value.

An η-function is inserted after the exit of each loop for every variable assigned within that loop. In particular, for every control flow graph edge that exits a loop, a *postexit* node is inserted as the target of that edge, thus assuring a unique successor for each exit. An edge exiting multiple loops requires only one postexit. Postexit nodes, analogous to loop preheaders, are inserted in our compiler during loop discovery phase.

The η-functions themselves are created as part of our SSA translation phase [CFR+91]. As the first step of the SSA algorithm, all variable definitions are marked. During this phase, we create an η-function for variable `x` in each postexit node of loop `L` if there is at least one definition of `x` within the body of `L`. We then mark the η as *both* a use and a definition of `x`. As the SSA algorithm proceeds, it creates ssalinks from uses to definitions; thus, a use of `x` from a position outside of `L` will have an ssalink to the η as its reaching definition and the η itself will have an ssalink to the reaching definition in `L`. The insertion of an η for `x` is essentially an insertion of the assignment `x := x`.

As described above, our goal is to use η-functions as placeholders for an expression representing the exit value of a variable assigned within a loop. The exit value will be either a constant or a symbolic expression, depending on the classification of the variable (linear expression, integer constant, etc), its value prior to the loop, and the ability of the compiler to determine the tripcount of the loop. If the variable of the η is an induction expression its exit value is a function of the tripcount of the loop, and the compiler performs symbolic algebra to "solve" the induction expression for the tripcount. If the tripcount is unknown, the exit value is undefined. It is important to note that the exit value of a variable will be an expression in terms of the current loop or the outermost level, not from within the loop the η is gating.

When performing our search of the SSA graph, if we encounter an η-function we do not follow its ssalink into the inner loop, as stated above. Instead, at that point we derive

the exit value of the gated variable and translate its symbolic representation into operators in our intermediate form. The resulting tree is then used as the target of the η-function's *value edge*. Our search then resumes, walking up the value edge.

In the example, when the η for $\mathtt{j}_2$ is reached, the compiler must determine the exit value of $\mathtt{j}_2$ after the iterations of L4. Owing to the semantics of DO loops, the occurrence of the assignment at the DO dominates the exit node in the flow graph, thus the tripcount of $\mathtt{j}_2$ is actually one more than tc_4. The exit value for $\mathtt{j}_2$ is calculated as shown. We use the notation $a@b$ to mean "the value of expression a after b iterations": in particular, since h starts at zero, $h@n = n - 1$.

$$\begin{aligned} j_3 &= j_2@(tc_4 + 1) \\ &= (1 + h_4)@(i_1 + 1) \\ &= (1@(i_1 + 1)) + (h_4@(i_1 + 1)) \\ &= 1 + (h_4@(i_1 + 1)) \\ &= 1 + i_1 \end{aligned}$$

The exit value expression for $\mathtt{j}_2$ is then $\mathtt{1} + \mathtt{i}_1$. This is translated into a fetch of $\mathtt{i}_1$ added to the integer constant 1 and connected to the η-function at $\mathtt{j}_3$.

By a similar process, $\mathtt{k}_4$ is set as the exit value expression $\mathtt{k}_2@(\mathtt{tc}_2 + \mathtt{1})$, which reduces to $\mathtt{k}_1 + \mathtt{i}_1$.

3.5 The Outer Loop

After the insertion of exit values, the outer loop has been transformed into figure 5. The compiler searches the SSA graph for the body of this loop, classifying $\mathtt{i}_1$ as a linear induction variable, $\mathtt{i}_2$ as a wrap-around variable, and $\mathtt{j}_3$ as a derived induction variable.

The SCR containing variable k, however, contains an addition of a loop invariant value ($\mathtt{k}_1$) and a linear induction expression ($\mathtt{i}_2$). As described in [Wol92], the compiler simulates the first 4 iterations of the loop to determine the values that the ϕ, k1, takes on. Initially, $\mathtt{k}_1@0 = \mathtt{k}_0@0 = \mathtt{k}_0$. On the next iteration, $\mathtt{k}_1 = \mathtt{k}_1@0 + \mathtt{i}_2@1$, which simplifies to $\mathtt{1} + \mathtt{k}_0$. The next two iterations produce $\mathtt{3} + \mathtt{k}_0$ and $\mathtt{6} + \mathtt{k}_0$. These expressions are then used to

```
L3:   do
          i2 = φ(i0, i1)
          i1 = dosequence(1, n0)
          j1 = φ(j0, j3)
          k1 = φ(k0, k4)
          ...
          j3 = η(1 + i1)
          k4 = η(k1 + i1)
          ...= A(k4)
      enddo
```

Figure 5: Outer loop, with exit value expressions

solve a system of polynomial equations, producing the expression $\mathtt{k}_0 + h_3/2 + h_3^2/2$. The induction expression for the subscript of A is defined by $\mathtt{k}_4$, so the subscript is represented by the polynomial equation $\mathtt{i}_2 + \mathtt{k}_0 + h_3/2 + h_3^2/2$. Since $\mathtt{i}_2$ is equal to $1 + h_3$, the polynomial is simplified to $\mathtt{k}_0 + 3h_3/2 + h_3^2/2 + 1$.

The example concludes by determining the η values for i, j, and k. Variable i exits with the value $\mathtt{imax}(1, \mathtt{n}_0 + 1)$ and k with an expression polynomial in $\mathtt{n}_0$. The exit value of j is unknown since the induction expression for $\mathtt{j}_5$ is a wrap-around variable. The tripcount tc_1 is not known to be greater than 1, so no simpler expressions can be produced. We attempt to use the same lower bound information we described earlier to determine if the tripcount indicates whether the initial value or the subsequent induction expression in the wrap-around variable is being used, and also in the **imax** expression, but in this case no information about n_0 is known.

3.6 Generalized Non-Linear Form

Linear induction variables are discovered by looking for strongly connected regions in the SSA graph. In fact, however, the SCR for a linear induction variable is actually a simple cycle; the algorithm used to determine the contribution of the cycle within the body of the loop implicitly takes advantage of this fact. The region will not be a cycle only if the

assignment to the variable contains the variable on the right hand side more than once or there is a non-loop-header ϕ in the region. In the latter case, the region may describe a monotonic variable. The former case is demonstrated by the fact that `i := i + i` is actually a geometric induction variable, since it is equivalent to `i := i*2`.

The original rule presented for determining that an SCR represents a geometric sequence must be extended. In addition to the condition that the cycle may contain a multiplication by a known integer, a geometric induction variable may occur when the variable carried around the loop occurs more than once on the right hand side of the assignment. This extension requires some modification to the original technique for classifying geometric induction variables.

The general method for determining equations for geometric induction variables proceeds by visiting each node in the SCR and accumulating its contribution to the variable. The compiler must find the first $n+1$ values of the variable by symbolically interpreting the loop, where n is the degree of the equation representing the sequence. The equation is then determined by solving a system of equations including a geometric term for the $n+1$ values. The base of the geometric term is precisely the factor by which the induction variable is multiplied. This process is performed symbolically, so we can also relax the restriction that the factor must be an integer; it need only be loop-invariant.

For an assignment of the form `i := i + i`, however, there is no multiplication, so the base factor of 2 is not immediately evident. A compiler may try to express this by rearranging and collecting terms in the intermediate representation. A simpler trick is to abstractly interpret the expression, collecting factors of the induction variable in the expression tree of the right hand side of the assignment.

Starting at the ϕ-function, we visit each node in the SCR in reverse order and assign it a *factor value*. For a *fetch* in the SCR (which must be a fetch of the induction variable), we assign a factor value of 1. For addition operators, we assign a factor value equal to the sum of the factor values of the parents, using a value of 0 if the parent is not in the SCR. For multiplication operations, we similarly assign a factor value equal to the product of

the parents' values. The factor value assigned the ϕ-function is the factor value of the ϕ's ssalink in the SCR. This factor is then used as the base in the geometric term.

In our implementation, multiple passes through the SCR nodes may be required before the factor value of the loop header ϕ is set. The list of nodes in the SCR is maintained as a stack in Tarjan's algorithm and does not necessarily correspond to a breadth-first traversal. As a result, we may attempt to set the factor for a node before we have visited both its parents. The maximum number of passes required is small, as it is equal to the number of ssalinks to the loop header ϕ, corresponding to the number of occurrences of the variable in the expression.

As an example, in figure 6 we show the SSA graph for the loop

```
        g0 := 1
L5:     loop
            g1 := φ(g0, g2)
            g2 := 5 * g1 - (2 + g1)
        endloop
```

The factor values assigned each node are shown next to the operations. The final value produced is 4, which is correct since `5*g-(2+g)` $\equiv$ `4*g-2`.

3.7 Other Implementation Details

As described in the original paper, SSA-based analysis can also be used to classify monotonic variables, i.e. those variables that are conditionally incremented (or decremented) by a positive (or negative) value on each iteration. We have implemented a conservative version of this, akin to abstract interpretation, that handles addition or subtraction of known values in the SCR and assigns lattice values (monotonically increasing, strictly monotonically increasing, etc.) to the operators in the SCR. Dependence solvers may be able to take advantage of this information; as the need arises, our implementation will be strengthened.

Some support for the recognition of *non-constant* periodic induction variables has also been added. *Constant* periodic induction variables cycle through a set of values, e.g. $\{1, 2, 3, 1, 2, 3, \ldots\}$. *Linear* periodics, however, use that set of values as increments from

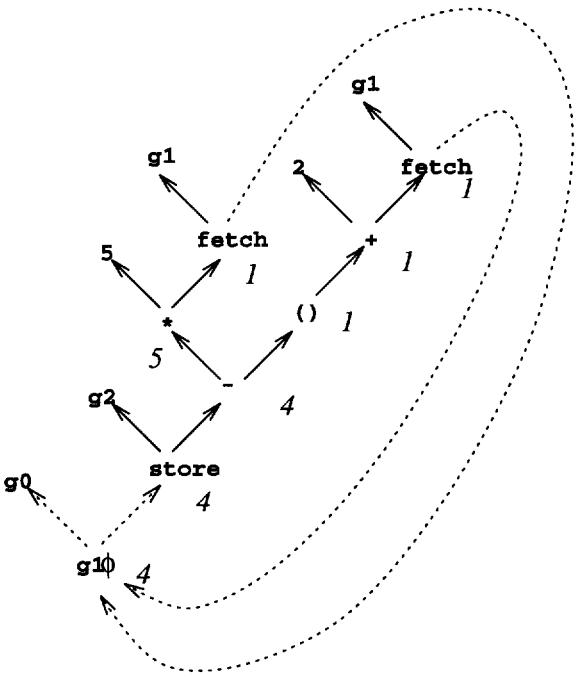


Figure 6: SSA graph with factor values

the previous value: using the set $\{1,3,5\}$, a variable with period 3, starting at 0, would take on the values $\{1,4,9,10,13,18,\ldots\}$. We use the term "linear" here since we only support the addition or subtraction of loop invariant values[3].

4 Experimental Results

In this section we present some preliminary data that provides insight into the nature of the data dependence problem in general. Data dependence tests vary greatly in efficiency and accuracy. For example, the GCD test considers only the coefficients of the loop indexes: if their GCD divides the constant term, there is an integer solution to the dependence equation and a dependence may exist (depending on the loop bounds). If the dependence equation contains unknown variables, this test may not be used. Other tests have other constraints. It is important to examine, then, the types of expressions that occur in the

[3]It is interesting to note that our implementation now distinguishes constant periodics as linear periodics: the sequence $\{1,2,3,1,2,3,\ldots\}$ is a linear periodic using the set $\{1,1,-2\}$.

program	*description*	*lines*	*routines*	*loops*	*subscripts*
ADM	fluid dynamics	6108	97	306	3104
ARC2D	fluid dynamics	3967	39	234	3858
EISPACK	eigensystem package	11466	70	673	4820
LINPACK	linear system package	10058	60	332	3242
OCEAN	fluid dynamics	4346	36	144	438
QCD2	chemical & physical model	2330	35	168	941
SPEC77	fluid dynamics	3888	65	413	2604
SPICE	circuit simulation	18524	128	666	5842
TRFD	chemical and physical model	488	7	79	137
total		61175	537	3015	24938

Table 1: Codes used

subscript expressions that produce the dependence equations [WB87]. Our algorithm for classifying induction variables was run on several common Fortran programs, described in table 1. EISPACK and LINPACK are common mathematical libraries, and the others are from the Perfect Club benchmark suite [CKPK90].

Each subscript expression that appeared within a loop was recorded by type and by the actual expression. Four types of induction expressions are distinguished: *invariant*, *linear*, *variant*, and *other*. Invariant expressions are constants such as integer values, subroutine parameters, loop-invariant expressions, or some combination of these. Linear expressions are defined in terms of linear functions of the indexes of the enclosing loops. (A linear expression with all coefficients equal to zero is considered invariant.) Variant expressions are those values for which the compiler cannot determine a closed form. "Other" expressions represent polynomial and geometric IVs, monotonic IVs, and so on. The type of expression is based on the occurrence of the array reference containing the expression, while the expression takes into account all information. In the following fragment, for example,

```
L6:   do i = 1, n
L7:      do j = 1, i
            A(i) = ...
         enddo
      enddo
```

we consider the subscript expression to be *invariant* because it occurs in the inner loop, even though the induction expression is actually $h_6 + 1$.

Figure 7 shows the relative frequencies of the four types of subscript expressions for the sample programs. While some programs vary considerably (SPICE, ARC2D), the average over all subscripts in all programs is 40% linear, 33% invariant, 26% variant, and 1% other. Figure 8 presents the same data (for all programs collectively), for each dimension of the array reference. As there are significantly more arrays of one dimension (69%) than two or three dimensions (24% and 7%), the first dimension data reflects the average. At the point of occurrence, it is not surprising that in the inner dimensions the number of invariant expressions decreases, presenting more possible parallelism.

Fully one-third of the subscript expressions in programs are invariant at the point of usage. Figure 9 presents a breakdown of these expressions by form (c represents an integer constant, v represents any unknown variable). This data is presented for 85% of the most frequently occurring types of expressions, but the trends shown hold for all expressions. Ignoring the forms containing variant terms, two-thirds (67%) of the invariant expressions are known constants or single-variable linear forms. Only 18% have unknown symbolic quantities.

Of the subscripts with linear expressions, figure 9 shows the 85% most frequent forms encountered (k represents a coefficient not equal to 1). Again ignoring variant expressions, half the subscript expressions are of the simple form $c \pm 1h$. Less than 10% have unknown variables. We have also found that of those linear coefficients not equal to ± 1, the majority are small integers, making tests for integer solutions easier in some dependence tests [SLY90].

Figure 7: Subscript classifications

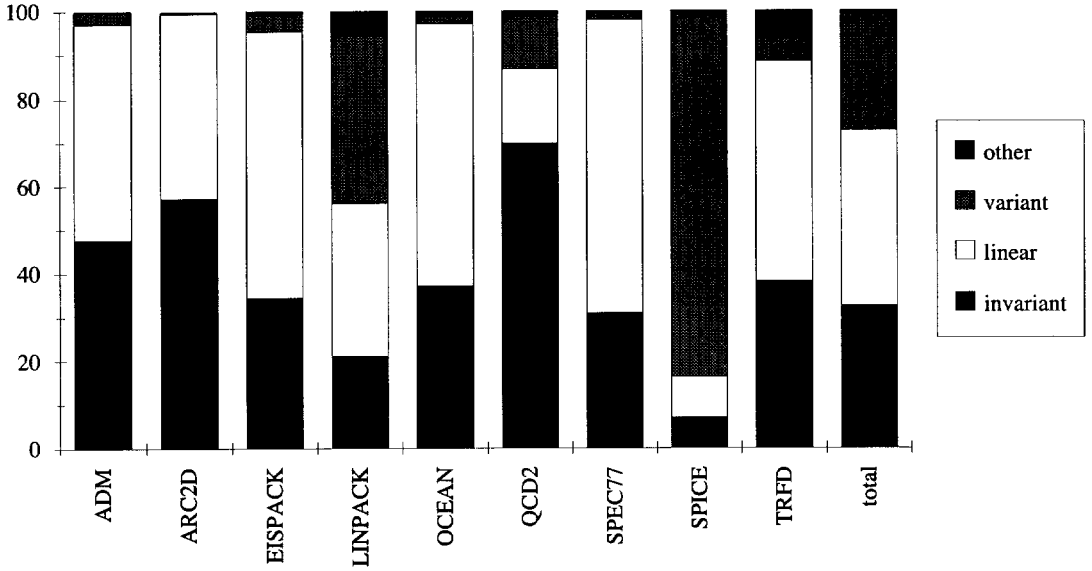
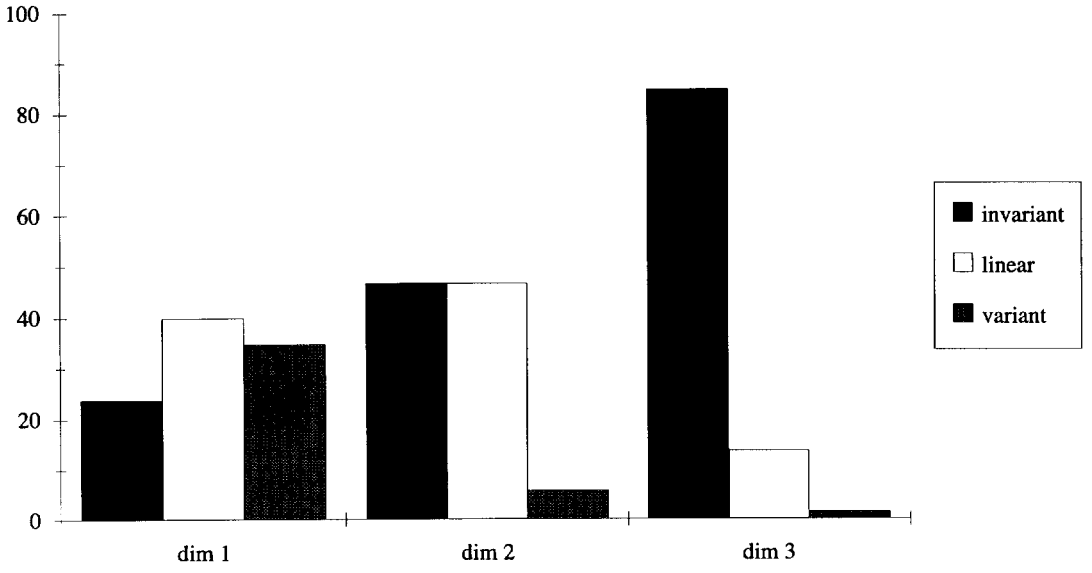


Figure 8: Subscript classifications by dimension



form	*invariant %*	*linear %*
c	39	*n/a*
$c \pm 1h$	28	49
$c \pm kh$	0	5
$c \pm v \pm 1h$	9	19
$c \pm v$	8	*n/a*
$c \pm h_i \pm h_j$	0	4
$c \pm v \pm kh_i \pm kh_j$	0	4
$c \pm v \pm h_i \pm h_j$	0	3
other	1	3
contains variant term	15	14

Figure 9: Invariant forms

One quarter of the subscripts have expressions which cannot be classified. We have identified two causes. Four percent of all the variant expressions result from the subscript expression containing a variable passed to a subroutine. Interprocedural analysis may be able to determine if these variables are not modified within the called routine. More significantly, we have found that many codes contain indexed array references, such as `A(B(i))`. In SPICE, which has the most variant expressions, 60% are caused by indexed references. Other codes average approximately 20%. Without user assertions, a compiler cannot eliminate these sources of variance. The remainder of the variant subscripts (three-quarters) have not been specifically identified by cause. In general, nodes in the SSA graph are classified as variant due to fetches of non-integer values and nontrivial SCRs not matching one of the induction variable forms.

Other researchers have recently examined various aspects of subscript expressions [SLY90, Hag90], with similar results. These results represent a first approximation to help determine what types of data dependence tests should be applied. Further study is needed, particularly in analyzing pairwise comparisons of references to the same array at all levels in each loop nest.

5 Related Work

Induction variable detection for strength reduction, particularly of array expressions, is well covered in the literature. The usual approach is to use reaching definition information within a loop body and search for assignments of the form `i := i` $\pm$ `c`, where `c` is loop invariant. This defines `i` as a *basic induction variable* . Other assignments of the form `j := c*i + k`, where `c` and `k` are loop invariant (possibly 0), associate with `j` the tuple $(\mathtt{i}, \mathtt{c}, \mathtt{k})$, putting `j` in the *family* of `i` [ASU86].

Mutually defined induction variables cannot be found by this algorithm, however, since the other variable on the right-hand is not known to be in any family of induction variables. Kennedy et al [ACK81] present a comprehensive treatment of strength reduction by recognizing more general linear cases. In the PTRAN compiler [ABC+88], such cases are solved by a dataflow technique which initially assumes all variables are linear induction variables until a contradiction exists.

Abstract interpretation has been used to recognize recurrence relations. In [AI90], a method is presented in which abstract interpretation is used to associate each variable assigned within a loop a symbolic expression; these symbolic expressions are then compared against known patterns representing recurrence relations.

The Parafrase-2 compiler uses a symbolic interpreter for its optimizations; for the recognition of induction variables, a scheme has been proposed whereby the compiler will solve a system of recurrence relations that describe the behavior of variables within loops [HP92]. In some respects, the approach used in Parafrase-2 is similar to ours: the symbolic interpretation approach determines a symbolic expression for each variable at each node of the program's flow graph by interpreting the expressions within that node, based on symbolic expressions of predecessor nodes. The key difference is that the Parafrase approach translates all induction variables into recurrence relations to be solved. With our technique, linear induction variables are classified directly. Our treatment of polynomial and geometric IVs is closer to Parafrase's.

A chart is presented in [HP92] listing 10 forms of induction expressions and sym-

bolic substitutions. Four compilers are compared using a test-suite containing these forms: Parafrase-2 recognizes all 10, the Titan compiler recognizes 2, and both the KAP compiler and the VAST-2 compiler recognize none. We are pleased to report Nascent scores 6 as of this writing. The remaining four are within the scope of our approach and are currently being incorporated; we anticipate no problems in this regard.

In [EHLP92], Eigenmann et al discuss their experiences in hand-parallelizing four of the Perfect benchmarks. They note the importance of having compilers detect *generalized induction variables* (GIVs) (referred to in this work as polynomial and geometric IVs). A factor of 8 speedup was obtained in one case by replacing a geometric induction variable with its closed form in terms of the loop index.

6 Conclusions

This paper has discussed the implementation of an SSA-based algorithm for detecting several forms of induction variables. In particular, we have presented some details needed for implementation and outlined experimental results showing the effectiveness of our technique.

We have added η-functions to the standard SSA form in our compiler to provide gates for exit value expressions. The overall cost of η-functions is comparatively small in both time and space, and no new phases have been added to the compiler for postexit node insertion in the CFG or η node insertion in the SSA graph. There is some cost in creating exit value expression trees, but this is dominated by the cost of the induction variable procedure as a whole. At present we attempt to build an exit value for *every* variable assigned within a loop, increasing the size of our intermediate form by about 10%. This approach is overly conservative: we do not need η expressions in all cases, but as a consequence of our flow graph algorithms, the value at each η is considered used. The side-effect of providing a convenient method to propagate exit values may prove useful, although this has not yet been measured; in the worst case the η-function's value expression can be treated as dead code.

To provide the most precise exit values, the compiler must be able to perform forward substitution of symbolic values and lower bounds. Local constant propagation and forward substitution is a natural extension of this work; the support for symbolic algebra we are implementing in our compiler for improved induction expression representation will contribute here.

We have improved upon the treatment of geometric forms and periodic forms. In the former case, the original definition was clearly inadequate. In the latter case, we wished to show our technique was as powerful as others. While such cases are not at all common, there are cases where their recognition has been an important step in parallelizing scientific programs.

Finally, we have begun to measure the characteristics of subscript expressions in scientific Fortran codes, and have found that the number of distinct forms may suggest the data dependence problem may not be as difficult as had been feared. This data may eventually provide insight into the directions for future research in dependence tests. Further analysis of the causes of *variant* expressions in general will result in better classification of subscript expressions.

SSA-based induction variable analysis has several advantages over previous methods. It is clearly a more general solution than traditional pattern matching and we have shown in our implementation that it can recognize the same types of expressions as other proposed schemes. Perhaps most importantly, this technique can be readily incorporated into existing compilers that use internal representations similar to SSA. For strength reduction, all linear induction expressions can be detected very quickly with only one pass over the SSA graph. With as much support added for symbolic forms as desired, compilers that require dependence information for subscripts will be able to detect linear dependences as well as more complex forms.

7 Acknowledgements

I am grateful to both my advisor, Michael Wolfe, and my fellow student, Eric Stoltz. Both provided much input into the development of the ideas in this paper.

This paper was submitted on 7 May 93 and presented on 21 May 93 in fufillment of the Research Proficiency requirement for advancement to candidacy at OGI-CSE. A more complete treatment of this work in is progress.

References

[ABC+88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, pages 617–640, May 1988.

[ACK81] F. E. Allen, John Cocke, and Ken Kennedy. Reduction in operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, pages 79–101. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[AI90] Zahira Ammarguellat and W. L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–295, June 1990.

[ASU86] Alfred V. Aho, Ravi Sethi, , and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BMO90] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, October 1991.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the perfect benchmarks. In *Proceedings of the International Conference on Supercomputing*, pages 254–266, March 1990.

[EHLP92] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 65–83. Spinger-Verlag, 1992. LNCS no. 589.

[Hag90] Mohammad Reza Haghighat. Symbolic dependence analysis for high performance parallelizing compilers. Technical Report 995, University of Illinois (CSRD), May 1990. M.S. Thesis.

[HP92] Mohammed R. Haghighat and Constantine D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Workshop on Languages and Compilers for Parallelism*, pages 355–369, 1992.

[SLY90] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.

[WB87] Michael Wolfe and Utpal Banerjee. Data dependence and its applications to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, April 1987.

[Wol92] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–174, June 1992.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.