Beyond Induction Variables

Michael Wolfe [*]
Oregon Graduate Institute of Science and Technology

Abstract

Induction variable detection is usually closely tied to the strength reduction optimization. This paper studies induction variable analysis from a different perspective, that of finding induction variables for data dependence analysis. While classical induction variable analysis techniques have been used successfully up to now, we have found a simple algorithm based on the the Static Single Assignment form of a program that finds all linear induction variables in a loop. Moreover, this algorithm is easily extended to find induction variables in multiple nested loops, to find nonlinear induction variables, and to classify other integer scalar assignments in loops, such as monotonic, periodic and wrap-around variables. Some of these other variables are now classified using ad hoc pattern recognition, while others are not analyzed by current compilers. Giving a unified approach improves the speed of compilers and allows a more general classification scheme. We also show how to use these variables in data dependence testing.

1 Introduction

In classical compiler analysis, induction variable recognition is inextricably linked to the *strength reduction* transformation; in fact, the two terms are sometimes used interchangeably. The most common candidates for strength reduction (and therefore the most important induction variable candidates) are array address calculations in inner loops.

[*]Supported in part by NSF Grant CCR-8906909 and by DARPA Grant MDA972-88-J-1004.

Nowadays, many compilers now include advanced loop transformations (such as *loop distribution* and *loop interchanging*). These transformations have been proven to be useful (and are commercially used) for a wide variety of systems, ranging from high performance vector multiprocessor supercomputers down to uniprocessor workstations, where the transformations improve the cache utilization. These transformations require analysis of array subscripts to determine the data dependence relations in loops. Current methods to test and characterize data dependence relations for subscripted array references require the subscript expressions to be a linear combination of induction variables in the enclosing loops.

The long history of research into data dependence testing has found that many variables used in subscript expressions are not induction variables, but can be characterized as one of several special cases. Current compilers classify these variables using ad hoc pattern recognition algorithms. Many compilers do not include these recognition algorithms at all, ignoring potential optimization opportunities.

This paper gives a simple algorithm for finding linear induction expressions in nested loops. The strength of the algorithm is that it is fast and that it will also characterize other scalar integer variables in the loops, such as

- wrap-around variables,
- flip-flop variables,
- families of periodic variables,
- non-linear induction variables (polynomial and geometric), and
- monotonically increasing or decreasing variables.

Finally, we look at how to apply dependence tests for these new cases. Our algorithm is based on the Static Single Assignment form of the program.

2 Induction Variables and Static Single Assignment

An *induction variable* is typically defined in compiler texts as a variable whose value is systematically incremented or decremented by a constant value in a loop [ASU86, FL88]. This is sometimes broken into *basic induction variables*, *i.e.* variables which appear only in statements of the form:

```
        i = i0
L1:     loop
           ⋮
           i = i + k
           ⋮
        endloop
```

where k is a constant or loop invariant, and other induction variables, which are linear combinations of known induction variables. Many programming languages include a loop construct with an index variable that is defined to be an induction variable. One of the early improvements in induction variable analysis was to extend the definition of basic induction variables to the maximal set of variables which are assigned by simple expressions of the form:

```
i = j ± k
```

where j and k are invariants or induction variables (note j may be zero) [CK77, ACK81]. This allows mutually-defined induction variables, such as:

```
        j = n
L2:     loop
           ⋮
           i = j + c
           ⋮
           j = i + k
           ⋮
        endloop
```

where c and k are invariants in the loop.

With each induction variable a compiler associates its initial value and its increment, or step. We represent this by a tuple $\langle l, i, s\rangle$, where l is the name of the loop, i is the initial value (represented symbolically if it cannot be determined) and s is the step. The examples here will number the loops; thus in the first example, i is a basic induction variable which can be described as $\langle L1, i0 + k, k\rangle$.

In nested loops, an induction variable in the inner loop may have an initial value or step that varies in the outer loop; they may even be induction variables in the outer loop:

```
        i = 0
L3:     loop
           i = i + 1
           j = i
L4:        loop
              j = j + i
           endloop
        endloop
```

Here, i is an induction variable $\langle L3, 1, 1\rangle$, while j is an induction variable $\langle L4, i+i, i\rangle$; even though the step of j varies in the outer loop, it is still a linear induction variable in the inner loop.

We define a *multiloop induction variable* as an induction variable in an inner loop with a step that is invariant in the outer loop, and an initial value that is an induction variable in the outer loop:

```
        i = 0
L5:     loop
           i = i + 2
           j = i
L6:        loop
              j = j + 1
           endloop
        endloop
```

Here, i=$\langle L5, 2, 2\rangle$, while j=$\langle L6, i+1, 1\rangle$. We will represent such multiloop induction variables with nested tuples, so that j=$\langle L6, \langle L5, 3, 2\rangle, 1\rangle$.

2.1 The Static Single Assignment Form

Since the algorithm presented here is based on the Static Single Assignment (SSA) form, we briefly cover the main ideas behind SSA and show a quick example; readers familiar with Static Single Assignment can skip to the next section. The SSA form of a program can be considered as a sparse representation of *use-def chains*. The algorithms to convert a sequential program into SSA form are based upon the *Control Flow Graph* (CFG), which is a graph $G = \langle V, E, \mathit{Entry}, \mathit{Exit}\rangle$, where V is a set of nodes representing basic blocks in the program, E is a set of edges representing sequential control flow in the program and *Entry* and *Exit* are distinguished nodes representing the unique entry point into the program and the unique exit from the program respectively.

After a program has been converted into its SSA form, it has two key properties:

1. Every use of any variable in the program has exactly one *reaching definition*, and
2. at confluence points in the CFG, merge functions called ϕ-functions are introduced. A ϕ-function for a variable merges the values of the variable from distinct incoming control flow paths, and it has one argument for each control flow predecessor.

(a)

```
      j = n
L7:   loop


        i = j + c
        j = i + k
      endloop
```

(b)

$$\begin{aligned}
&\mathtt{j}_1 = \mathtt{n}_1 \\
\mathtt{L7'}\!:\ &\mathtt{loop} \\
&\quad \mathtt{i}_2 = \phi(\mathtt{i}_1, \mathtt{i}_3) \\
&\quad \mathtt{j}_2 = \phi(\mathtt{j}_1, \mathtt{j}_3) \\
&\quad \mathtt{i}_3 = \mathtt{j}_2 + \mathtt{c}_1 \\
&\quad \mathtt{j}_3 = \mathtt{i}_3 + \mathtt{k}_1 \\
&\mathtt{endloop}
\end{aligned}$$

$$\begin{aligned}
\mathtt{j}_2 &= \langle L7', \mathtt{n}_1, \mathtt{c}_1 + \mathtt{k}_1 \rangle \\
\mathtt{i}_3 &= \langle L7', \mathtt{n}_1 + \mathtt{c}_1, \mathtt{c}_1 + \mathtt{k}_1 \rangle \\
\mathtt{j}_3 &= \langle L7', \mathtt{n}_1 + \mathtt{c}_1 + \mathtt{k}_1, \mathtt{c}_1 + \mathtt{k}_1 \rangle
\end{aligned}$$

Figure 1: Example loop with SSA form.

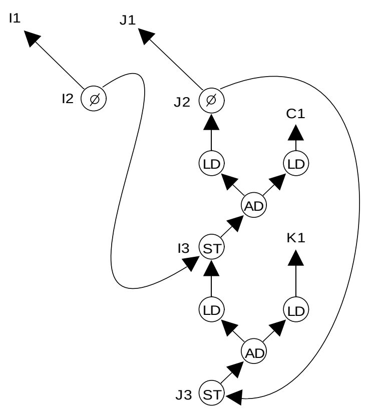
The two key algorithms used to convert a program into SSA form are the *placement* of ϕ-functions to satisfy the second property, and the *renaming* of variables to satisfy the first property. Intuitively, a ϕ-function for variable `X` is placed at the first CFG vertex v where two distinct definitions of `X` reach; the ϕ-function itself counts as a new definition of `X`, and so the algorithm iterates. Once ϕ-functions are inserted, each variable generation is renamed to a unique name (here represented by subscripted names), and each use of a variable is replaced by the unique reaching SSA definition; ϕ-function arguments are renamed to the SSA name that reaches the corresponding control flow edge. The reader is encouraged to read the SSA paper [CFR+91] for more details on these algorithms, and [AWZ88, RWZ88, WZ91] for details on applications of the SSA form. An example of a loop with its SSA form is given in Figure 1; the SSA names that correspond to induction variables are shown.

3 The SSA Graph

As with previous related work, we assume the program is represented by a CFG, as explained earlier. Moreover, we assume that each basic block is represented by a linked list of tuples of the form

$$\langle \mathit{op}, \mathit{left}, \mathit{right}, \mathit{ssalink} \rangle$$

where *op* is the operation code, *left* and *right* are the two operands, and *ssalink* is used only for loads (and indexed stores, which are not discussed here) to indicate the single reaching SSA name for this variable. The operators available include those given in the table in Figure 2. Load and store operations have additional arguments describing the variable for which they apply.



AD	addition
SB	subtraction
MP	multiplication
DV	division
EX	exponentiation
NG	negate
PH	ϕ-function
LD	load
ST	store
LT	literal

Figure 2: SSA graph for example program.

Indexed loads and stores are denoted by the presence of subscripts; since we are concerned only with scalars here, we will not elaborate.

The algorithm to analyze the program uses a graphical abstraction called the SSA graph. When analyzing a loop, the vertices in SSA graph are the tuples representing operations within that loop. The edges go from each tuple to the *left* and *right* operands as well as (for loads) to the *ssalink*. The SSA graph for the loop in Figure 1 is given in Figure 2. Note that the edges go from the operators to the *source* operands. Edges from the `LD` vertices correspond to the *ssalink* field.

3.1 Basic Linear Induction Variables

Our algorithm to find the induction variables is based on Tarjan's well-known algorithm to find strongly connected regions (SCRs) in directed graphs [Tar72]. The primary observation is that each basic linear induction variable will belong to a nontrivial SCR in the SSA graph. Each such SCR in the SSA graph must include a loop-header ϕ-function, since all values cycling around the loop must pass through a ϕ-function. The advantage to using Tarjan's algorithm is that when it identifies an SCR in the graph, it will have visited all the successors of the SCR; because of the way the edges are directed in our graph, when an SCR is identified, all the source operands reaching the SCR will already have been visited and identified. Our modifications to Tarjan's algorithm are to classify each SCR as an induction variable (or something else) at the time the SCR is identified; this information will then be immediately available for other SCRs which use the value of any operator in this SCR.

A simple method to identify an SCR in the SSA graph as a family of basic linear induction variables is to require that it satisfy the following simple constraints:

- The SCR contains only one ϕ-function (at the header of the loop).
- The SCR contains only addition and subtraction operators, and the right operand of the subtraction is not part of the SCR (no `i=n-i` assignments).
- The other operand of each addition or subtraction is loop invariant.
- Any loads and stores are to unsubscripted variables (*i.e.* load/store addresses are loop invariant).

There is one strongly connected region in the graph in Figure 2 which satisfies all these constraints, thus we know that it defines a family of basic linear induction variables. The *step* for all variables in the family is computed as the cumulative effect of all the increment and decrements in the SCR. The first member of the family is the left hand side of the ϕ-function; the initial value of this linear induction variable is the "other" argument to the ϕ-function, the reaching SSA name from outside the loop. Thus we can describe the linear induction variable j_2 as $\langle L7', j_1, c_1 + k_1 \rangle$. The other members of the family differ only by a constant in the initial value. Often the initial value coming in from outside the loop can be evaluated and substituted, using an algorithm such as constant propagation [WZ91].

Actually, the rules for basic linear induction variables can be relaxed a bit. As long as the variable is incremented by the same amount on each path through the loop, the compiler can classify it as an induction variable. An example is shown in Figure 3. The only operators in the strongly connected region in the SSA graph are addition and ϕ-functions and the only values added are loop invariants. Starting with the loop-header ϕ-function, the compiler can determine the offset of each SSA variable from the value of the first ϕ-function for that iteration. As long as each ϕ-function within the loop (e.g., at `endif` statements) has the same offset for all arguments, the SCR defines a family of basic linear induction variables.

```
         (a)                       (b)

         i = 1                     i₁ = 1
   L8:   loop               L8':   loop
                                      i₂ = φ(i₁,i₅)
            if exp then               if exp then
               i = i+2                   i₃ = i₂+2
            else                      else
               i = i+2                   i₄ = i₂+2
            endif                     endif
                                      i₅ = φ(i₃,i₄)
         endloop                   endloop
```

$$\begin{aligned} i_2 &= \langle L8, 1, 2 \rangle \\ i_3 &= \langle L8, 3, 2 \rangle \\ i_4 &= \langle L8, 3, 2 \rangle \\ i_5 &= \langle L8, 3, 2 \rangle \end{aligned}$$

Figure 3: Example loop with SSA form.

4 Beyond Induction Variables

4.1 Wrap-Around Variables

After examining many user programs to find how to improve their compilers, vendors and researchers found some particular patterns that appeared often enough to warrant special recognition and handling in a compiler. One of these is called a *wrap-around variable* [PW86], such as `im1` below:

```
        im1 = n
  L9:   for i = 1 to n loop
           A(i) = A(im1) + ...
           im1 = i
        endloop
```

In all iterations but the first, the value of `im1` is equal to `i-1`; in the first iteration, however, `im1` has some other value, usually the index of the last element of the array `A`. Typically, `im1` is used to wrap the array `A` around a cylinder, hence the name. The standard compiler trick, once a wrap-around variable is found, is to peel off the first iteration of the loop and replace the wrap-around variable with the appropriate induction variable. Typically, wrap-around variables are found with a separate pattern matching analysis of the loops, following induction variable analysis.

In the SSA graph, a wrap-around variable is possible when a loop-header ϕ-function appears in a strongly connected region by itself. If the SSA name carried around the loop is an induction variable of some type, then the loop-header value is a first-order wrap-around value; that is, in all but the first iteration, its value

```
      i = 1              $i_1$ = 1
 L10: loop          L10':loop
                         $k_2$ = $\phi(k_1,k_3)$
                         $j_2$ = $\phi(j_1,j_3)$
                         $i_2$ = $\phi(i_1,i_3)$
        k = j            $k_3$ = $j_2$
        j = i            $j_3$ = $i_2$
        i = i + 1        $i_3$ = $i_2$ + 1
      endloop          endloop
```

Figure 4: Wrap-around variable in SSA form.

will be an induction variable in the loop. This is the case with j_2 in Figure 4; as we will see in the next section, this will then define k_3 as a first-order wrap around variable. Wrap-around variables can be cascaded; any loop-header ϕ-function with one argument corresponding to a n-th order wrap-around variable is itself an $(n+1)$-th order wrap-around variable. Thus, k_2 is a second-order wrap-around variable. In general, any of the other known classes of variables described below could also be "wrapped around" the loop, meaning that the variable will have that property only after some fixed number of iterations around the loop. A wrap-around variable would then be identified by its order and the class of the variable that is wrapped around the loop.

If the initial value for the wrap-around variable fits the induction sequence, it may be more precisely identified as an induction variable. For instance, if the initial value of j_1 in loop L10 above had been 0, then j_2 could have been identified as the induction variable $\langle L10, 0, 1\rangle$.

4.2 Flip-Flop and Periodic Variables

A method used in some relaxation codes is the generation of "new" values for a matrix from the "old" values; one common way to program this is to add a dimension to the matrix with size 2, then to use `A(1,*,*)` as the "old" values and generate the new values into `A(2,*,*)`. For the next iteration of the relaxation, the program flips the definition of "old" and "new". Two programming constructs are typically used to implement this. The first switches the values of the two variables:

```
      j = 1
      jold = 2
 L11: for iter = 1 to n loop
         ··· relaxation code ···
         jtemp = jold
         jold = j
         j = jtemp
      endloop
```

```
 L13: loop          L13':loop
                         $t_2$ = $\phi(t_1,t_3)$
                         $j_2$ = $\phi(j_1,j_3)$
                         $k_2$ = $\phi(k_1,k_3)$
                         $l_2$ = $\phi(l_1,l_3)$
         t = j           $t_3$ = $j_2$
         j = k           $j_3$ = $k_2$
         k = l           $k_3$ = $l_2$
         l = t           $l_3$ = $t_3$
      endloop         endloop
```

Figure 5: Example periodic variable in SSA form.

Another method takes advantage of the equality `jold=3-j` to compute the new values of `jold` and `j`:

```
      j = 1
      jold = 2
 L12: for iter = 1 to n loop
         ··· relaxation code ···
         j = 3-j
         jold = 3-jold
      endloop
```

Methods have been proposed to recognize the latter form of flip-flop variable, but are not typically implemented in commercial products. In either case, it is extremely important and useful for the compiler to realize that for any fixed value of `iter`, `j` and `jold` have different values, though it may not know what those values are.

A generalization of flip-flop variables is a tuple of variables whose values are rotated around the tuple, as around a ring; see Figure 5. Here, we see the tuple ⟨`j`,`k`,`l`⟩ which comprises a set of *periodic variables* with period 3. Again, it is useful to know that for any given iteration, no two of these variables will have the same value (assuming the initial values of these variables are distinct). Flip-flop variables as shown in loop L11 are then immediately recognized as periodic variables with period 2.

In the SSA graph, when a strongly connected region contains more than one loop-header ϕ-function and has no arithmetic and no other ϕ-functions, then it corresponds to a family of periodic variables. The *period* is equal to the number of loop-header ϕ-functions in the SCR. In the example in Figure 5, this number is three; note that t_2 does not appear in the strongly connected region with the other variables. It is important to note that the compiler will need to determine that the initial values for the periodic variables are distinct in order to take advantage of the periodicity for data dependence testing; these initial values are easily found from the "other" arguments to the loop-header ϕ-functions in the SCR.

4.3 Non-Linear Induction Variables

The classical definition of an induction variable is that it follows a linear sequence of values. With little additional effort, we can also find polynomial and geometric series. In the program below the various induction variables are given in the table on the right:

```
          j = 1
          k = 1
          l = 1
     L14: for i = 1 to n loop
             j = j + i
             k = k + j + 1
             l = l * 2 + 1
          endloop
```

variable	sequence	closed form
i	$1, 2, 3, 4, \ldots$	$i = \langle L14, 1, 1\rangle$
j	$2, 4, 7, 11, \ldots$	$(h^2 + 3h + 4)/2$
k	$4, 9, 17, 29, \ldots$	$(h^3 + 6h^2 + 23h + 24)/6$
l	$3, 7, 15, 31, \ldots$	$2^{h+2} - 1$

where h is a basic loop counter that starts at zero ($h = \langle L14, 0, 1\rangle$). Incrementing a variable by a polynomial induction variable produces an induction variable of the next higher order. In general, finding the closed form for polynomial and geometric induction variables is simplified since (a) the nonzero coefficients can always be identified by the compiler, and (b) the coefficients will always be rational. We use simple matrix inversion with rational arithmetic to find the coefficients, as shown below. Finding polynomial induction variables for strength reduction may not be so interesting, since the operation is already only addition. From the point of view of data dependence, there are currently few dependence testing algorithms that can take advantage of this additional knowledge. We will revisit this point briefly in a later section. It is interesting to note that the second form of flip-flop variable described above can be recognized as a geometric induction variable with base -1; in loop L12 at iteration i, the value assigned to variable j is $-1^i + 3 \times \sum_{k=0}^{i-1} -1^k$.

In the SSA graph, when Tarjan's algorithm has found a strongly connected region that fails one of the conditions of a linear induction variable, the SCR can be analyzed to see if it is an identifiable non-linear induction variable. In all cases analyzed here, the SCR must include a single ϕ-function at the loop header. The key is to classify the cumulative effect of all the operations in the SCR on the value reaching the loop-header ϕ-function relative to the value of that ϕ-function in the previous iteration.

- If the effect is to increment the loop-header value by a linear induction variable, the SCR is a family of polynomial induction variables of second order.
- If the effect is to increment the value by a polynomial induction variable of order n, this SCR is a family of polynomial induction variables of order $n + 1$.
- If the cumulative effect is to multiply the loop-header value by a known integer, the SCR corresponds to a family of geometric induction variables.
- If the cumulative effect is to subtract the loop-header value from a loop invariant, the SCR corresponds to a flip-flop variable, equivalent to a periodic variable of period two.

This could be taken to extreme, such as recognizing that multiplying by a linear induction variable generates a factorial sequence. A polynomial induction variable can be represented in the compiler by extending the linear induction variable tuple to include the coefficients of each power of the basic loop counter; thus we write $\langle l, i, s_1, s_2, \ldots, s_m\rangle$ represents an induction variable for loop l whose value on the first iteration is i, and whose value on iteration h is $i + \sum_{k=1}^{m} s_k h^k$, or letting $s_0 = i$, $\sum_{k=0}^{m} s_k h^k$. A compiler can find the coefficients s_k by matrix inversion. The number of unknowns (the values of s_k) is equal to $m+1$, where m is the order of the polynomial; since m can be determined by the compiler, the number of unknowns is fixed. By inverting the matrix of elements

$$a_{ij} = i^j,\ 0 \le i \le m,\ 0 \le j \le m$$

and multiplying the inverse by the computed (perhaps symbolic) first k values of the induction variable, the coefficients can be found. Since the entries of the matrix a_{ij} are all integer, the inverse will have only rational entries. For instance, in loop L14 above, since **k** is a third-order polynomial induction variable, the corresponding matrix is:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{bmatrix}$$

The inverse of this matrix is:

$$A^{-1} = \frac{1}{6}\begin{bmatrix} 6 & 0 & 0 & 0 \\ -11 & 18 & -9 & 2 \\ 6 & -15 & 12 & -3 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

which, when multiplied by the computed first four values of **k** gives the coefficients of the closed form of the induction variable:

$$A^{-1} \begin{bmatrix} 4 & 9 & 17 & 29 \end{bmatrix}^T = \begin{bmatrix} 4 & \frac{23}{6} & 1 & \frac{1}{6} \end{bmatrix}^T$$

A geometric induction variable has both geometric and polynomial terms; we represent this by the polynomial coefficients followed by the coefficients of each exponential term. We write

$$\langle\langle l, s_0, s_1, \ldots, s_m\rangle, g_p, g_{p+1}, \ldots, g_q\rangle$$

to represent the induction variable for loop l with initial value s_0, and whose value on iteration h is

$$\sum_{k=0}^{m} s_k h^k + \sum_{k=p}^{q} g_k k^h$$

A matrix inversion technique to find the coefficients also applies. A basic geometric induction variable will have only a single exponential term (though it may have any number of polynomial terms); if the geometric base is g, the matrix to be inverted is:

$$a_{ij} = \begin{cases} i^j, & 0 \le i \le m+1,\ 0 \le j \le m \\ g^i, & 0 \le i \le m+1,\ j = m+1 \end{cases}$$

Suppose we had an assignment in loop L14 above of the form `m=3*m+2*i+1` (with initial value `m=0`); the geometric base of this induction variable is 3, while the polynomial part would seem to have a quadratic term (since a linear induction variable is being added). Thus, the matrix to be inverted is:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 3 \\ 1 & 2 & 4 & 9 \\ 1 & 3 & 9 & 27 \end{bmatrix}$$

The inverted matrix, multiplied by the first four computed values, is:

$$\frac{1}{8}\begin{bmatrix} 9 & -3 & 3 & -1 \\ -12 & 16 & -4 & 0 \\ 6 & -14 & 10 & -2 \\ -1 & 3 & -3 & 1 \end{bmatrix}\begin{bmatrix} 3 \\ 14 \\ 49 \\ 156 \end{bmatrix} = \begin{bmatrix} -3 \\ -1 \\ 0 \\ 6 \end{bmatrix}$$

Thus, `m` can be described by $m = 6 \times 3^h - h - 3$ (note there is no quadratic term after all).

4.4 Monotonic Variables

Variables that are conditionally incremented or decremented can not generally be recognized as induction variables. However, they often do exhibit useful properties. A common pattern is to increment a variable conditionally, for instance, to pack certain values from one vector to another:

```
     k = 0
L15: for i = 1 to n loop
        if A(i) > 0 then
           k = k + 1
           B(k) = A(i)
        endif
     endloop
```

```
L16: loop                L16':loop
                              k2 = φ(k1,k5)
        if exp then           if exp then
           k = k+1               k3 = k2+1
        else                  else
           k = k+2               k4 = k2+2
        endif                 endif
     endloop                  k5 = φ(k3,k4)
                           endloop
```

Figure 6: Example monotonic variable in SSA form.

Some compilers recognize this whole pattern as a pack operation, as might be converted into the Fortran-90 PACK intrinsic function. Our approach is to realize that within the range of the conditional, `k` will never have the same value for two different iterations of the loop. Moreover, even outside the range of the conditional, the compiler can determine that the value of `k` will never decrease for subsequent iterations; we say that `k` is *monotonically increasing.*

Sometimes our approach can discover stronger relations. If the variable is incremented for each iteration, even though it is not always incremented by the same value, we can say that the variable is *monotonically strictly increasing*, as in Figure 6. Even though we have no way to determine a closed form for `k` here, we will see that this information can be used effectively in dependence analysis.

In the SSA graph, a monotonic variable appears as a strongly connected region that contains ϕ-functions other than at the loop-header, perhaps at an `endif`, as in Figure 6. Since the arguments to the ϕ-function at the `endif` assigning k_5 have have different offsets from the loop-entry value k_2, namely $k_2 + 1$ and $k_2 + 2$, the SCR does not correspond to a family of induction variables. Nonetheless, it is easy to realize that regardless of the path through the loop, the value of `k` is always monotonically increasing. Regardless of the operations in the SCR, if the compiler can determine that the values being merged at a ϕ-function are always larger than the loop-header value, it can classify all variables in the SCR as monotonic. Multiply operations can also be allowed, such as `2*i+1` as long as the initial value of `i` is known. If the value of a monotonic variable is *always* changed in the loop, then it can be called *strictly monotonic.* We will see how this distinction is used in dependence testing.

5 Other Concerns

5.1 Non-Basic Induction Variables

Basic induction variables (linear and nonlinear) and monotonic and periodic variables arise from nontrivial strongly connected regions (cycles) in the SSA graph. Variables which are functions of these variables must also be classified. To reiterate, one of the advantages of using Tarjan's algorithm is that when it comes time to classify a non-basic variable, all its arguments will have been already visited and classified. The result of each type of operation depends on how its operands have been classified.

The rules for classifying some operators are simple. For a load instruction, if the address is not invariant (such as indexed arrays, or computed pointer dereferencing) the type is unknown; if the address is invariant, and the *ssalink* of the load comes from outside the loop, the load is classified as invariant; otherwise, the load takes the classification of the *ssalink*. A store always takes the classification of the value being stored. A literal is always invariant.

The classification of arithmetic operators can be much more complex. The simple cases, adding or multiplying an invariant to an induction variable, etc., are easily handled. A generalized induction variable added to another induction variable will always be an induction variable, but multiplication may not be. For instance, multiplying (2^i+1) by (3^i-5) results in the geometric induction variable $(6^i+3^i-5\times 2^i-5)$, but multiplying (2^i+i) by (3^i-2i) results in the expression $(6^i+i3^i-i2^{i+1}-2i^2)$, which does not match the form of a generalized induction variable; it may, however, be classified as monotonic. In general the compiler needs an algebra of types and operators; space prevents a complete table of the algebra here, but the reader can come up with obvious examples (adding wrap around variables to get another wrap around variable, adding a monotonic variable to an induction variable to get another monotonic variable, etc.).

5.2 Trip Counts

The trip count of a loop can often be determined by finding the sequence of values used for the exit condition. If there is a single loop exit and the condition is an integer comparison, the compiler can convert the comparison into the form:

```
    if( $left \le right$ ) exit the loop
```

using the table below. If the loop conditional expression is as given on the left and the true branch exits the loop, it can be converted to the expression in the middle column (where we use rules of integer arithmetic to convert $\le$ to $<$); if the false branch exits the loop, then the condition must first be negated, so we get the expression in the right column below:

condition	exit	stay in
$a < b$	$a \le b-1$	$b \le a$
$a \le b$	$a \le b$	$b \le a-1$
$a > b$	$b \le a-1$	$a \le b$
$a \ge b$	$b \le a$	$a \le b-1$

The compiler can treat the comparison as a subtraction, and try to classify it as a linear induction sequence $\langle L, i, s\rangle$. The trip count of the loop (that is, the number of times the loop exit condition chooses to stay in the loop) can be computed as

$$\mathit{tripcount} = \begin{cases} 0 & \text{if } i \le 0 \\ \lceil i/(-s)\rceil & \text{if } i > 0 \text{ and } s < 0 \\ \infty & \text{if } i > 0 \text{ and } s \ge 0 \end{cases}$$

If the trip count can be found, the loop is called a *countable* loop. This is necessary below for handling multi-level induction variables. When a loop has multiple exits, the compiler may not be able to determine the exact number of iterations, but it may be able to find a maximum trip count; this information is useful for dependence testing, to place bounds on the solution space.

Note that some code in the loop above the exit test may have been executed even if the trip count is zero; in general, if the trip count is tc, some code above the exit test will have been executed $tc+1$ times while code below the exit test can have only been executed tc times. In single exit loops (where this is important), any unconditional code in the loop that is not *dominated* by the exit test will be executed one more iteration than unconditional code below the exit.

5.3 Nested Induction Variables

When dealing with nested loops, induction variable recognition proceeds from the inner loops outward. While processing any loop, any SSA links to code outside the loop are treated as loop invariant (for the purposes of Tarjan's algorithm and induction variable classification); these links may be chased down to find constant values. When an inner loop is classified as a countable loop, the cumulative effect of the execution of the loop on all induction variables in the loop can be expressed in closed form. Given a trip count tc, an induction variable $\langle L, i, s\rangle$ will end up with the value $i + tc \times s$, if all the increments are below the exit condition. By our definition of trip count, however, some induction variable increments may have been above the loop exit; the correct general formula is $i{+}tc{\times}s{+}s_{early}$, where s_{early} is the cumulative effect of any increment statements that can happen before the loop exits.

When the inner loop is processed, the compiler can construct an exit value for the induction variables, as

```
          $k_1$ = 0
     L17: loop
             $k_2$ = $\phi(k_1, k_5)$
             $i_1$ = 1
     L18:    loop
                $k_3$ = $\phi(k_2, k_4)$
                $i_2$ = $\phi(i_1, i_3)$
                $k_4$ = $k_3$ + 2
                if $i_2$ > 100 exit
                $i_3$ = $i_2$ + 1
             endloop
             $k_5$ = $k_4$ + 2
          endloop
```

Figure 7: Nested induction variables.

shown in Figures 7 and 8. This value can be assigned to a new variable, and all references outside this inner loop to the exit value are changed to refer to the new variable. In this example, the induction variables for the inner loop are:

$$\begin{aligned} k_3 &= \langle L18, k_2, 2\rangle \\ k_4 &= \langle L18, k_2 + 2, 2\rangle \\ i_2 &= \langle L18, 1, 1\rangle \\ i_3 &= \langle L18, 2, 1\rangle \end{aligned}$$

where i_1 is resolved to its constant value. The exit condition is converted to $100 \leq i_2 - 1$, which is treated as the induction expression $100-(i_2-1)$ or $\langle L18, 100, -1\rangle$; thus the trip count is 100. To compute the exit value of the induction variables `k` and `i`, the compiler notices that the k_4 statement preceeds the exit condition, so its increment value must be counted one extra time, and the k_6 and i_4 statements are added.

Now the induction variables for the outer loop can be found; if there are any remaining SSA links to variables in inner loops, these must correspond to non-induction variables or induction variables for which the exit value is unknown; in either case, the value can be treated as an unknown without tracing further. In this example, the outer loop has the induction variables:

$$\begin{aligned} k_2 &= \langle L17, 0, 204\rangle \\ k_6 &= \langle L17, 202, 204\rangle \\ k_5 &= \langle L17, 204, 204\rangle \end{aligned}$$

When the outer loops have all been processed, an outer-to-inner loop traversal can replace any outer loop induction variable expressions in inner loop induction variable initial values, if desired, changing k_3 and k_4 to:

```
          $k_1$ = 0
     L17':loop
             $k_2$ = $\phi(k_1, k_5)$
             $i_1$ = 1
     L18':   loop
                $k_3$ = $\phi(k_2, k_4)$
                $i_2$ = $\phi(i_1, i_3)$
                $k_4$ = $k_3$ + 2
                if $i_2$ > 100 exit
                $i_3$ = $i_2$ + 1
             endloop
             $k_6$ = $k_2$ + 101*2
             $i_4$ = $i_1$ + 100*1
             $k_5$ = $k_6$ + 2
          endloop
```

Figure 8: After processing inner loop induction variables.

```
          $j_1$ = 0
     L19: for i = 1 to n loop
             $j_2$ = $\phi(j_1, j_4)$
             $j_3$ = $j_2$ + 1
     L20:    for k = 1 to i loop
                $j_4$ = $\phi(j_3, j_5)$
                $j_5$ = $j_4$ + 1
             endloop
          endloop
```

Figure 9: Generalized induction variable from triangular inner loop.

$$\begin{aligned} k_3 &= \langle L18, \langle L17, 0, 204\rangle, 2\rangle \\ k_4 &= \langle L18, \langle L17, 2, 204\rangle, 2\rangle \end{aligned}$$

It is interesting to note that the case of generalized induction variables found to be so difficult in [EHLP92] is simple in this framework. Their given example has a doubly nested loop where the inner loop is *triangular*, that is, its upper limit depends on the outer loop index variable, as in Figure 9. Our method will process the inner loop, recognizing j_4 and j_5 as linear induction variables in that loop, with the trip count of that loop equal to a variable computed in the outer loop. The exit value of `j` will be computed as $j_6 = j_3 + i \times 1$, and j_4 will be replaced by j_6 in the assignment to j_2. When processing the outer loop, `i` will be recognized as a linear induction variable, while j_2, j_3 and j_6 will be recognized as a family of quadratic induction variables:

```
         k₁ = 0
L15':for i = 1 to n loop
         k₂ = φ(k₁,k₄)
         F(k₂) = A(i)
         if A(i) > 0 then
            C(k₂) = D(i)
            k₃ = k₂ + 1
            B(k₃) = A(i)
            E(i) = B(k₃)
         endif
         k₄ = φ(k₃,k₂)
         G(i) = F(k₄)
      endloop
```

Figure 10: Mixed monotonic and strictly monotonic variable

$$\begin{aligned} j_2 &= \langle L19, 0, \frac{3}{2}, \frac{1}{2}\rangle \\ j_3 &= \langle L19, 1, \frac{3}{2}, \frac{1}{2}\rangle \\ j_6 &= \langle L19, 2, \frac{5}{2}, \frac{1}{2}\rangle \end{aligned}$$

When the value for j_3 is substituted into the induction representations for the inner loop, we get:

$$\begin{aligned} j_4 &= \langle L20, \langle L19, 1, \frac{3}{2}, \frac{1}{2}\rangle, 1\rangle \\ j_5 &= \langle L20, \langle L19, 2, \frac{3}{2}, \frac{1}{2}\rangle, 1\rangle \end{aligned}$$

5.4 Loop Exit and Conditional Gates

When processing nested induction variables, our algorithm inserted a new assignment to a new SSA name (*e.g.* k_6 in Figure 8). Proper engineering can make this a low-cost insertion, but it would have been nice if there had already been an SSA name to distinguish the value of the variable upon loop exit. The loop exit η gating function used in the Program Dependence Web [BMO90] serves exactly this purpose. While we don't feel we need all the generality of the PDW, we are investigating adding some of its elements to the factored use-def chains used in our compiler internal representation.

In one other place the SSA representation is less precise than we would like. The SSA form of the monotonic variable in loop L15 (where we add a few more uses of k is shown in Figure 10. In this example, the compiler can easily find the SCR in the SSA graph, including the assignments to $\{k_2, k_3, k_4\}$. The loop entry value is k_2, and the ϕ-function for k_4 has two different offsets, k_2 and $k_2 + 1$; thus this is not a linear induction variable. However, all variables in the SCR are monotonically increasing. Even stronger, we can say that k_3 is monotonically *strictly* increasing, since if the k_3 assignment occurs more than once, it must assign a larger value each time. Thus, the dependence due to the assignment and reuse of array `B` will have dependence direction $(=)$ (see discussion in the next section); on the other hand, since k_2 and k_4 are only monotonic, the flow dependence due to array `F` has dependence direction $(\leq)$ (and there is an anti-dependence with direction $(<)$). However, we would also like to notice and take advantage of the fact that within the body of the conditional statement (*e.g.* at the assignment to array `C`), k_2 also must be strictly monotonic. One way to detect this would be to notice that any uses of k_2 in this region are *post-dominated* by the strictly monotonic assignment. Another way would be to again use elements of the Program Dependence Web, in this case to add a *switch* at the conditional; this would essentially give a unique name for each value entering a new control region.

6 Use in Dependence Testing

The driving force for classifying the variables in loops as shown in this paper is to improve the generality of dependence testing in loops, generating more precise dependence graphs and allowing more aggressive optimizations. Dependence testing for linear induction variables is generally well-covered in the literature [AK87, BCKT79, GKT91, MHL91]. Banerjee presents some algorithms for handling polynomial induction variables in his MS thesis [Ban76]. After a brief review, we focus on dependence with wrap-around variables, monotonic variables and periodic variables.

The algorithm used to classify variables will actually classify each subexpression as one of the generalized variable types. Thus, each subscript expression will be classified as an induction expression, monotonic expression, etc. Dependence testing in such a system involves constructing a dependence equation from the expressions. For instance, in the program:

```
          i = 0
          j = 3
     L21: loop
             i = i + 1
             A(i) = A(j-1)
             j = j + 2
          endloop
```

the left hand side subscript will be classified as the linear induction expression $\langle L21, 1, 1\rangle$, while the right hand subscript will be classified as $\langle L21, 2, 2\rangle$. From this, the compiler can immediately read off the coefficients to get the dependence equation:

$$1i' + 1 = 2i'' + 2$$

The dependence decision algorithm will determine whether there are integer values of the free variables i', i'' that solve the dependence equation and lie within the loop limits. Often, dependence distance or direction information can be found, determining that the solution only occurs when (for instance) $i'' - i' = 1$ or $i' < i''$ [WB87]. This information is critical to many optimization algorithms.

In the case where one of the expressions has the wrap-around attribute, the same dependence equation can be constructed and solved, but the dependence relation should be flagged as holding only after k iterations, the order of the wrap-around variable. This allows the compiler to decide whether the loop can be optimized before going through the headache of peeling off the first k iterations of the loop.

If the subscript expressions are members of the same periodic variable family, such as:

```
          j = 1
          k = 2
          l = 3
     L22: loop
             A(2*j) = A(2*k)
             temp = j
             j = k
             k = l
             l = temp
          endloop
```

the dependence equation should be constructed in terms of `j` and `k`:

$$2j = 2k$$

The dependence solvers determine whether there are integer values of j and k such that the dependence equation has a solution, and may determine distance or direction information in terms of `j` and `k`. What we need is to translate that distance or direction information into the appropriate information in terms of the loop iteration numbers. Suppose, as in this case, the decision algorithm determines that there is a solution when $j = k$. Let j_h refer to the value of `j` during iteration h. Because of the periodic nature of the variables, the compiler also knows that $j_{h'} = k_{h''}$ only when $h' \neq h''$. Thus, the $=$ direction for the dependence equation translates into a $\neq$ direction for the dependence relation; this is exactly the information needed to optimize the relaxation codes described earlier.

For monotonic variables, a similar situation occurs. The dependence equation is constructed in terms of the monotonic variables; if `m` is monotonic increasing, and the dependence equation is

$$2m' = 2m''$$

the dependence solver will find that there is dependence only when $m' = m''$. Now the compiler needs to translate this into direction information for the loop; since `m` is monotonic, $m_{h'} = m_{h''}$ can occur when $h' = h''$ or when $h' < h''$ (since `m` may not be incremented every iteration). Similar cases hold for strictly monotonic and monotonic decreasing variables.

6.1 Loop Normalization

Loop normalization is a linear transformation on the index set of a `for` loop to change the sequence of values of the loop variable to start at zero (or one) with a step of one; thus the loop:

```
for i = 2 to 11 by 3 loop
   ··· A(i) ···
```

will be changed to:

```
for i = 0 to (11-2)/3 loop
   ··· A(i*3+2) ···
```

This transformation was initially designed to simplify the formulation of data dependence testing algorithms [BCKT79]. Since normalization always puts the loop lower limits into subscript expressions, it can complicate life for simple dependence analyzers when the lower limit contains other variables, as shown by the work on Parafrase [SLY90]. For this reason, and because it can adversely affect the kinds of transformations allowed in programs (such as loop interchanging), this author has in the past argued against implementing loop normalization [Wol86]. It is interesting to note that this formulation of induction variables essentially normalizes all loops. One example used to argue against normalization is:

```
L23: for i = 1 to n loop
L24:    for j = i+1 to n loop
           A(i,j) = A(i-1,j)
```

Modern dependence analysis applied to this example will typically find a dependence *distance vector*, with one distance computed for each nested loop; see the tons of literature on this subject for further details [AK87, BCKT79, GKT91, MHL91]. In this case, the distance vector is $(1, 0)$, meaning distance one in the `i` loop and distance zero in the `j` loop. Normalizing the inner loop (to a lower limit of one) changes it to:

```
L23':for i = 1 to n loop
L24':   for j = 1 to n-i loop
           A(i,j+i) = A(i-1,j+i)
```

The computation in the loop hasn't changed, but the dependence distance vector has; now the distance vector is $(1, -1)$. While this may seem innocuous, some important transformations (such as loop interchanging) are prevented by this case.

In our framework, however, the shape of the loop iteration space is not part of the induction variable recognition strategy; in fact, *both* examples above will have the same results after induction variable recognition:

```
L23: for i = ··· loop
L24:     for j = ··· loop
              A(⟨L23, 1, 1⟩,⟨L24, ⟨L23, 2, 1⟩, 1⟩)
                 = A(⟨L23, 0, 1,⟨L24, ⟨L23, 2, 1⟩, 1⟩)
```

Intuitively, this strategy implicitly normalizes all induction variables with respect to a simple loop counter with initial value zero and step one (recall $h = \langle L14, 0, 1\rangle$). One strategy would be to keep the induction expression corresponding to the lower loop limits and factor these into the dependence equations. More likely, use of this induction variable representation will force compilers to abandon the *direction vector* dependence representation, which has been criticized by many researchers as too coarse [IT88]. A direction vector encodes the *sign* of the elements of the distance vector; while less precise in general, it can be used effectively when the distance vector is not constant. It may also force compilers to implement *loop skewing* and *loop interchanging* as a single transformation [Wol86]; this has been formulated in the past as a linear transformation on the index set [KMW67], and is currently in vogue as "unimodular transformations" [WL91, Ban91].

7 Summary and Conclusions

We have shown a fast and simple algorithm for classifying all induction variables in a loop; this algorithm is linear in the size of the SSA graph, not iterative. The algorithm is a somewhat obvious application of Tarjan's SCR algorithm on the SSA graph. It has several advantages, such as finding basic and non-basic induction variables in one pass. While usually used for applying strength reduction, we focus here on induction variable detection for dependence testing.

More importantly, we use the same algorithm to classify many different types of integer variables in loops, including polynomial and geometric induction variables, as well as flip-flop, periodic, wrap-around and monotonic variables. While some of these cases have been classified before, they were done by special case analysis instead of in a unified framework.

We have shown that the classification scheme is efficient, and how to use this information in data dependence testing. We have implemented the part of this algorithm that detects linear induction variables in our research compiler, and are adding the full algorithm at this time.

References

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

[AK87] John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[AWZ88] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages* [POP88], pages 1–11.

[Ban76] Utpal Banerjee. Data dependence in ordinary programs. M.S. thesis UIUCDCS-R-76-837, Univ. Illinois, Dept. Computer Science, November 1976.

[Ban91] Utpal Banerjee. Unimodular transformations of double loops. In Nicolau et al. [NGGP91], pages 192–219.

[BCKT79] Utpal Banerjee, Shyh-Ching Chen, David J. Kuck, and Ross A. Towle. Time and parallel processor bounds for Fortran-like loops. *IEEE Trans. on Computers*, C-28(9):660–670, September 1979.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 257–271, White Plains, NY, 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 Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CK77] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications ACM*, 20(11):850–856, November 1977.

[EHLP92] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*. Springer-Verlag, 1992. (to appear).

[FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Benjamin-Cummings, Menlo Park, CA, 1988.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* [SIG91], pages 15–29.

[IT88] François Irigoin and Rémi Triolet. Supernode partitioning. In *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages* [POP88], pages 319–329.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* [SIG91], pages 1–14.

[NGGP91] Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors. *Advances in Languages and Compilers for Parallel Computing*. Research Monographs in Parallel and Distributed Computing. MIT Press, Boston, 1991.

[POP88] *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages*, San Diego, CA, January 1988.

[PW86] David A. Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications ACM*, 29(12):1184–1201, December 1986.

[RWZ88] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages* [POP88], pages 12–27.

[SIG91] *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, June 1991.

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

[Tar72] R. 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 application to parallel processing. *International J. Parallel Programming*, 16(2):137–178, April 1987.

[WL91] Michael E. Wolf and Monica S. Lam. An algorithmic approach to compound loop transformations. In Nicolau et al. [NGGP91], pages 243–259.

[Wol86] Michael Wolfe. Loop skewing: The wavefront method revisited. *International J. Parallel Programming*, 15(4):279–294, August 1986.

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