Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers [*]

Peng Tu and David Padua
Center for Supercomputing Research and Development
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, Illinois 61801-2307
tu,padua@csrd.uiuc.edu

Abstract

In this paper, we present a GSA-based technique that performs more efficient and more precise symbolic analysis of predicated assignments, recurrences and index arrays. The efficiency is improved by using a backward substitution scheme that performs resolution of assertions on-demand and uses heuristics to limit the number of substitution. The precision is increased by utilizing the gating predicate information embedded in the GSA and the control dependence information in the program flow graph. Examples from array privatization are used to illustrate how the technique aids loop parallelization.

1 Introduction

Recent developments in parallelizing compilers have resulted in the increased use of the symbolic analysis technique to facilitate parallelism detection and program transformation. Several research compilers such as *Parafrase-2* [HP92] and *Nascent*[GSW] use symbolic analysis to identify and transform induction variables. In the *Polaris* [BEF$^+$94] restructuring compiler, symbolic analysis is also used for dependence analysis, symbolic range propagation and array privatization [BEF$^+$94] [TP93].

Simple symbolic analysis techniques are used regularly in the traditional optimizing compilers. Constant propagation detects symbolic variables that are equivalent to constants. Also, common subexpression elimination determines the equivalence of two symbolic expressions to avoid redundant computation. Symbolic analysis has also been used to prove assertions for program verification and debugging [CH78].

Recent experiments on the effectiveness of parallelizing compilers have found that symbolic variables present problems for dependence analysis and parallelization transformations. The most common case is induction variables, which are incremented or decremented by a certain amount in each iteration of a loop [ASU86]. If a loop contains an induction variable in the form of a recurrence, it cannot be executed in parallel due to the flow dependence in the recurrence. If the recurrence can be transformed into an equivalent expression that only depends on the loop index, replacing the induction variable with the equivalent expression enables the loop to be executed in parallel. Various techniques have been developed for linear induction variable recognition [Ken81] [ASU86] [ABCF88]. Recently, more aggressive symbolic analysis techniques have been developed to detect and replace nonlinear induction variables [HP92] [Wol92]. However, to successfully parallelize a broad range of application programs such as the Perfect Benchmarks [CKPK90], the compiler needs more symbolic analysis capabilities [BE94b]. These techniques include analysis of the possible values of variables to prove independent array accesses for dependence analysis, induction and reduction recognition to transform flow dependence, and scalar and array privatization to eliminate memory-related dependences.

Consider the following loop for dependence analysis:

```
do I = 1, 2*N
   if (I ≤ N) then
      A(I) = A(I+N) + C
   else
      B(I) = A(I+N) + C
   endif
enddo
```

If a naive compiler ignores the condition $(i \leq N)$ guarding the assignment to `A(I)`, it cannot prove that there exists no dependence on array `A`. The loop will then be executed in sequential mode. With more aggressive symbolic analysis, a compiler can find out that the variable `I` in the assignment statement to `A(I)` is only in the range of `[1,N]`. Therefore, the compiler can determine the loop assigns to the array section `A(1:N)` and uses the array section `A(N+1:2*N)`. Because these two sections do not overlap, the loop is parallel.

In our experience, *Array privatization* is also very important to eliminate memory-related dependences [TP93].

Consider the following segment of a program:

```
if (P)  then JLOW = 2, JUP = JMAX - 1
        else JLOW = 1, JUP = JMAX
do K = 1, N
        WORK(JLOW:JUP) = ...
        if (P)  then ...= WORK(2:JMAX-1)
enddo
```

The loop above cannot be executed in parallel because each iteration would read and write to the same location of array

[*]The research described was supported by contract DABT63-92-C-0033 from the Advanced Research Project Agency This work is not necessarily representative of the positions or policies of the U. S. Army or government.

WORK. The array privatization algorithm tries to determine if the loop can be executed in parallel if we allocate a private copy of array WORK to each iteration of the loop. This transformation is safe only when no iteration uses any data in the array WORK that were computed by other iterations. The WORK array is *privatizable* in the loop if we can prove that the section read, WORK(2:JMAX-1), is *covered* by the section written, WORK(JLOW:JUP). Again, we need symbolic analysis to prove that JLOW is less than or equal to 2 and JUP is greater than or equal to (JMAX-1).

This paper presents a new symbolic analysis technique for parallelizing compilers. It is based on Static Single Assignment (SSA) representation [CFR+91] of program. We use a more elaborate version of the SSA known as the Gated Single Assignment (GSA) form [BMO90]. The value of a symbolic variable is represented by a symbolic expression involving other symbolic variables, constants, and *pseudo-functions* in the GSA. We present algorithms to build and analyze the symbolic expressions. It combines the demand-driven interpretation of GSA and the path conditions to analyze complicated symbolic expressions. This technique has been implemented in the Polaris compiler. When combined with the known induction variable analysis techniques based on SSA, it serves as a tool for solving the following symbolic analysis problems arising in parallelizing compilers:

- Computing the symbolic values of expressions using program control dependences.
- Determining the relationship between symbolic expressions.
- Identifying recurrences, computing their closed form expressions, and determining symbolic upper and lower bounds.
- Determining the symbolic value of array elements used as array subscripts.

In Section 2, we introduce the GSA intermediate representation and the demand-driven backward substitution technique for building symbolic expressions. In Section 3, several rewriting rules are introduced to refine the possible values of symbolic expressions using control dependences. In Section 4, we present a technique to analyze recurrences. In Section 5, we extend our technique to determine the symbolic value of statically assigned index arrays. In Section 6, we present some preliminary experimental results of applying this technique for array privatization. We discuss related work in Section 7 and conclude in Section 8. Examples from dependence analysis and array privatization are used throughout the paper to illustrate the working principles and applicability of the technique.

2 Representing Symbolic Expression

The value of symbolic variables can be represented as symbolic expressions. Symbolic expressions consist of functions (operators) and arguments. In the absence of conditional statements, the symbolic value of a variable can be easily obtained by constructing a symbolic expression in terms of program input variables. For example, in the following straight-line code:

```
read(X,Y)
...
A = X + Y
...
B = A + 2
...
```

the value of A is (X+Y). If A is not reassigned, the value of B is (A+2). If X and Y are not re-assigned, A can be substituted by (X+Y). After the substitution, the value of B in terms of program input variables is (X+Y+2).

2.1 Static Single Assignment Form

For straight-line code, the symbolic value of a variable can be determined by repeatedly substituting arguments with their symbolic values. To do this substitution safely, all the variables in the expression must not be re-assigned from the definition to the use. For example, in the following code:

```
read(X,Y)
...
A = X+Y
X = 2*X
B = A+2
...
```

the X is reassigned. It is not safe to substitute A with (X+Y) in B = (A+2). Static Single Assignment (SSA) form can be used to relieve this problem. In the SSA form, each variable in the original program is replaced with several new variables, such that there is only one assignment to each new variable in the whole program. In this way, we can identify the different values of X with different names. The following is the same code in SSA form:

```
read($X_1$,$Y_1$)
...
$A_1 = X_1 + Y_1$
$X_2 = 2 * X_1$
$B_1 = A_1 + 2$
...
```

The symbolic value of B_1 can then be safely represented as $(A_1 + 2) = (X_1 + Y_1 + 2)$ in the SSA form.

2.2 Gated Single Assignment

The SSA form assigns different names to the different assignments to a variable. In this way, the *def-use* chain is embedded in the distinct names, such that for each use of a variable, there is only one reaching definition. In the presence of branch statements, a special merging *ϕ-function* is introduced in the SSA form. In the following SSA code:

```
read($X_1$,$Y_1$)
...
if (P) then
   $A_1 = X_1$
else
   $A_2 = Y_1$
endif
$A_3 = \phi(A_1, A_2)$
$B_1 = A_3 + 2$
...
```

an assignment to A_3 is introduced at the confluence of branches. It specifies that the value of A_3 is either A_1 or A_2 depending on which branch of the if statement is executed.

The ϕ-function can be directly used in the symbolic expression representation to account for the effect of branches. Hence, the symbolic value of B is $\phi(X_1, Y_1) + 2$.

Because ϕ-function does not contain the predicate for choosing the branches, symbolic expressions with ϕ-functions have only limited use. (Later in this paper we will discuss a technique to compute the lower and upper bounds of symbolic expressions that does not need the predicate for choosing branches.) Specialized *gating functions* are introduced in GSA to include the predicate of conditional branches. Different pseudo-functions replace the ϕ-functions at different confluence nodes in the program control flow graph.

A program *control flow graph* $CFG = (N, E)$ is a directed graph whose nodes N are the statements in a program. Each edge $x \rightarrow y \in E$ in the graph represents a possible flow of control from x to y. Two additional nodes, **Entry** and **Exit**, are added to the flow graph such that every entrance statement of the program has an edge from the **Entry** node and every exit statement of the program has an edge to the **Exit** node.

In a CFG, a node x *dominates* another node y, denoted as $x \underline{\gg} y$, if every path from **Entry** to y contains x. Node x *strictly* dominates y, denoted as $x \gg y$, if x dominates y and $x \neq y$. Node x is the *immediate dominator* of y, denoted as $x = idom(y)$, if x strictly dominates y and every other dominator of y (except y itself) dominates x. Every node in a flow graph except **Entry** has a unique immediate dominator. The edges $\{idom(y) \rightarrow y | y \in N - \{Entry\}\}$ form a *dominator tree* (DT) such that x dominates y if and only if x is a proper ancestor of y in the dominator tree (where the word *proper* means $x \neq y$).

Given a $CFG = (N, E)$ and its DT, edges in E from descendents to ancestors in DT are called *back edges*. Given a back edge $n \rightarrow d$, the *natural loop* [ASU86] of the edge are the subgraph of d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

As an extension of SSA, GSA introduces three new pseudo-functions.

- μ function : Replaces ϕ-functions at the head of a loop, as in the case of a *do* loop header.
- γ function : Replaces those ϕ-functions located at the confluence nodes that have no incoming back edges. A γ function takes the predicate guarding the paths to the confluence node, such as the conditions in *if* statements, as an additional parameter.
- η function : Replaces ϕ-functions at the nodes that contain loop exit edges as incoming edges. It selects the last value at the end of the loop.

In the previous example, the *gated* expression of A_3 is $\gamma(P, A_1, A_2)$. With the extra information on the predicate, we can determine that A_3 will have the value of A_1 if P is true, and the value of A_2 otherwise.

The following example further illustrates the use of those pseudo-assignment functions in GSA.

```
      X := 1
   L: do I = 1,N
         if (P) then X := X + 1
      enddo
in GSA form:
      $X_0$ := 1
   L: do I = 1,N {$X_1 := \mu((I = 1,N), X_0, X_3)$}
         if (P) then $X_2 := X_1 + 1$
         $X_3 = \gamma(P, X_2, X_1)$
      enddo
      $X_4 := \eta(I > N, X_1)$
```

The γ function in the assignment $X_3 = \gamma(P, X_2, X_1)$ returns X_2 or X_1, depending on the value of P. If P is true, the first argument is selected; otherwise the second argument is selected. The μ assignment is located in the control flow graph just before the do loop exit test. It merges the value of X computed outside the loop, X_0, with the value computed within the loop body, X_3. The loop header control condition (I=1,N) is the first argument of this function. In the first iteration, the μ function returns the second argument X_0, which is the value assigned before entering the loop; otherwise it returns the third argument X_3, which is the value from the previous iteration. The η function selects the last value of X computed by the loop. The η function, as defined in [BMO90], handles loops with a zero-trip count awkwardly. For this reason, we prefer to replace the assignment containing the η function above with

$$X_4 := \gamma(N < 1, X_0, \eta(I > N, X_1))$$

If the loop is a zero-trip loop, then X_4 will take the value X_0 from outside the loop. Otherwise, X_4 will take the value from inside the loop when the loop exit condition is satisfied. The $\eta(I > N, X_1)$ in the assignment above can be refined to $\eta(I = N + 1, X_1)$ because the step of I is 1 and N is an integer variable.

The most widely used algorithm to construct the SSA form is presented in [CFR$^+$91]. For GSA construction, the original algorithm in [BMO90] uses an SSA representation and the Program Dependence Graph (PDG). Another algorithm in [Hav93] uses SSA and the program control flow graph. These two algorithms can only handle reducible control flow graphs. Recently, we developed a more efficient algorithm that constructs GSA directly from program control flow graphs without going through the SSA conversion step [TP95]. It can be extended to handle irreducible flow graphs.

After converting a program into GSA form, we can use the symbolic expression with the pseudo gating functions to represent symbolic values of program variables.

2.3 Demand-Driven Backward Substitution

Traditional symbolic analysis propagates symbolic expressions by forward substitution. In a parallelizing compiler, the need for symbolic analysis is sparse. In many cases, we do not need symbolic analysis. Full scale forward substitution is expensive. Much of the information generated and propagated by forward substitution is never used. Furthermore, in many cases representing everything in terms of program inputs is not necessary, as illustrated next. Consider the following code segment:

```
R: JMAX = Expr
S: if (P)  then J := JMAX - 1
           else J := JMAX
T: assert(J ≤ JMAX)
```

To determine whether the assertion $(J \leq JMAX)$ is true at T, we need to know the symbolic value of J. Forward substitution starts at statement R. After it completes, J and JMAX at statement T are replaced by (*if* P *then* Expr$-$1 *else* Expr) and Expr respectively. Thus, the boolean expression $(J \leq JMAX)$ evaluates to true. It is easy to see that the substitution of JMAX by Expr is unnecessary. In a real program, forward substitution could produce long and complex expressions. Therefore, determining whether an assertion is true could be very time consuming. Approximate summary information could be used to improve the efficiency of this process. However, approximation usually decreases the accuracy of the analysis.

In Polaris, we use a demand-driven approach to improve the efficiency and accuracy of symbolic analysis. In a demand-driven approach, we seek information only when it is needed. Instead of propagating all symbolic values forward, our demand-driven strategy is goal directed and moves backward. Given a symbolic expression, we backward substitute arguments in the expression. The backward substitution stops when enough information to satisfy a specific objective has been obtained. In a forward substitution strategy, the requirements are not known and therefore it is difficult or impossible to determine which subset of the available information to propagate and where to start the propagation. For example, the code segment listed above has the following GSA form:

```
R:  JMAX$_1$ := Expr
S:  if (P)  then J$_1$ := JMAX$_1$ - 1
            else J$_2$ := JMAX$_1$
S': J$_3$ := $\gamma$(P, J$_1$, J$_2$)
T:  $assert$(J$_3$ $\leq$ JMAX$_1$)
```

Demand-driven analysis starts at T and performs backward substitution following the SSA links of the variables in the expression. The intermediate statements, which do not affect the variables used in T, are skipped. The steps of the substitution are:

$$\begin{aligned} J_3 &= \gamma(P, J_1, J_2) \\ &= \gamma(P, JMAX_1 - 1, JMAX_1) \end{aligned}$$

The backward substitution stops at this point because enough information for proving $J_3 \leq JMAX_1$ has been obtained. The redundant substitution of $JMAX_1$ by Expr is avoided.

Because the gating function captures the predicates of condition branches in its first argument, a regular function that contains no gating conditions can be safely distributed into the non-predicate arguments of a gating function. The rule is as follows:

$$\mathcal{F}(\mathcal{G}(P,X,Y),Z) = \mathcal{G}(P,\mathcal{F}(X,Z),\mathcal{F}(Y,Z)) \qquad (1)$$

where $\mathcal{F}$ is a regular function, and $\mathcal{G}$ is a gating function. For example, $\gamma(P,X,Y)^2 + expr = \gamma(P, X^2 + expr, Y^2 + expr)$.

2.4 Ordering of Backward Substitution

When we compare symbolic expressions, if there are several arguments that can be back-substituted, we need to decide which argument should be substituted first. A good order will result in a shorter substitution sequence. For instance, in the previous example, if we first substitute $JMAX_1$ with Expr, then J_3 would also have to be substituted until it becomes a function of Expr. This results in a longer substitution sequence.

Our strategy is as follows. Given two arguments *u,v* that can be back-substituted, if in the program flow graph the assignment statement of u (S_u) *dominates* the assignment statement of v (S_v), then v should be back-substituted before u. Because S_u dominates S_v, the value of u cannot depend on the value of v, while the value of v can depend on the value of u. When there are loops in the program, this order may not be valid since u can potentially depend on the v through some back edges. In the case of loop, we use a special technique to identify the set of variables that belong to the same *strongly connected region* (SCR). If S_u and S_v belong to different SCRs, the order is then determined by the dominance relation between the SCRs. To compute the substitution sequence, we use the *dominator tree* derived from program control flow graph. The dominator tree is needed for constructing the SSA form. Hence, it is readily available for use in our symbolic analysis.

When comparing two partially back-substituted symbolic expressions s and t, we can use the dominator tree to determine which argument in s, or t should be substituted first. If the arguments are substituted in the bottom-up order in the dominator tree, then it is possible to avoid expanding an expression beyond what is necessary for the comparison. This simple algorithm is shown below.

```
Algorithm Unify
Input: expressions $s$ and $t$
1. Mark constants and matching arguments
     in $s$, $t$ as dead.
2. while $\exists$ active arguments $\in s, t$ do
     Substitute an argument whose assignment
        statement is the lowest in the dominator tree.
     Mark the constants and matching arguments
        in the substituted expression as dead.
```

For instance, in the previous example, when comparing J_3 with $JMAX_1$, the assignment statement R for $JMAX_1$ dominates the assignment statements S, S', and T. Hence, J_3 is substituted first. J_1 and J_2 are substituted next. The substituted expression $\gamma(P, JMAX_1 - 1, JMAX_1)$ is comparable with $JMAX_1$. At this point, the backward substitution stops.

3 Determining Symbolic Values by Path Projection

To derive the value of a variable at a point p in a program, we first perform backward substitution as presented in the previous section. The result of backward substitution is a symbolic expression, *SE*, where each literal is either a constant or a variable name. In the backward substitution, pseudo-functions η, γ, and μ are treated as if they were ordinary functions. The pseudo-functions have special meaning representing the predicated conditional values of symbolic expressions.

Because the pseudo-functions capture the predicated conditional values of a symbolic expression, the symbolic expression obtained by backward substitution contains the set of possible values of the expression as a function of the predicates. This set of possible values can be narrowed down if the predicates in the relevant if statements are taken into account. A symbolic *path condition PC* is a predicate specifying the control flow condition under which the program flow will reach the statement p.

For instance, in the following program:

```
    if (P) then JUP$_0$ := JMAX - 1
           else JUP$_1$ := JMAX
    JUP$_2$ = $\gamma$(P, JUP$_0$, JUP$_1$)
    if (P) then
p:              ... := JUP$_2$
```

the path condition for statement p being executed is (P). The possible values of JUP_2 are $\gamma(P, JMAX - 1, JMAX)$. Since (P) is true at p, the value of JUP_2 at p can be determined to be JMAX-1.

To compute the path condition for each statement, we need to use the concept of *iterative control dependence*. In a *CFG*, a node v *post-dominates* a node w if every path from w to **Exit** contains v. A statement v is *control dependent* on statement u if v post-dominates a successor of u but does not strictly post-dominates u. The iterative control dependence is the transitive closure of the control dependence relation; that is, if v is control dependent on u and u is control dependent on w, then v is iteratively control dependent on w.

If p is iteratively control dependent on a collection C of branch statements, its path condition PC can be represented as a boolean expression involving the predicates in C. The path-restricted value PV of a symbolic expression at statement p is the projection of the possible values of SE, and is written as:

$$PV = SE(PC) \qquad (2)$$

To compute the projection we can use the following rules:

$$SE(PC) = SE \text{ if } SE \text{ contains no gating functions} \qquad (3)$$

$$\gamma(P, V_t, V_f)(PC) = \begin{cases} V_t(PC) & \text{if } PC \supset P \\ V_f(PC) & \text{if } PC \supset \neg P \\ \gamma(P, V_t(PC), V_f(PC)) & \text{otherwise} \end{cases} \qquad (4)$$

$$\mu(L, V_{init}, V_{iter})(PC) = \mu(L, V_{init}(PC), V_{iter}(PC)) \qquad (5)$$

$$\eta(P, V)(PC) = V(P \wedge PC) \qquad (6)$$

We present next some examples of backward substitution and path projection. The examples illustrate how these techniques improve the effectiveness of array privatization. The techniques are also useful in improving the accuracy of dependence analysis [BE94b, BE94a]. The main task when performing privatization of an array A is to determine whether in all iterations of the do loop each access to an element of A is dominated by an assignment to the same element. In order to prove that the use region of an array is always covered by a definition region, it is often necessary to determine the relationship between symbolic variables. The first example only requires the use of path conditions. Consider the following code segment:

```
   dimension XE(10000)
S: NDFE0 := NDDF0 * NNPED0
D: do i = 1, NDFE0
       XE(i) := ...
   enddo
U: do i = 1, NDDF0
       do j = 1, NNPED0
           ... := XE((i - 1) * NNPED0 + j)
       enddo
   enddo
```

To prove that the array region $\mathtt{XE(1 : NDFE_0)}$ defined in loop D covers the array region $\mathtt{XE(1 : NDDF_0 * NNPED_0)}$ accessed in loop U, we need to prove that $\mathtt{NDFE_0} \geq \mathtt{NDDF_0 * NNPED_0}$. This is easily done after $\mathtt{NDFE_0}$ is replaced by $\mathtt{NDDF_0 * NNPED_0}$ using backward substitution. The path condition for those points within loop U where XE is accessed is $PC_U = (\mathtt{NDDF_0} \geq 1 \wedge \mathtt{NNPED_0} \geq 1)$. The path condition at those points where XE is defined is $PC_D = (\mathtt{NDFE_0} \geq 1)$ or, after the backward substitution, $PC_D = (\mathtt{NDDF_0 * NNPED_0} \geq 1)$. It is easy to see that $PC_U \supset PC_D$ and, therefore, whenever loop U has a non-zero trip count, loop D does too.

The next example illustrates the use of the projection rule for γ functions. Backward substitution involving the μ function and associated recurrences will be discussed in the next section. Consider the following code segment:

```
    if (P) then JLOW0 := 2,  JUP0 := JMAX - 1
           else JLOW1 := 1,  JUP1 := JMAX
    JLOW2 = γ(P, JLOW0, JLOW1)
    JUP2 = γ(P, JUP0, JUP1)
L:  do K = 1, N
D:         WORK(JLOW2 : JUP2) = ...
U:         if (P) then ... = WORK(2 : JMAX - 1)
    enddo
```

For the array WORK to be private to the loop L, we need to determine that the use of $\mathtt{WORK(2, JMAX - 1)}$ at U is covered by the definition of $\mathtt{WORK(JLOW_2 : JUP_2)}$ at D. The PC at U is P. Using the projection rule for the γ function under the condition P, we have the following replacements which prove the desired condition:

$$\begin{aligned} \mathtt{JLOW_2} &= \gamma(\mathtt{P}, \mathtt{JLOW_0}, \mathtt{JLOW_1})(\mathtt{P}) \\ &= \mathtt{JLOW_0}(\mathtt{P}) \\ &= 2 \\ \mathtt{JUP_2} &= \gamma(\mathtt{P}, \mathtt{JUP_0}, \mathtt{JUP_1})(\mathtt{P}) \\ &= \mathtt{JUP_0}(\mathtt{P}) \\ &= \mathtt{JMAX} - 1 \end{aligned}$$

The path condition may be used to derive useful information about a symbolic variable even if the symbolic expression does not contain any predicated condition values. For instance, consider again our first example:

```
    do I = 1, 2*N
       if (I ≤ N) then
R:        A(I) := A(I + N) + C
       else
          B(I) := A(I + N) + C
       endif
    enddo
```

At statement R, the PC is $1 \leq \mathtt{I} \leq \mathtt{N}$. This makes it possible to determine that there are no cross iteration dependences for array A in the loop. Incorporating the path condition in the analysis provides us with more power than GSA or SSA alone.

3.1 Bounds of Symbolic Expression

The problem of determining whether $PC \supset P$ is undecidable in general and NP-Complete [GJ79] when all the conditions are boolean variables. Fortunately, path projection is most useful when the path condition obviously implies the predicates in the symbolic expressions. When the number of boolean variables in PC and SE is small, the cost of computing the path projection is also small. Sometimes, as in array privatization and dependence tests, it is sufficient to obtain the upper and lower bounds of a symbolic variable.

In the case of bounds, we can often ignore the predicates that cannot be easily resolved by path projection and apply the minimum and maximum functions directly to the possible values:

$$\max(\gamma(P, V_t, V_f)) \leq \max(V_t, V_f) \qquad (7)$$

$$\min(\gamma(P, V_t, V_f)) \geq \min(V_t, V_f) \qquad (8)$$

Using these two rules in our previous example, we obtain:

$$\begin{aligned} \mathtt{max(JLOW_2)} &\leq \mathtt{max}(\gamma(\mathtt{P}, \mathtt{JLOW_0}, \mathtt{JLOW_1})) \\ &= \mathtt{max(JLOW_0, JLOW_1)} \\ &= \mathtt{max(2, 1)} \\ &= 2 \\ \mathtt{min(JUP_2)} &\geq \mathtt{min}(\gamma(\mathtt{P}, \mathtt{JUP_0}, \mathtt{JUP_1})) \end{aligned}$$

$$
\begin{aligned}
&= \min(\mathtt{JUP}_0, \mathtt{JUP}_1)\\
&= \min(\mathtt{JMAX}-1, \mathtt{JMAX})\\
&= \mathtt{JMAX}-1
\end{aligned}
$$

which also prove the desired condition.

3.2 Comparing Symbolic Expressions

The symbolic expression may still contain γ functions after path projection. In symbolic analysis, we sometimes need to compare these expressions. Alpern, Wegman, and Zadeck [AWZ88] define a *congruence relation* between expressions containing ϕ-assignments. The congruent variables are shown to have equivalent values under structural isomorphism, i.e., if they can be mapped to the same canonical form. Structural isomorphism can only be used to determine equality; it cannot determine, for example, if one expression is always larger than another. We define next a class of expression pairs whose inequality relationship can be determined at compile-time.

We loosely call two expressions *compatible* if, after backward substitution, the non-constant literals in one expression are a subset of the terms in the other. Only when two expressions are compatible can their relationship be determined symbolically. Notice that the structurally isomorphic expressions are compatible. For the purpose of comparison, two compatible expressions E^1, E^2 can be classified as follows:

1. **None of E^1 and E^2 contains any pseudo-function:** The expressions can be compared by simplifying $E^1 - E^2$ symbolically. Their relationship can be determined if the result is a constant.

2. **Only one of E^1 and E^2 contains pseudo-functions:** The comparison will be based on the arguments of the γ function. To illustrate the method, assume that $E^2 = \gamma(P, V_t, V_f)$, where V_t, V_f may also contain pseudo-functions. To determine whether $E^1 > E^2$, we reduce it to case 1 using the following necessary and sufficient condition:

$$(E^1 > \gamma(P, V_t, V_f)) \equiv (E^1 > V_t) \wedge (E^1 > V_f) \tag{9}$$

   If V_t, V_f contains any pseudo-functions, the same procedure should be used recursively on the right-hand side of the above equation. The result can then be evaluated as in case 1. An equivalent approach is to compute the minimum and maximum values for E^2 using the technique discussed above. Because E^1 does not contain any pseudo-function, it can be proven that:

$$(E^1 > E^2) \equiv (E^1 > \max(E^2)) \tag{10}$$

3. **Both E^1 and E^2 contain pseudo-functions:** There are several ways to handle this case. We will illustrate just one here. Assume again that $E^2 = \gamma(P, V_t, V_f)$. To prove $E^1 > E^2$, the necessary and sufficient condition is:

$$E^1 > \gamma(P, V_t, V_f) \equiv (E^1(P) > V_t) \wedge (E^1(\neg P) > V_f) \tag{11}$$

   Because E^1 contains a pseudo-function, the path projections $E^1(P)$ and $E^1(\neg P)$ are necessary in the above equation to cast the branching conditions to E^1. For instance, if $E^1 = \gamma(P, V_t', V_f')$, the condition can be evaluated as follows:

$$
\begin{aligned}
&(\gamma(P, V_t', V_f')(P) > V_t) \wedge (\gamma(P, V_t', V_f')(\neg P) > V_f)\\
&\quad = (V_t' > V_t) \wedge (V_f' > V_f)
\end{aligned}
$$

   Each application of this rule eliminates one pseudo-function. It is applied recursively until the right-hand side is free of pseudo-functions. The problem is then reduced to case 2.

We will call these three rules *distribution rules.* Rule 3 subsumes rule 2 because path projection has no effect on an expression that does not contain any pseudo-function. Note that for determining equalities, techniques based on structural isomorphism cannot identify equalities when the order of the γ functions in two expressions is different. Using the distribution rules, we can define a canonical ordering of predicates and fully distribute every expression into a canonical form to identify these equalities.

The distribution rule discussed also applies to more complex expressions. For example, consider $E^2 = \gamma(P, V_t, V_f) + exp$. This expression can be normalized to $\gamma(P, V_t + exp(P), V_f + exp(\neg P))$. In the analysis algorithm, normalization is deferred until a distribution is made on the γ function in order to allow the common components in E^1 to be canceled out with $\gamma(P, V_t, V_f)$ or exp.

The following are some examples of using these rules. We have used rule 2 in one of our previous examples:

$$
\begin{aligned}
\mathtt{J}_3 &= \gamma(\mathtt{P}, \mathtt{JMAX}_1 - 1, \mathtt{JMAX}_1)\\
&\le \mathtt{JMAX}_1
\end{aligned}
$$

In the following example, we use rule 3 to derive the condition for $\mathtt{JUP}_2 > \mathtt{JLOW}_2$ under P in our early example.

$$
\begin{aligned}
(\mathtt{JUP}_2 > \mathtt{JLOW}_2) &\equiv \gamma(\mathtt{P}, \mathtt{JMAX}-1, \mathtt{JMAX}) > \gamma(\mathtt{P}, 2, 1)\\
&\equiv \gamma(\mathtt{P}, \mathtt{JMAX}-1 > 2, \mathtt{JMAX} > 1)\\
&\equiv \gamma(\mathtt{P}, \mathtt{JMAX} > 3, \mathtt{JMAX} > 1)
\end{aligned}
$$

When two predicated expressions contain a lot of predicated conditional values, using the distribution rule 3 can result in fast growth of the expression size. In our experience, restricting the nesting level of gating functions to 2 seems to be enough. Larger than 2, the expressions are usually not comparable. For levels larger than 2, we compute the bounds and ignore the predicates. It uses the following approximation rule:

$$(\min(E^1) > \max(E^2)) \supset (E^1 > E^2) \tag{12}$$

4 Recurrence and the μ Function

Backward substitution of an expression involving a μ function will form a recurrence because of the back edge in a loop. The value carried from the previous iteration of a loop is placed in the second argument of a μ function. Hence, rule for the substitution of a μ function is to substitute the second parameter of the μ function until it becomes an expression of the variable itself or an expression of another μ assigned variables. This is illustrated in the following example.

```
L: do I = 1, N {$J_1 := \mu((I = 1,N), J_0, J_3)$}
        if (P) then $J_2 := J_1 + A$
        $J_3 := \gamma(P, J_2, J_1)$
   enddo
```

Substituting terms in the μ function leads to:

$$
\begin{aligned}
\mathtt{J}_1 &= \mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \mathtt{L}_3)\\
&= \mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \gamma(\mathtt{P}, \mathtt{J}_2, \mathtt{J}_1))\\
&= \mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \gamma(\mathtt{P}, \mathtt{J}_1 + \mathtt{A}, \mathtt{J}_1))
\end{aligned}
$$

The recurrence can be interpreted as the following λ-function over the loop index.

$$\lambda\mathtt{i}.(\mathtt{J}_1(\mathtt{i})) \equiv \gamma(\mathtt{i}=1, \mathtt{J}_0, \gamma(\mathtt{P}, \mathtt{J}_1(\mathtt{i}-1)+\mathtt{A}, \mathtt{J}_1(\mathtt{i}-1)))$$

This λ-function can be interpretated in the terms of recursive sequence in combinatorial mathematics as follows:

$$J_0 = J_0$$

$$J_i = \begin{cases} J_{i-1}+1 & \text{if (P)} \\ J_{i-1} & \text{otherwise} \end{cases} \quad \text{for } i \in [1:N]$$

After the mathematical form of a recurrence is identified, standard methods for solving a linear recurrence can be used to find a closed form. For instance, if P is always true and A is loop invariant, then $\mathtt{J}_1(\mathtt{i})$ is an induction variable with value $\mathtt{J}_0 + (\mathtt{i}-1) * \mathtt{A}$. When P is always true and A is a linear reference to an array, for example X(i), $\mathtt{J}_1(\mathtt{i})$ is a reduction sum over X.

Wolfe and others [Wol92] developed a comprehensive technique to classify and solve recurrence sequences using a graph representation of SSA. In this graph representation, a recurrence is characterized by a Strongly Connected Region (SCR) of use-def chains. The backward substitution technique for μ functions is equivalent to Wolfe's SCR approach. Their technique for computing the closed form of constant coefficient linear recurrence can be directly used in our scheme. However, we feel that our backward substitution scheme which works directly with the algebra of names, functions and expressions is better than their approach which works indirectly through graph and edges. We can also deal with the cases where no closed form expression can be obtained. For instance, if the condition P is loop invariant, symbolic substitution can determine that the value of $\mathtt{J}_1(\mathtt{i})$ is either $\mathtt{J}_0 + (\mathtt{i}-1) * \mathtt{A}$ or $\mathtt{J}_0$.

When the closed form of a recurrence cannot be determined, it may still be possible to compute a bound by selecting the γ arguments with maximum or minimum increment to the recurrence. For example:

$$\begin{aligned}
\max(\mathtt{J}_1) &\le \max(\mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \gamma(\mathtt{P}, \mathtt{J}_1+\mathtt{A}, \mathtt{J}_1))) \\
&= \mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \max(\gamma(\mathtt{P}, \mathtt{J}_1+\mathtt{A}, \mathtt{J}_1))) \\
&= \mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \mathtt{J}_1+\mathtt{A}) \\
\min(\mathtt{J}_1) &\ge \min(\mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \gamma(\mathtt{P}, \mathtt{J}_1+\mathtt{A}, \mathtt{J}_1))) \\
&= \mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \min(\gamma(\mathtt{P}, \mathtt{J}_1+\mathtt{A}, \mathtt{J}_1))) \\
&= \mu((\mathtt{I}=1,\mathtt{N}), \mathtt{J}_0, \mathtt{J}_1)
\end{aligned}$$

The resulting two recurrence functions can now be solved to obtain the upper bound $\mathtt{J}_0 + \mathtt{N} * \mathtt{A}$ and the lower bound $\mathtt{J}_0$.

5 Index Array

The use of array elements as subscripts makes dependence analysis and array privatization more difficult than when just scalars are used. When the values of an index array depend on the program's input data, run-time analysis has to be used. However, in many cases the index arrays used in the program are assigned symbolic expressions. For instance, a wide range of applications use regular grids. Although the number of grids is input dependent, the structure of the grid is fixed. The structure is statically assigned to index arrays with symbolic expressions parameterized by the some unknown input variables. In these cases, the value of an index array can be determined at compile-time. Consider the following segment of code:

```
L: do J=1, JMAX
       JPLUS(J) := J + 1
   enddo
   JPLUS(JMAX) := Q
U: ...
```

It is possible to determine at compile-time that the array element JPLUS(J) has value of J + 1 for $\mathtt{J} \in [1, \mathtt{JMAX}-1]$ and Q for J = JMAX. We can use the GSA representation to find out the value of JPLUS(J) at statement U. To this end, we use an extension of the SSA representation to include arrays in the following way [CFR+91]:

1. Create a new array name for each array assignment.
2. Use the subscript to identify which element is assigned.
3. Replace the assignment with an update function $\alpha(array, subscript, value)$.

For example, an assignment of the form JPLUS(I) = exp will be converted to $\mathtt{JPLUS}_1 = \alpha(\mathtt{JPLUS}_0, \mathtt{I}, \mathtt{exp})$. The semantics of the α function is that $\mathtt{JPLUS}_1(\mathtt{I})$ receives the value of exp while the other elements of $\mathtt{JPLUS}_1$ will take the values of the corresponding elements of $\mathtt{JPLUS}_0$. This representation maintains the single assignment property for array names. Hence the def-use chain is still maintained by the links associated with unique array name. Using this extension, our example can be transformed into the following SSA form:

```
L: do J = 1, JMAX
       {$\mathtt{JPLUS}_2 := \mu((\mathtt{J}=1,\mathtt{JMAX}), \mathtt{JPLUS}_0, \mathtt{JPLUS}_1)$}
       $\mathtt{JPLUS}_1 := \alpha(\mathtt{JPLUS}_2, \mathtt{J}, \mathtt{J}+1)$
   enddo
   $\mathtt{JPLUS}_3 := \alpha(\mathtt{JPLUS}_2, \mathtt{JMAX}, \mathtt{Q})$
```

To find out the value of an element $\mathtt{JPLUS}_3(\mathtt{K})$ in $\mathtt{JPLUS}_3$, we can use backward substitution as follows:

$$\begin{aligned}
&\mathtt{JPLUS}_3(\mathtt{K}) = \alpha(\mathtt{JPLUS}_2, \mathtt{JMAX}, \mathtt{Q})(\mathtt{K}) \\
&= \gamma(\mathtt{K}=\mathtt{JMAX}, \mathtt{Q}, \mathtt{JPLUS}_2(\mathtt{K})) \\
&= \gamma(\mathtt{K}=\mathtt{JMAX}, \mathtt{Q}, \mu((\mathtt{J}=1,\mathtt{JMAX}), \mathtt{JPLUS}_0, \mathtt{JPLUS}_1)(\mathtt{K})) \\
&= \gamma(\mathtt{K}=\mathtt{JMAX}, \mathtt{Q}, \gamma(1 \le \mathtt{K} \le \mathtt{JMAX}, \mathtt{JPLUS}_1(\mathtt{K}), \mathtt{JPLUS}_0(\mathtt{K}))) \\
&= \gamma(\mathtt{K}=\mathtt{JMAX}, \mathtt{Q}, \gamma(1 \le \mathtt{K} \le \mathtt{JMAX}, \mathtt{K}+1, \mathtt{JPLUS}_0(\mathtt{K})))
\end{aligned}$$

In the above symbolic evaluation process, an expression of the form $\alpha(\mathtt{X}, \mathtt{i}, \mathtt{exp})(\mathtt{j})$ is evaluated to $\gamma(\mathtt{j}=\mathtt{i}, \mathtt{exp}, \mathtt{X}(\mathtt{j}))$. An expression of the form $\mu(\mathtt{L}, \mathtt{Y}, \mathtt{Z})(\mathtt{j})$ is instantiated to a list of γ functions that select different expressions for different values of j. These evaluation rules are straightforwardly derived from the definitions of the gate functions. To avoid unnecessary array renaming, GSA conversion is done only on those arrays that are used as subscripts.

6 Implementation and Experiments

We have partially implemented the technique for predicated conditional value analysis, bounds analysis of monotonic variables, and index array analysis in the Polaris parallelizing compiler. Currently, it serves to provide symbolic information for array privatization.

Table 1 shows the improvement we obtained using the symbolic analysis for array privatization. The columns in the table show the number of privatizable arrays identified with and without symbolic analysis. Of the six programs tested, four show an increased number of privatizable arrays. Also shown are the increase in the parallel loop coverage in percentage of the total execution time, the speedups

Program	Pri. (w/o SA)	Pri. (with SA)	Increased Coverage %	Speedups (8 proc)	Cond. Value	Bounded Recurrence	Index Array
ARC2D (SR)	0	2	15.3	4.5	x		x
BDNA (NA)	16	19	33.6	4.5	x	x	x
FLO52 (TF)	4	4	0.0	2.6			
MDG (LW)	18	19	97.7	4.9	x		x
OCEAN (OC)	4	7	42.7	1.2	x		
TRFD (TI)	4	4	0.0	2.7			

Table 1: Effect of symbolic analysis on array privatization

obtained on an 8 processor SGI Challenge, and the techniques applied in each program.

The experiment shows that a small increase in the number of private arrays can make a big difference in the parallelism discovered. In *MDG*, privatization with symbolic analysis privatizes only one more array than that without symbolic analysis. But it makes a loop that accounts for 97 percent of the program execution time to be parallel. Among the analysis techniques, the predicated conditional value is used most often, followed by statically assigned symbolic index array analysis. The bounded recurrence column shows only the cases where there is no closed form for the recurrence due to conditional increment. Induction variables with closed forms occur in all these programs. At this moment, Polaris uses other techniques for induction variable substitution. The effect of induction variable substitution on array privatization is not shown here.

The array privatizer in Polaris has been converted to work directly on the GSA representation. Using the GSA representation of use-def chains, the privatizer shows an average speedup of 6 when comparing with the original implementation based on the control flow graph. The demand-driven symbolic analysis part in the privatizer takes less than 1 percent of the total execution time of the privatizer, mainly because the frequency of requiring symbolic analysis is very low. We will report the timing results in a coming technical report.

7 Discussion and Related Work

The problem of determining the relationship between symbolic expressions is undecidable in general. In practice, symbolic expressions often contain program input variables whose run-time values may be needed in the analysis. Therefore, the goal of symbolic analysis is to handle with reasonable efficiency some of the situations that arise frequently in practice.

In symbolic analysis, the values of variables are represented by symbolic expressions. Some of the techniques designed in the past proceed by computing *path values*, the set of symbolic values of a variable on all possible paths reaching a program point. Corresponding to each path there is a *path condition*, a boolean expression that is true if the path is executed [CR85] [CHT79]. These techniques have exponential time and space requirements which limit their applicability in practical situations.

The GSA representation of symbolic expression in this paper is a new way to represent *predicated path values*. With GSA, the condition and value is represented in a compact way. Expression in GSA can be manipulated in a way similar to the normal arithmetic expression with special treatment for gating functions. We also use the iterative control dependences to represent path conditions.

Symbolic analysis is also related to the problem of automatic proof of invariant assertions in program. Symbolic execution can be considered as abstract interpretation [CH92]. In [CH78], abstract interpretation is used to discover the linear relationships between variables. It can be used to propagate symbolic linear expressions as possible values for symbolic variables. However, the predicate guarding the conditional possible values is not included in the representation and cannot be propagated. The *parafrase-2* compiler also uses a similar approach to evaluate symbolic expressions. They use a *join function* to compute the intersection of possible values at the confluence nodes of the flow graph to cut down the amount of information to be maintained. Again the predicated conditional information is lost in the join function. Abstract interpretation has also been used in [AH90] for induction variable substitution.

Most systems use symbolic global *forward substitution* or propagation to compute values and conditions at all points in the program. The problem is that all the points and all the variables in the program are treated in the same way regardless of the needs for symbolic analysis. The result is that most of the information propagated is not actually used. To reduce the cost of the analysis, they usually have to use more restrictive join (union) functions at the confluence nodes to cut down the information. However, the restricted information may not be accurate enough to be of any use.

The demand-driven approach in this paper is a special case of the more general approach of *Subgoal Induction* [MW77]. Two facts make it possible: (1) In the SSA representation, the variable name is unique and the use-def chain is embedded in the unique variable name; (2) backward substitution can be done in the Gated SSA by following the use-def chain in the name without going through the flow graph. Using the GSA sparse representation and the demand-driven approach, our approach can do more aggressive analysis only on demand.

The SSA form has been used to determine the equivalence of symbolic variables and construct global value graph in a program [AWZ88] [RWZ88]. *Nascent* [Wol92] uses SSA to do a comprehensive analysis of recurrences. Their representation does not include the gating predicate. The SSA representation in Nascent is through an explicit use-def chain which they called a demand-driven form of SSA. Their approach for constructing strongly connected regions for recurrences in the graph is similar to our backward substitution of names. GSA has also been used in *Parascope* to build the global value graph [Hav93]. Our demand-driven backward substitution in GSA and using control dependences to project values are unique.

The currently most used algorithm for building SSA form is given in [CFR$^+$91]. GSA was introduced in [BMO90] as a part of the Program Dependence Web (PDW) which is an extension of the Program Dependence Graph (PDG) [FOW87]. An algorithm for constructing GSA from PDG and SSA is given in [BMO90]. Havlak [Hav93] develops another algorithm for building GSA from a program control flow graph and SSA. Recently, we developed a new algorithm to efficiently construct the GSA directly from the control flow graph [TP95]. Control dependence was introduced in [FOW87] as part of PDG. The SSA construction paper [CFR+91] also has an algorithm for building control dependences. The algorithm in [CFR+91] for building iterative dominance frontiers can be used on the reverse flow graph to build the iterative post-dominance frontiers (iterative control dependences) used in this paper.

8 Conclusion

Symbolic analysis is important to parallelizing compilers. With the traditional forward substitution technique for symbolic analysis, it is difficult to strike a balance between efficiency and accuracy. Forward substitution usually propagates too much unnecessary information, and propagates too little information for the few important variables in the analysis.

We have proposed a technique to derive information about symbolic variables in a demand-driven way that is more efficient and accurate. In contrast to global forward substitution, this demand-driven technique introduces no overhead when there is no need for propagation. Because it only uses the more elaborate techniques on demand, it can do more aggressive analysis for complicated symbolic expressions.

We have illustrated the use of this technique in array privatization to determine the symbolic values of array reference regions. We have also shown its effectiveness in the environment of the Polaris parallelizing compiler. The experiment has shown that the aggressive analysis technique presented is important for parallelizing some important loops in the Perfect Benchmarks.

References

[ABCF88] F. Allen, M. Burke, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, (5):617–640, October 1988.

[AH90] Z. Ammarguellat and W. L. Harrison. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–295. ACM Press, 1990.

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

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[BE94a] William Blume and Rudolf Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of Supercomputing '94*, November 1994.

[BE94b] William Blume and Rudolf Eigenmann. Symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. Technical Report 1332, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., January 1994.

[BEF+94] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Proc. 7th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1994.

[BMO90] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN'90 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*, 13(4):451–490, October 1991.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, January 1978.

[CH92] P. Cousot and N. Halbwachs. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CHT79] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, 5(4):402–417, 1979.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the perfect benchmarks. In *Proceedings of ICS, Amsterdam, Netherlands*, pages 162–174, March 1990.

[CR85] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, 1985.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependency graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, June 1987.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of the NP-Completeness.* Freeman, 1979.

[GSW] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *to appear on, ACM TPLAS.*

[Hav93] Paul Havlak. Construction of thinned gated single-assignment form. In *Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, August 1993.

[HP92] M. R. Haghighat and C. D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Proc. 5rd Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[Ken81] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis.* Prentice-Hall, 1981.

[MW77] J. H. Morris and B. Wegbreit. Subgoal induction. *Communication of ACM*, 20(4):209–222, 1977.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computation. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[TP93] Peng Tu and David Padua. Automatic array privatization. In *Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, August 1993.

[TP95] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[Wol92] Michael Wolfe. Beyond induction variables. *Proc. the SIGPLAN'92 Conference on Programming Language Design and Implementation*, June 1992.