Gated SSA Based Demand-Driven Symbolic Analysis *

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

1 Introduction

It has become increasingly evident that symbolic manipulation of expressions is necessary to support a number of analysis and transformation techniques used by parallelizing compilers. For example, symbolic analysis increases the accuracy of dependence analysis and array privatization [BEF+94][TP93], which are perhaps the two most important techniques for automatic parallelization. This paper presents a technique to determine the equality and inequality relations between symbolic expressions. The problem of determining the relationship between symbolic expressions is undecidable in general. In practice, symbolic expressions often contain the program input variables whose run-time values may be needed in the analysis. Therefore, the goal of the techniques presented below 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. Other techniques discussed in the literature use abstract interpretation [CH78][HP92]. Instead of computing and keeping all the values and conditions reaching a point in the program, a join function is used to compute the intersection of these values and conditions. In this approach (and depending on the join function), some information might be lost at the confluence points in the control flow graph of the program. Both approaches use symbolic global *forward substitution* or propagation to compute values and conditions at all the points in the program. The ultimate objective is to represent program expressions exclusively in terms of input values and constants.

Symbolic forward substitution is not very efficient for three reasons. First, the size of the expressions may grow exponentially with the number of paths reaching them. The use of join functions solves this problem, but at the expense of accuracy. Second, much of the information generated and propagated by forward substitution is never used. Finally,

*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 the government.

in many cases, representing everything in terms of program inputs is not necessary, as illustrated below. 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 ≤ 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 ≤ 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, in general this decreases the accuracy of the analysis.

The objective of the techniques discussed in this paper is to improve the efficiency and accuracy of symbolic analysis by using a demand-driven approach, that is, by seeking information only when it is needed. Instead of propagating all symbolic values forward, a demand-driven strategy is goal directed and moves backward. *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. In the previous example, a demand-driven analysis of values of J and JMAX would start from the assertion at T and would stop at statement S where enough information to prove (J ≤ JMAX) has been obtained. The redundant substitution of JMAX by Expr does not have to be performed.

To implement backward substitution, we use a Static Single Assignment (SSA) [AWZ88] [RWZ88] program representation. The SSA form is obtained by renaming the scalar variables in the program so that the left-hand sides of all scalar assignments are different. A special function, ϕ, is used at the confluence points of the control flow graph when it is necessary to select one of several renamed versions of a scalar variable. SSA has been used for determining equivalence of symbolic variables [AWZ88] and parallelization of imperative programs [CF87]. An efficient algorithm for constructing a minimal SSA representation of an imperative program can be found in [CFR+91].

For example, the code segment listed above has the following SSA form:

```
R:  JMAX1 := Expr
S:  if (P)  then J1 := JMAX1 − 1
            else J2 := JMAX1
S': J3 := φ(J1, J2)
T:  assert(J3 ≤ JMAX1)
```

A 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 result of the substitution is:

$$J_3 = \phi(J_1, J_2)$$

$= \phi(\texttt{JMAX}_1 - 1, \texttt{JMAX}_1)$
$\leq \texttt{JMAX}_1$

We will discuss below a simple technique for deciding when to stop the backward substitution. For example, in the code we are discussing, this technique would avoid the substitution of $\texttt{JMAX}_1$ by `Expr` which is redundant.

The ϕ function in the example above could be augmented with the predicate of the conditional statement preceding it. Statement `S'` would then take the form:

$\texttt{J}_3 := \phi(\texttt{P}, \texttt{J}_1, \texttt{J}_2)$

This form of the ϕ function returns the second or third parameter depending on the value of the predicate. This type of function is called a high-level ϕ function and was introduced in [AWZ88]. The Gated Single Assignment (GSA) which was proposed in [BMO90] as a part of the Program Dependence Web (PDW) also uses this form of the ϕ function. In this paper, we will use GSA as the intermediate representation for a program.

The rest of the paper describes a uniform symbolic analysis framework that can be applied to the following three problems:

- Deriving symbolic values from symbolic expressions and the program control dependences and determining the relationship between symbolic expressions (Section 2).
- Identifying recurrence and computing the symbolic upper and lower bounds of the value of a recurrence (Section 3).
- Obtaining the symbolic value of array elements used as array subscripts (Section 4).

2 Demand-Driven Symbolic Interpretation

In this section, we first give a brief overview of GSA. Then we present a backward substitution scheme and show how to use the constructs in GSA together with the control dependence to determine the symbolic value of an expression.

2.1 Gated Single Assignment

GSA introduces three pseudo-assignment functions, which are extensions of the ϕ function used in SSA:

- γ function : Replaces those ϕ functions located immediately after an `if` statement. It includes the predicate of the `if` statement as an additional parameter.
- μ function : Replaces ϕ functions at the head of a `do` loop.
- η function : Replaces ϕ functions at the exit of a loop. It selects the last value produced by a μ function.

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

```
    X := 1
L:  do I = 1,N
        if (P) then X := X + 1
    enddo
```

```
    X0 := 1
L:  do I = 1,N {X1 := μ(L, X0, X3)}
        if (P) then X2 := X1 + 1
        X3 = γ(P, X2, X1)
    enddo
    X4 := η(I > N, X1)
```

The γ function in the assignment $\mathtt{X}_3 = \gamma(\mathtt{P}, \mathtt{X}_2, \mathtt{X}_1)$ returns $\mathtt{X}_2$ or $\mathtt{X}_1$, 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, $\mathtt{X}_0$, with the value computed within the loop body, $\mathtt{X}_3$. The loop label L is the first argument of this function. 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

$$\mathtt{X}_4 := \gamma(\mathtt{N} < 1, \mathtt{X}_0, \eta(\mathtt{I} > \mathtt{N}, \mathtt{X}_1))$$

If the loop is a zero-trip loop, then $\mathtt{X}_4$ will take the value $\mathtt{X}_0$ from outside the loop. Otherwise, $\mathtt{X}_4$ will take the value from inside the loop when the loop exit condition is satisfied. In [BMO90], an efficient algorithm is presented to construct GSA from the SSA representation and the control flow graph of a program.

2.2 Backward Substitution and Path Projection

To derive the value for a symbolic variable at a point p in the GSA form of a program, we first perform backward substitution. This process substitutes a variable with its reaching definitions. The result is a symbolic expression, *SE*, where each term is either a constant or a variable name. In the backward substitution, pseudo-assignment functions η, γ, and μ are treated as if they were ordinary functions.

The possible values of a backward-substituted symbolic expression 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 a statement p. If p is control dependent on a collection C of if statements, its *PC* can be defined as an expression involving the predicates in C. The path-restricted value *PV* of a symbolic expression at statement p is the following projection of its symbolic value *SE*:

$$PV = SE(PC) \tag{1}$$

To compute the projection we can use the following rules:

$$\gamma(P, V_t, V_f)(PC) = \begin{cases} V_t(PC) & \text{if } PC \supset P \\ V_f(PC) & \text{if } PC \supset \neg P \\ \gamma(P, V_t(PC), V_f(PC)) & \text{otherwise (unknown)} \end{cases} \tag{2}$$

$$\mu(L, V_{init}, V_{iter})(PC) = \mu(L, V_{init}(PC), V_{iter}(PC)) \tag{3}$$

$$\eta(P, V)(PC) = V(P \wedge PC) \tag{4}$$

We present next some examples of backward substitution and path projection. The examples illustrate how these techniques improve the effectiveness of array privatization [TP93]. 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 or not 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 $\texttt{XE}(1 : \texttt{NDFE}_0)$ defined in loop `D` covers the array region $\texttt{XE}(1 : \texttt{NDDF}_0 * \texttt{NNPED}_0)$ accessed in loop `U`, we need to prove that $\texttt{NDFE}_0 \geq \texttt{NDDF}_0 * \texttt{NNPED}_0$. This is easily done after $\texttt{NDFE}_0$ is replaced by $\texttt{NDDF}_0 * \texttt{NNPED}_0$ using backward substitution. The path condition for those points within loop `U` where `XE` is accessed is $PC_{\texttt{U}} = (\texttt{NDDF}_0 \geq 1 \wedge \texttt{NNPED}_0 \geq 1)$. The path condition at those points where `XE` is defined is $PC_{\texttt{D}} = (\texttt{NDFE}_0 \geq 1)$ or, after the backward substitution, $PC_{\texttt{D}} = (\texttt{NDDF}_0 * \texttt{NNPED}_0 \geq 1)$. It is easy to see that $PC_{\texttt{U}} \supset PC_{\texttt{D}}$ 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.

```
        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
D:              Assign to array region WORK(JLOW2 : JUP2)
U:              if (P) then Use array region WORK(2 : JMAX - 1)
        enddo
```

For the array `WORK` to be private to the loop `L`, we need to determine that the use of $\texttt{WORK}(2, \texttt{JMAX} - 1)$ at `U` is covered by the definition of $\texttt{WORK}(\texttt{JLOW}_2 : \texttt{JUP}_2)$ at `D`. The PC at `U` is `P`. Using projection rule for the γ function under the condition `P`, we have the following replacements which prove the desired condition:

$$\texttt{JLOW}_2 = \gamma(\texttt{P}, \texttt{JLOW}_0, \texttt{JLOW}_1)(\texttt{P})$$

$$
\begin{aligned}
&= \mathtt{JLOW}_0(\mathtt{P}) \\
&= 2 \\
\mathtt{JUP}_2 &= \gamma(\mathtt{P}, \mathtt{JUP}_0, \mathtt{JUP}_1)(\mathtt{P}) \\
&= \mathtt{JUP}_0(\mathtt{P}) \\
&= \mathtt{JMAX} - 1
\end{aligned}
$$

Sometimes in array privatization it is sufficient to know the upper and lower bounds of a symbolic variable. In this case we can ignore the predicate in the γ function and the PC predicate and apply the minimum and maximum functions directly:

$$\max(\gamma(P, V_t, V_f)) = \max(V_t, V_f) \tag{5}$$

$$\min(\gamma(P, V_t, V_f)) = \min(V_t, V_f) \tag{6}$$

Using these two rules in the example above, we obtain:

$$
\begin{aligned}
\mathtt{max}(\mathtt{JLOW}_2) &= \mathtt{max}(\gamma(\mathtt{P}, \mathtt{JLOW}_0, \mathtt{JLOW}_1)) \\
&= \mathtt{max}(\mathtt{JLOW}_0, \mathtt{JLOW}_1) \\
&= \mathtt{max}(2, 1) \\
&= 2 \\
\mathtt{min}(\mathtt{JUP}_2) &= \mathtt{min}(\gamma(\mathtt{P}, \mathtt{JUP}_0, \mathtt{JUP}_1)) \\
&= \mathtt{min}(\mathtt{JUP}_0, \mathtt{JUP}_1) \\
&= \mathtt{min}(\mathtt{JMAX} - 1, \mathtt{JMAX}) \\
&= \mathtt{JMAX} - 1
\end{aligned}
$$

which also prove the desired condition. The problem of determining whether $PC \supset P$ is NP-Complete [GJ79]. However, because the number of boolean variables in a program is usually small, we expect the cost of computing the path projection to be small. The upper and lower bound computations can be used to obtain the range of a variable when the number of boolean variables in $PC \supset P$ is large.

The path condition may be used to derive information about symbolic values. For instance, consider the following loop:

```
    do I = 1, N
R:     X(I) := X(I + N)
S:     if (I > L) goto U
T:     ...
U:     ...
    enddo
```

At statement R, the PC is $1 \leq \mathtt{I} \leq \mathtt{N}$. This makes it possible to determine that there is no data dependence between the instances of R. The PC at statement T is $(\mathtt{I} \leq \mathtt{L}) \wedge (1 \leq \mathtt{I} \leq \mathtt{N})$. It tells us more about the possible values of I at T. Incorporating path condition in the analysis provides us with more power than GSA or SSA alone.

2.3 Comparison of Symbolic Expressions

The symbolic expression may still contain γ functions after path projection. In symbolic analysis, we sometimes need to compare these expressions. [AWZ88] defines a *congruence*

relation between expressions with ϕ assignments. The congruent variables are shown to have equivalent values under structural isomorphism. 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 call two expressions *compatible* if, after backward substitution, the non-constant terms 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:

$$(E^1 > \gamma(P, V_t, V_f)) \equiv (E^1 > V_t) \wedge (E^1 > V_f) \tag{7}$$

   If V_t, V_f contain 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 we discussed before. Because E^1 does not contain any pseudo-function, it can be proven that:

$$(E^1 > E^2) \equiv (E^1 > \max(E^2)) \tag{8}$$

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:

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

   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:

$$(\gamma(P, V_t', V_f')(P) > V_t) \wedge (\gamma(P, V_t', V_f')(\neg P) > V_f) = (V_t' > V_t) \wedge (V_f' > V_f)$$

   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 are different. Using the distribution rules, we can identify those equalities.

The rules discussed above also apply 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.

2.4 Ordering of Backward Substitution

We conclude this section with a technique for ordering backward substitution to obtain compatible expressions. As we mentioned before, it is not necessary to fully expand two expressions to the point that they contain only constants and program inputs. It is more efficient to stop expanding as soon as two expressions become compatible. If reaching definitions are substituted in the order they appear in the program's dominator tree, then it is possible to avoid expanding an expression beyond what is necessary for the comparison. The algorithm is shown below.

Algorithm Unify
Input: expressions s and t
1. Mark constants and matching identifiers in s, t as dead.
2. while $\exists$ *active identifier* $\in s, t$ do
   Expand the identifier whose reaching definition is the lowest in the dominator tree.
   Mark dead constants and matching identifiers as dead.

To summarize, the backward substitution scheme has several advantages over forward substitution. First, it is goal directed. In contrast, the traditional forward substitution will blindly propagate all the information, even though most of it will never be used. Second, it can stop expanding when all the variables match. With backward substitution, it is seldom necessary to expand the expression in terms of the program inputs. Finally, in a forward propagation scheme, everything must start from the most primitive variables and, in the case of several levels of conditionals, the number of branches may quickly explode and complexity may grow out of control. Since the backward tracing is goal directed and incremental, we can easily set complexity constraint such as the maximum level of unwinding. After that, the algorithm gives up and tries to obtain an approximate solution by computing maxima and minima.

3 Recurrences 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 rule for 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 {J1 := μ(L, J0, J3)}
       if (P) then J2 := J1 + A
       J3 := γ(P, J2, J1)
```

```
    enddo
```

Substituting terms in the μ function leads to:

$$\begin{aligned} J_1 &= \mu(L, J_0, L_3) \\ &= \mu(L, J_0, \gamma(P, J_2, J_1)) \\ &= \mu(L, J_0, \gamma(P, J_1 + A, J_1)) \end{aligned}$$

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

$$\lambda i.(J_1(i)) \equiv \gamma(i = 1, J_0, \gamma(P, J_1(i-1) + A, J_1(i-1)))$$

After identifying the recurrence, we could use pattern matching to identify induction and reduction variables. For instance, if P is always true and A is loop invariant, then $J_1(i)$ is an induction variable with value $J_0 + (i-1) * A$. When P is always true and A is a linear reference to an array, for example X(i), $J_1(i)$ is a reduction sum over X. For induction variable identification, this approach is equivalent to the Strong Connected Region (SCR) algorithm given by [Wol92]. However, the symbolic substitution scheme is more general because it can deal with those 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 $J_1(i)$ is either $J_0 + (i-1) * A$ or 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:

$$\begin{aligned} \max(J_1) &= \max(\mu(L, J_0, \gamma(P, J_1 + A, J_1))) \\ &= \mu(L, J_0, \max(\gamma(P, J_1 + A, J_1))) \\ &= \mu(L, J_0, J_1 + A) \\ \min(J_1) &= \min(\mu(L, J_0, \gamma(P, J_1 + A, J_1))) \\ &= \mu(L, J_0, \min(\gamma(P, J_1 + A, J_1))) \\ &= \mu(L, J_0, J_1) \end{aligned}$$

The resulting two recurrence functions can now be solved to obtain the upper bound $J_0 + N * A$ and the lower bound J_0.

4 Index Arrays

The use of array elements as subscripts makes dependence analysis and array privatization more difficult than when just scalars are used. When the value of an index array depends on the program input, run-time analysis has to be used. Sometimes when an index array is assigned with a symbolic expression, its value 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 compiler time that the array element JPLUS(J) has value of J + 1 for J $\in$ [1, 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 $\texttt{JPLUS}_1 = \alpha(\texttt{JPLUS}_0, \texttt{I}, \texttt{exp})$. The semantics of the α function is that $\texttt{JPLUS}_1(\texttt{I})$ receives the value of exp while the other elements of $\texttt{JPLUS}_1$ will take the values of the corresponding elements of $\texttt{JPLUS}_0$. 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 {JPLUS2 := μ(L, JPLUS0, JPLUS1)}
        JPLUS1 := α(JPLUS2, J, J + 1)
    enddo
    JPLUS3 := α(JPLUS2, JMAX, Q)
```

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

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

In the above symbolic evaluation process, an expression of the form $\alpha(\texttt{X}, \texttt{i}, \texttt{exp})(\texttt{j})$ is evaluated to $\gamma(\texttt{j} = \texttt{i}, \texttt{exp}, \texttt{X}(\texttt{j}))$. An expression of the form $\mu(\texttt{L}, \texttt{Y}, \texttt{Z})(\texttt{j})$ 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, SSA conversion is done only on those arrays that are used as subscripts.

5 Conclusion and Implementation

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 that is not used in the end, and propagates too little information for the few important variables in the analysis.

We proposed in this paper a technique to derive information about symbolic variables in a demand-driven way that is both efficient and accurate. In contrast to global forward substitution, the demand-driven technique introduces no overhead when there is no need for

the propagation and resolves complicated symbolic expressions on demand. We illustrated the use of this technique in array privatization to determine the symbolic values of array reference regions.

The technique in this paper has been implemented in the POLARIS restructuring compiler. Preliminary experience shows that it is both effective and efficient. It can often meet the symbolic analysis requirement of array privatization and dependence analysis.

References

[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. Technical Report 1345, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., April 1994.

[BE94b] William Blume and Rudolf Eigenmann. Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Ben chmarks. 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. Technical Report 1375, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, June 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.

[CF87] Ron Cytron and Jeanne Ferante. What's in a Name? or The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proc. 1987 International Conf. on Parallel Processing*, pages 19–27, August 1987.

[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.

[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.

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

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

[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.

[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.

[Wol92] Michael Wolfe. Beyond induction variables. *ACM PLDI'92*, 1992.