Efficient Building and Placing of Gating Functions *

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 (217) 333-6884

1 Introduction

The Gated Single-Assignment (GSA) program representation is an extension of the Static Single Assignment (SSA) representation[CFR+91]. GSA was introduced by Ballance, Maccabe and Ottenstein as a part of Program Dependence Web (PDW)[BMO90]. It is a convenient representation for several program analysis and optimization techniques, including constant propagation with conditional branches[WZ91]; equality of symbolic expressions[AWZ88, Hav93]; induction variable substitution[Wol92]; symbolic dependence analysis [BE94] and demand-driven symbolic analysis for array privatization[TP94, TP93]. In the SSA representation, $\phi-$*functions* of a single type are placed at the confluence nodes of a program flow graph to represent different definitions of a variable reaching from different incoming edges. The condition under which a definition reachs a confluence node is not represented in the $\phi-$function. By contrast, in the GSA representation, several types of *gating functions* are defined to represent the different condition classes at different confluence nodes. Some extra parameters are introduced in the gating functions to represent the conditions. In this paper, we present an almost linear time algorithm to construct the GSA. The new algorithm is more efficient and simpler than the existing algorithms for GSA construction [BMO90, Hav93]. Since SSA is a special case of GSA, it can also be used as an efficient alternative algorithm for SSA construction.

The existing algorithms for building the GSA follow two steps. The first step is the same $\phi-$function placement procedure as in the SSA construction[CFR+91]. In the second step, the GSA conversion algorithms collect the control dependences of the definitions reaching a $\phi-$function and transforms the $\phi-$function into a gating function. The original GSA conversion algorithm[BMO90] assumed a Program Dependence Graph (PDG)[FOW87] as its initial representation. Havlak developed another algorithm[Hav93] to construct a variant of the GSA, known as Thinned GSA. Because it starts with the program flow graph, it is, therefore, somewhat simpler. For each $\phi-$function, both algorithms traverse the control flow graph to find the gating conditions for each reaching definition. To convert a $\phi-$function to a gating function, $O(E)$ edges may be visited (where E is the number of edges in the flow graph). Since the number of $\phi-$functions in the program is $O(N)$ (where N is the number of nodes in the program), and the same edge may be visited for every $\phi-$function, the time complexity of these algorithms is $O(E \times N)$.

The algorithm in this paper constructs and places the gating functions from a program control flow graph in a single step. In our algorithm, SSA and GSA constructions are unified under a single process of *gating path* construction. It uses the *path compression* technique[Tar79] to reduce the total number of visits to the edges in the flow graph. Tarjan describes two ways to implement the path compression. A simple method has an $O(E \log(N))$ time bound; a sophisticated off-line algorithm maintaining balanced subtrees has an $O(E\alpha(E,N))$ time bound. Yet another on-line $O(E\alpha(E,N))$

*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.

method, called *stratified path compression* by Farrow [Far77, Tar79], can also be used. This GSA algorithm is also almost as efficient as the best known algorithms for $\phi-$function placement in SSA conversion.

The rest of the paper is divided into the following sections. In Section 2, we introduce some background and notations. In Section 3, we define the notation of *gating path* and discuss its relation to the *dominance frontiers* in the context of $\phi-$placement. Our gating path based GSA construction algorithm is presented in Section 4. In section 5, we work through an example. Section 6 shows some timing results using the simple path compression implementation. Some conclusions are presented in Section 7.

2 Background and Notations

Representing data flow and control flow properties of programs is an important issue in optimizing compilers for accurate and efficient transformations. SSA form has been shown to be useful in capturing the data flow information required by some important program optimizations[AWZ88, RWZ88]. In the SSA form, each definition of a variable is given a unique new name, and each use of a variable is renamed to refer to a single reaching definition. When several definitions of a variable, $a_1, a_2, ..., a_m$, reach a confluence node in program control flow graph, a $\phi-$function assignment statement, $a_n = \phi(a_1, a_2, ..., a_m)$, is created to merge them into a single new definition a_n. Hence, in the SSA representation every use of a variable except those in a $\phi-$function has only one reaching definition that is identified by a unique variable name. SSA captures the data flow information (def-use chains) of a program in a compact form.

An efficient algorithm for constructing SSA with a minimal number of $\phi-$functions was originally designed by Cytron, Ferrante, Rosen, Wegman and Zadeck[CFR+91]. The algorithm for placing the $\phi-$functions is $O(N^2)$ in the worst case, but often appears to be linear when applied to real programs. Johnson and Pingali[JP93, JPP94] proposed another algorithm to place $\phi-$functions in $O(E)$ time. Later, Cytron and Ferrante proposed an almost linear time $O(E\alpha(E))$ using path compression. Recently, Sreedhar and Gao[SG94] have developed another $O(E)$ time algorithm. We should point out that although our algorithm also uses the path compression technique, our problem is more complicated and our approach is completely different from Cytron and Ferrante's.

Whereas a $\phi-$function represents the merge of multiple reaching definitions, it does not contain the condition that specifies which reaching definition will be the value of the function. Gating functions were introduced by Ballance, Maccabe and Ottenstein[BMO90] to capture the control conditions that guard the paths to a $\phi-$function. There are three types of *gating function*:

- The γ function, which is an $if-then-else$ construct, captures the condition for each definition to reach a confluence node. For instance, $X_3 = \gamma(B, X_1, X_2)$ represents $X_3 = X_1$ if B and X_2 if $\neg B$.
- The μ function, which only appears at loop headers, selects the initial and loop-carried values. For instance, $X_2 = \mu(X_0, X_3)$ represents that X_2's initial value is X_0 and its subsequent value is X_3.
- The η function determines the value of a variable at the exit of the end of the loop.

In [BMO90], the conversion to GSA is done after $\phi-$placement. The algorithm works by expanding each ϕ node into a GSA gating tree that contains the control information for the different reaching definitions. The translation for each ϕ node may potentially scan all the edges in the flow graph. Therefore, $O(E)$ is its worst case time complexity. The translation for all the ϕ nodes is $O(E \times N)$ because there can be $O(N)$ confluence nodes in a program. Paul Havlak introduced a variant of GSA, called Thinned GSA, and an algorithm for ϕ translation that is similar to the original GSA algorithm with the same complexity. Both algorithms start from the ϕ nodes after the SSA $\phi-$placement. Using the control dependence graph, they extract the related control information by walking through the flow graph along the paths that connect the definitions and the $\phi-$function.

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

A node v *dominates* another node w, denoted as $v \geqslant w$, if every path from **Entry** to w contains v. A node u *postdominates* another node w if every path from w to **Exit** contains u. Node v *strictly* dominates w, denoted as $v \gg w$, if v dominates w and $v \neq w$. Node v is the *immediate dominator* of w, denoted as $v = idom(w)$, if v dominates w and every other dominator of w dominates v. Every node in a flow graph except **Entry** has a unique immediate dominator. The edges $\{idom(w) \rightarrow w | w \in N - \{Entry\}\}$ form a *dominator tree* such that v dominates w if and only if v is a proper ancestor of w in the dominator tree (where the word *proper* means $v \neq w$). The *postdominator tree* is defined similarly using the post-dominating relation. In the rest of the paper, the words *predecessor*, *successor*, and *path* refer to the flow graph, and the words *parent*, *child*, *ancestor*, and *descendant* refer to the dominator tree.

The *dominance frontier*[CFR+91] $DF(X)$ of a CFG node X is the set of nodes $Y \in CFG$ such that X dominates a predecessor P of Y but does not strictly dominate Y:

$$DF(X) = \{Y | (\exists P \rightarrow Y)(X \geqslant P \text{ and } X \not\gg Y)\}.$$

Given a set φ of the CFG nodes, the set $DF(\varphi)$ is the union of the dominance frontiers defined by each node in φ:

$$DF(\varphi) = \cup_{X \in \varphi} DF(X).$$

The *iterated dominance frontier* $DF^+(X)$ is the transitive closure of $DF(X)$:

$$DF^1(X) = DF(X);$$

$$DF^i(X) = DF(X \cup DF^{i-1}(X)).$$

The iterated dominance frontier $DF^+(\varphi)$ for a set of nodes φ is defined as a union of individual iterated dominance frontiers.

A fundamental result proven in [CFR+91] states that if φ is the set of assignment nodes for a variable V, then $DF^+(\varphi)$ is the minimum set of nodes that need $\phi-$function assignment nodes for V.

3 Gating Paths and Φ-Function Nodes

In this section, we present another way to determine the set of nodes that need $\phi-$function assignment nodes. We prove that the $phi-$nodes computed are the same as those using the iterated dominance frontier algorithm in [CFR+91]. This provides a way to look at the problem from a different perspective.

Definition 1. Given a control flow graph CFG, a *gating path* for a node v is a path in the CFG from $idom(v)$ to v containing only the proper descendants of $idom(v)$ in the dominator tree as intermediate nodes.

In other words, a gating path is a path from $idom(v)$ to v in the CFG such that every node in the path is dominated by $idom(v)$.

In the rest of this section, we prove a theorem that relates the gating paths to the placement of ϕ-functions.

The Lemma 1 and its Corollary establish the existence of gating paths in CFG.

Lemma 1. For any path $d \xrightarrow{+} v$ in CFG, if $d \gg v$ and d only occurs once in the path, d must dominate every node in the path.

Proof. If there is a node u in the path such that u is not dominated by d, i.e., $d \xrightarrow{+} u \xrightarrow{*} v$, $d \not\gg u$, then the path **Entry** $\xrightarrow{*} u \xrightarrow{*} v$ avoids the d. This is a contradiction of $d \gg v$.

Corollary 1. For every node v, there is a gating path from $idom(v)$ to v.

Proof. Since $idom(v) \gg v$, there is a path P from $idom(v)$ to v. Removing all the $idom(v) \xrightarrow{*} idom(v)$ cycles, we obtain a new path from $idom(v)$ to v where $idom(v)$ only occurs once. By Lemma 1, the $idom(v)$ dominates every node in the new path. Hence the new path is a gating path.

it is a gating path.

Lemma 2 and Lemma 3 establish the fact that dominance frontier relations only exist among the nodes in the sibling subtrees under a common parent in the dominator tree of CFG. That is, if $v \in DF^+(X)$, then $idom(v) \gg X$ and, therefore, $idom(v)$ is a proper ancestor of X in the dominator tree. Because $X \not\gg v$, X and v must belong to different subtrees under $idom(v)$.

Lemma 2. If $v \in DF(X)$, then $idom(v) \gg X$.

Proof. From the definition of $DF(X)$, $X \neq idom(v)$. If $idom(v) \not\gg X$, then there is a path from **Entry** to X that avoids the $idom(v)$. Because $v \in DF(X)$, there is a $w \in Pred(v)$, such that $X \gtrsim w$, but $X \not\gg v$. Let $X \xrightarrow{*} w \rightarrow v$ be a path from X to v. Removing the $X \xrightarrow{*} X$ cycles in the path, we obtain a path such that X appears only once. The $idom(v)$ must not be in the subpath $X \xrightarrow{*} w$; otherwise, by Lemma 1, $X \gg idom(v)$ and, therefore, $X \gg v$. Concatenating this path with the $idom(v)$ avoiding path from **Entry** to X, we obtain a path from **Entry** to v which avoids $idom(v)$. This is a contradiction. Therefore, we have proven $idom(v) \gg X$.

Lemma 3. If $v \in DF^+(X)$, then $idom(v) \gg X$.

Proof. By applying Lemma 2, if $v \in DF^1(X)$, then $idom(v) \gg X$. Assume it is true for $v \in DF^{i-1}$. By induction, for $v \in DF^i(X)$, let $v \in DF(u), u \in DF^{i-1}(X)$. We have $idom(u) \gg X$ by induction premises and $idom(v) \gg u$ by Lemma 1. Because $idom(v) \gg u$ implies $idom(v) \gtrsim idom(u)$, we obtain $idom(v) \gg X$.

The Lemma 4-6 relates the gating paths to iterated dominance frontiers.

Lemma 4. If $v \in DF(X)$, then there is a gating path from $idom(v)$ to v passing through X.

Proof. From the definition of $DF(X)$ and Corollary 1, there is a gating path $X \xrightarrow{*} w$ where $w \in Pred(v)$. From Lemma 1 and Lemma 2, we have $idom(v) \gg X$ and, hence, a gating path from $idom(v)$ to X. Because the path $X \xrightarrow{*} w$ contains only the proper descendants of X and, hence, of $idom(v)$, concatenating the paths results in a gating path $idom(v) \xrightarrow{+} X \xrightarrow{*} w \rightarrow v$ from $idom(v)$ to v passing through X.

Lemma 5. If $v \in DF^+(X)$, then there is a gating path from $idom(v)$ to v passing through X.

Proof. Immediate by induction from Lemma 3 and Lemma 4.

Lemma 6. If there is a gating path from $idom(v)$ through X to v where $idom(v) \neq X$, then $v \in DF^+(X)$.

Proof. We prove this by induction on the number of confluence nodes on the subpath $X \xrightarrow{*} w \rightarrow v$. Since there is a gating path from $idom(v)$ through X to v and $idom(v) \neq X$, v must be a confluence node. Otherwise, $w \neq idom(v)$ would be v's immediate dominator. Let the number of confluence nodes on the subpath $X \xrightarrow{+} v$ be n.

1. If $n = 1$, v is the only confluence node. In the path from X to v, $X \xrightarrow{*} w \rightarrow v$, every intermediate node can have only one predecessor in the flow graph. Hence $X \gtrsim w$. Because $idom(v) \neq X$ and $idom(v) \gg X$, $X \not\gg v$. Hence $v \in DF(X)$.

2. Assume the Lemma is true for $n < i$. For $n = i$, let u be the second to last confluence node in $X \xrightarrow{*} w \rightarrow v$, and the path $P_u = idom(u) \xrightarrow{+} u$ be the subpath from $idom(u)$ to u. If X is in P_u and $X \neq idom(u)$, then P_u is a gating path for u passing through X with $n = k < i$. Therefore, $u \in DF^{k<i}(X)$. Moreover, since there is only one non-confluence node from u to v, $v \in DF(u) = DF^{k+1}(X)$. If $X = idom(u)$ or X is not on P_u, then $X \gtrsim idom(u) \gg u \gtrsim w$ where w is the second to last node on $X \xrightarrow{*} idom(u) \xrightarrow{*} u \xrightarrow{*} w \rightarrow v$. Hence $v \in DF(X)$. This proves the Lemma.

Using the results from Lemma 5 and Lemma 6, we obtain the following Theorem for determining if a node v is in $DF^+(\varphi)$.

Theorem 1. Given an initial set φ of CFG nodes, for any node v in CFG, $v \in DF^+(\varphi)$ if and only if there is a gating path from $idom(v)$ to v containing a node that belongs to φ (i.e., there is a gating path $idom(v) \xrightarrow{+} X \xrightarrow{+} v$ where $X \in \varphi$).

Proof. Immediate from Lemma 5, Lemma 6, and the fact that $DF^+(\varphi) = \cup_{X \in \varphi} DF^+(X)$.

Applying theorem 1, if we can compute a gating path expression for each node v in a CFG and determine if v has a gating path that contains nodes in φ, we can determine if v needs a $\phi-$function assignment. In the next section, we present an algorithm which builds a gating path expression of v in such a way that it is exactly the gating function for $\phi-$conversion if $v \in DF^+(\varphi)$.

4 Algorithm for GSA Construction

Given a $CFG(N, E)$, we can treat any path in the CFG as a string of edges in E, but not all such strings over E are paths in CFG. A *path expression* [Tar81b] P of *type* (u, v) is a simple regular expression over E such that every string in $\sigma(P)$ is a path from node u to node v (where $\sigma(P)$ represents the string generated by the regular expression P). Every subexpression of a path expression is also a path expression whose type can be determined as follows.

Let P be a path expression of type (u, v).

- If $P = P_1 \cup P_2$, then P_1 and P_2 are path expressions of type (u, v).
- If $P = P_1 \cdot P_2$, then there exists a unique node w such that P_1 is a path expression of type (u, w) and P_2 is a path expression of type (w, v).
- If $P = P_1^*$, then $u = v$ and P_1 is a path expression of type $(u, v) = (u, u)$.

For instance, in the following statements

if (B) **then**
    Block1
else
    Block2
endif

there are two paths from the **if** node to the **endif** node represented by path expressions: $p_t = (if\ (B) \xrightarrow{t} Block1 \rightarrow endif)$ and $p_f = (if\ (B) \xrightarrow{f} Block2 \rightarrow endif)$. The notation $(a \rightarrow b \rightarrow c)$ represents a path of two edges from a to b and from b to c. To simplify the discussion, we assume the **endif** is the entry of a basic block. The union of the two path expressions, $(p_t \cup p_f)$, is of type $(if\ (B), endif)$ and represents all paths from the **if (B)** to the **endif**.

4.1 Path Expressions Represented as Gating Functions

Different paths reaching a $\phi-$function node are represented by path expressions. Our strategy is to define the symbols used to represent the edges such that a path expression takes the same form as a gating function. Only the outgoing edges from conditional statements (or conditional edges) are necessary to unambiguously represent a path. We will represent paths using a form of gating functions containing only the conditional edges. However, in the process of building such gating functions, we also need to use the unconditional edges. We use a *white space symbol* Λ to represent an unconditional edge. For example, each of $P(Block1, endif) = \Lambda$ and $P(Block2, endif) = \Lambda$ represents an unconditional edge.

We now define the gating symbols for the edges of branch statements. A branch statement like $if(B)$ has two outgoing edges. Let's call them B_t and B_f. To build the gating function for a path, we use the gating function notation to represent the edges. The B_t edge is represented by the expression $\gamma(B, \Lambda, \emptyset)$, and the B_f edge is represented by the expression $\gamma(B, \emptyset, \Lambda)$. Here we

use another white space symbol $\emptyset$ to represent a branch that is not taken. It is easy to extend the notation to statement types with more than two outgoing branches. In summary, a path expression is represented as a gating function using the following symbols:

- A white space symbol Λ represents an unconditional edge.
- A white space symbol $\emptyset$represents an edge not taken at a branch node.
- A γ expression $\gamma(P, e_1, e_2, ..., e_n)$ where only one e_i is Λ and all the other e's are $\emptyset$represents the i's edge from an n-way branch statement with condition P.

Given a gating expression R, the following equations define the properties of the symbols that can be applied to simplify R.

$R = R_1 \cup R_2$: **case** $R_1 == \emptyset$
  return R_2
 case $R_2 == \emptyset$
  return R_1
 case $R_1 == \gamma(B, R_{1_t}, R_{1_f})$ **and** $R_2 == \gamma(B, R_{2_t}, R_{2_f})$
  return $\gamma(B, (R_{1_t} \cup R_{2_t}), (R_{1_f} \cup R_{2_f}))$

$R = R_1 \cdot R_2$: **case** $(R_1 == \emptyset)$ **or** $(R_2 == \emptyset)$
  return $\emptyset$
 case $R_1 == \Lambda$
  return R_2
 case $R_2 == \Lambda$
  return R_1
 case $R_1 == \gamma(B, R_{1_t}, R_{1_f})$
  return $\gamma(B, (R_{1_t} \cdot R_2), (R_{1_f} \cdot R_2))$

Note that in the case where $R = R_1 \cup R_2$ and $R_1 = \gamma(B, R_{1_t}, R_{1_f})$, R_1 and R_2 must have the same type. That is, R_2 must have the same starting node as R_1. Therefore, R_2 must be in the form of $\gamma(B, R_{2_t}, R_{2_f})$.

Back to the above example, we can obtain

$$p_t(if, endif) = \gamma(B, \Lambda, \emptyset) \cdot \Lambda = \gamma(B, \Lambda, \emptyset);$$

$$p_f(if, endif) = \gamma(B, \emptyset, \Lambda) \cdot \Lambda = \gamma(B, \emptyset, \Lambda);$$

$$P(if, endif) = p_t(if, endif) \cup p_f(if, endif) = \gamma(B, \Lambda, \emptyset) \cup \gamma(B, \emptyset, \Lambda) = \gamma(B, \Lambda, \Lambda).$$

Applying Cytron et al.'s renaming procedure[CFR+91] to insert variable names into a gating function $R(u, v) = \gamma(B, R_t, R_f)$, we need to know from which predecessors of v that each of R_t and R_f reach v. This is done by labeling each path with the predecessor number of v. If $R = R_1 \cup R_2$, and the path R_1 enters v from the ith predecessor of v, then R_1 is labeled with i. This is done by labeling all the Λ in R_1 with a superscript i. Hence,

$$P(if, endif) = p_t^1(if, endif) \cup p_f^2(if, endif) = \gamma(B, \Lambda^1, \emptyset) \cup \gamma(B, \emptyset, \Lambda^2) = \gamma(B, \Lambda^1, \Lambda^2).$$

The superscript label can be used in the renaming procedure to determine which parameter in a gating function should be substituted by the name on the top of the stack for a variable. For further details of this procedure, interested readers should refer to [CFR+91]. Consider, for example, that $Block1$ contains an assignment to a variable A, which after renaming becomes A_{B1} and that $Block2$ has no assignment to A. Let the definition of A reaching the if statement be A_{Orig}. Then the gating function for A takes on the form:

$$A_{New} = \gamma(B, \Lambda^1, \Lambda^2).$$

In the renaming process, the reaching definition to v from predecessor 1 is A_{B1}, and from predecessor 2 is A_{Orig}. Hence, Λ^1 in the gating function is replaced by A_{B1} and Λ^2 is replaced by A_{Orig}. The gating function

$$A_{New} = \gamma(B, A_{B1}, A_{Orig})$$

correctly reflects the reaching definitions from different paths.

4.2 Algorithm for Building Gating Functions

In the previous section, we defined a representation and operations on gating paths that lead to gating functions. In this section, we present an algorithm to construct the gating path expression and, therefore, the gating function for each node. Our algorithm assumes that a CFG and its dominator tree DT are given. We also assume that each node in the CFG is assigned a *depth-first number dfn*. The dominator tree can be computed in $O(E\alpha(E,N))$ time using the dominator algorithm of Lengauer and Tarjan[LT79], or in $O(E)$ time using a more complicated algorithm of Harel[Har85]. The dfn can be computed in linear time[ASU86]. The *dfn* number has the property of $dfn(idom(v)) < dfn(v)$ for each node $v \neq$ **Entry**.

Each *loop* step of the algorithm processes a set of sibling nodes with a common parent u. The outer loop processes nodes in reverse *dfn* sequence. Since $dfn(idom(v)) < dfn(v)$ all the siblings of v and their descendants must have already been visited by the outer loop before $idom(v)$ is visited. In the *derive* phase, the algorithm processes the set of siblings $children(u)$. Depending on the class to which an edge $e = (w, v) \in E$ belongs, one of the following step is taken

1. If e comes directly from v's immediate dominator (i.e. $w = u = idom(v)$), then the edge itself is a gating path from $idom(v)$ to v. The algorithm updates the gating path of $GP(v)$ with the union of e and the old $GP(v)$.

2. If e comes from a node that is a descendant of a sibling of v (i.e. $w \neq u$ and $v \not\gg w$), the algorithm calls $EVAL(e)$ to compute a path expression $p(subroot(w), v)$. The $subroot(w)$ is the root of the subtree to which w belongs (i.e. the sibling of v dominating w). The path expression $p(subroot(w), v)$ represents all paths from $p(subroot(w))$ to v which end with edge e and contain only the proper descendants of $subroot(w)$ as intermediate nodes. $EVAL(e)$ also returns **true** if one of the nodes along the path belongs to φ, indicating by Theorem 1 that v is a $phi-$function node. The gating expression returned from $EVAL(e)$ is added to $ListP(v)$, which is a list containing all the paths starting from the siblings of v to v.

3. If e comes from a node which is a descendant of v (i.e. $w \neq u$ and $v \geqslant w$), then it forms a loop with header v. The algorithm works exactly as in step 2 above. The the path expression computed by $EVAL(e)$ is later used to update $G^*(v)$ to represent the path from the loop back edge e. $G^*(v)$ is used to build a μ function for the loop.

Using Tarjan's technique for operations on a forest[Tar79], we define the following operations on the forest of subtrees in a dominator tree:

- $EVAL(e)$: Let $e = (w, v)$. If $r = subroot(w) \rightarrow w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_k = w$ is the tree path from the root of tree containing w to w, then $EVAL$ returns a path expression representing $(R(r) \cdot R(w_1) \cdot R(w_2) \cdot \ldots \cdot R(w) \cdot e)$ with each Λ superscripted by the predecessor number of e to v. It also returns the value of $(\Phi(r) \vee \varphi(r) \vee \Phi(w_1) \vee \varphi(w_1) \ldots \vee \Phi(w) \vee \varphi(w))$ indicating whether there is a node in the path that belongs to φ. In this expression, $\vee$ represents the logic *or* operation. $\Phi(x)$ is *true* if node x needs a $phi-$function, and $\varphi(x)$ is *true* if $x \in \varphi$. In the process, $EVAL(e)$ performs path compression which updates the R's for the intermediate nodes and relinks them directly to r.

- $LINK(u, v)$: If u and v are roots, combine the trees with roots u and v by making u the parent of v. $LINK$ may also adjust the tree and the R's to construct a balanced tree for the almost linear time algorithm.

Algorithm: Building gating path expression

```
input:          The assignment nodes φ
initialize:     foreach v ∈ N do
                    Φ(v) ← false
                    GP(v) ← Ø
                    G*(v) ← Ø
                    ListP(v) ← Ø
                    if v ∈ φ
                        φ(v) ← true
                    fi
                od
loop:           foreach u ∈ N in reverse dfn do
derive:             foreach v ∈ children(u) do
                        foreach e = (w, v) ∈ E do
idom node:                  if w == u then
                                GP(v) ← GP(v) ∪ (e)
sibling node:               else
                                (φ, p(subroot(w), v)) ← EVAL(e)
                                Φ(v) ← Φ(v) or φ
                                ListP(v) ← ListP(v) with p(subroot(w), v)
                                (* The with operator insert an element to a list *)
                            fi
                        od
                    od
sequence:           Topsort(children(u))
merge:              foreach v ∈ children(u) in Toporder do
                        foreach p(subroot(w), v) ∈ ListP(v) do
mu entry:                   if subroot(w) == v then
                                G*(v) ← G*(v) ∪ p(subroot(w), v)
gamma:                      else
                                GP(v) ← GP(v) ∪ (GP(subroot(w)) · p(subroot(w), v))
                                Φ(v) ← Φ(v) or Φ(subroot(w))
                            fi
update:                     UPDATE(v, GP(v))
                            LINK(u, v)
                        od
                    od
                od
```

- $UPDATE(v, P)$: If v is a root, assign $R(v)$ to P if v is not the immediate post-dominator of $idom(u)$; otherwise, assign $R(v)$ to Λ.

If the CFG is reducible, then the paths between the sibling trees is cycle free. We can then obtain a topsort order among the siblings in $children(u)$. Irreducible graphs can also be handled by computing a path sequence for each dominator strong components. Due to limited space, we cannot detail how to compute the path sequence. Interested readers should refer to [Tar81a]. The two existing GSA algorithms only handle reducible graph, but the algorithm here can be extended to handle irreducible graph. In the merge phase, the algorithm follows the topsort order and computes for each child of u a path expression $GP(v)$ representing all the gating paths of v. In the processes, if there is a $p(v, v)$ in $PList(v)$, indicating that v is a loop header, step *mu entry* is executed. $G^*(v)$ is used to construct a μ function for the loop header v.

The algorithm completes the processing of the sibling set by executing $UPDATE(v, GP(v))$ and $LINK(u, v)$ for each child v of u. The $LINK(u, v)$ operation is straightforward. Two path expressions are stored at each node v. $GP(v)$ represents the gating path from $idom(v)$ to v. It is the gating function placed at v if $\Phi(v)$ is **true**. $R(v)$ represents the path expression from the *parent* of v to v in the tree where v currently belongs. $R(v)$ is used by $EVAL$ to compute the path expression from v to the root of tree where v currently belongs. $R(v)$ may be changed by path compression. The $UPDATE(v, GP(v))$ is slightly different from the Tarjan's algorithm. Here, it contains an optimization step to simplify subsequent path expressions. $UPDATE(v, GP(v))$ sets the value of the label $R(v)$ to $GP(v)$. If a v is the immediate post-dominator of $idom(v)$, then any path from $idom(v)$ unconditionally passes through v. Therefore, we can set $R(v)$ to Λ in the $UPDATE$ operation to represent the unconditional reach and, in this way, to simplify the path expressions for subsequent calls to $EVAL$.

This algorithm is a variant of Tarjan's fast algorithm for solving path problems using *dominator strong components* decomposition [Tar81a]. Its correctness can be derived from the following Lemma, which we quote without proof here. We will work through an example to illustrate the algorithm.

Lemma 7.[Tar81a]

- For edges $e = (w, v)$ in CFG such that $w \neq u$, the corresponding path expression in the $ListP(v)$ computed by the *derive* phase is an unambiguous path expression representing exactly the paths from $subroot(w)$ to v that end with e and contain only proper descendants of $subroot(w)$ as intermediate nodes.

- For each node v in CFG, $GP(v)$ as computed by the algorithm is an unambiguous path expression representing exactly the paths from $idom(v)$ to v that contain only proper descendants of $idom(v)$ as intermediate nodes.

Lemma 7 in conjunction with Theorem 1 prove that the algorithm correctly builds and inserts the gating function for a CFG.

Theorem 2. The gating path expression algorithm builds and inserts the gating functions correctly.

The algorithm requires N time on the step *initialize*; $N - 1$ calls on $UPDATE$; $N - 1$ calls on $LINK$; E calls on $EVAL$. The top sort at each major loop iteration sorts on disjoined subsets. For the whole algorithm, the topsort time is the summation of individual subset sizes, which is $O(N)$. The merge step takes $O(N)$. Hence, the time complexity is $O(E\alpha(E, N))$ if the stratified path compression is used to implement the forest operations, and $O(E \log N)$ if path compression is used. Because the sequence of $EVAL$ and $LINK$ can be easily determined beforehand, the off-line algorithm in [Tar79] can also be used to achieve $O(E\alpha(E, N))$ time complexity.

Theorem 3. The time complexity of the algorithm is $O(E\alpha(E, N))$.

For each node v, the algorithm computes the $\Phi(v)$, $GP(v)$ and $G^*(v)$. If $\Phi(v)$ is *true*, then a gating function is placed at v. Let X be the variable requiring the gating functions. The gating functions are built from $GP(v)$ and $G^*(v)$ as follows:

- If $G^*(v) = \emptyset$, then v is not a loop header. The γ function $X = GP(v)$ is placed at v.

- If $G^*(v) \neq \emptyset$, then v is a loop header. A μ function $X = \mu(GP(v), G*(v))$ is placed at v and an η function $X = \eta(\neg G^*, X)$ is placed immediately after v.

The placement of γ function is straightforward. For a loop header node v, we construct a μ function to select the first parameter for the first iteration of the loop and the second parameter for the rest. Hence, μ's first parameter should be $GP(v)$ and its second parameter should be $G^*(v)$. The η function determines the exiting value of a variable from v. The exiting paths of a loop are the paths that avoid all the back edges(i.e., avoid $G^*(v)$). The $\neg G^*(v)$ is constructed by reversing all the $\emptyset$'s in $G^*(v)$ to Λ and the rest to $\emptyset$'s.

5 Working through an Example

We use the following program to illustrate how the algorithm works.

```
1:      read(A)
2:      if (P) then goto 5
3:      if (Q) then
4:          A := 5
5:          while (R) do
6:              A := A + 1
7:          enddo
8:      else
9:          if (T) then
10:             A := A * 3
11:         else
12:             A := A + 6
13:         endif
14:     endif
15:     write(A)
```

Shown in Figure 1 is the dominator tree of the program. Each node is also labeled with its depth-first number *dfn*. Solid edges are the dominator tree edges; dashed edges are nontree edges in the CFG. Note that tree edges do not necessarily exist in the CFG. Edges from branch nodes are labeled by the branch conditions.

Let us first examine the situation when the algorithm processes node 11(i.e. $u = if\ (T)$). The children of node 11 are nodes 12,13,14. Node 12 has u as its only predecessor. Since u is its immediate dominator, the *idom node* branch of the algorithm is executed and obtains $GP(12) = T_t$. Node 13 also has u as its only predecessor. Similarly, the algorithm computes $GP(13) = T_f$. In both cases, there is no assignment statement along the path after u. Hence, $\Phi(11), \Phi(12)$ remains **false**, indicating that node 12 and 13 do not need $\phi-$functions. Node 14 has two predecessors: node 12 and 13. In both cases, the *sibling node* branch of the algorithm is executed. Both nodes are roots of trees containing no other nodes. Because node 12 contains an assignment to A and the edge is an unconditional branch and 12 is the first predecessor of node 14, $EVAL$ returns $(true, \Lambda^1)$. Similarly, $EVAL$ returns Λ^2 and **true** on the edge from node 13. Hence $\Phi(14)$ becomes **true** and $ListP(14)$ contains two paths starting from its siblings: node 12 and 13. Following the toporder to process the sibling set ensures that when merging paths in $ListP(14)$, $GP(12), GP(13)$ already count all the paths from the immediate dominator u. The result after the *merge* phase for node 14 is hence $(GP(12) \cdot \Lambda^1) \cup (GP(13) \cdot \Lambda^2) = \gamma(T, \Lambda^1, \Lambda^2)$. It can then be used by renaming pass to make $A_{14} = \gamma(T, A_{12}, A_{13})$. $LINK$ is called to link the subtree rooted at each child to u and form a larger subtree of the dominator tree. $UPDATE$ is called on each child to store the gating path expression for future $EVAL$ calls. Since node 14 immediately post-dominates u, its gating path expression is set to Λ for future $EVAL$ calls.

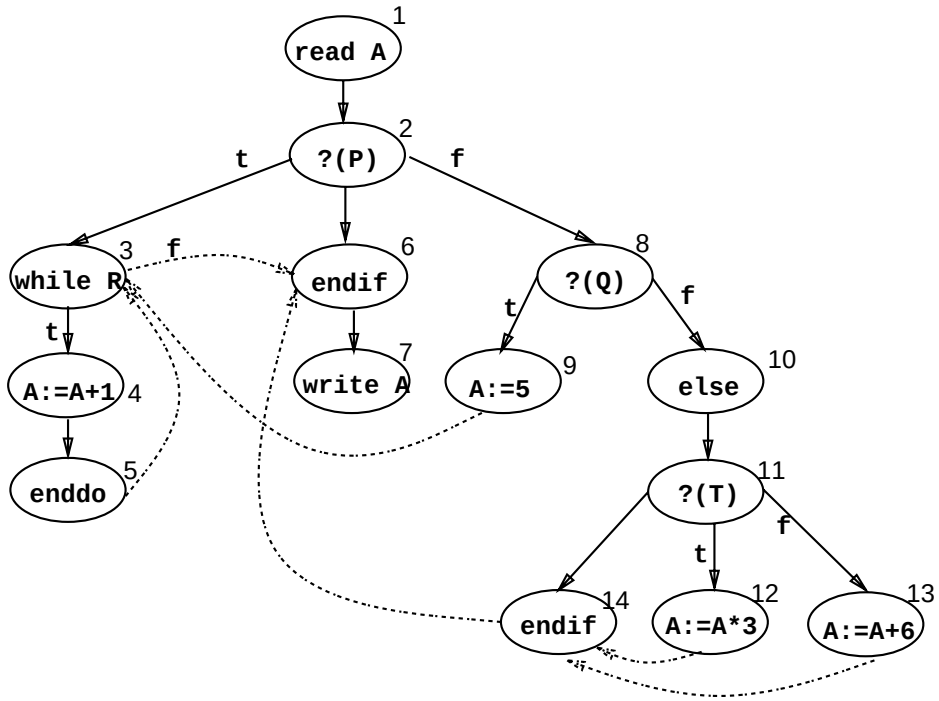

Figure 1. Dominator tree of the example program.

A more complicated situation happens when branch node 2 is processed (i.e., $u = if\ (P)$). Node 2 has three children in the dominator tree: node 3, 6 and 8. In the case of node 3, it has three predecessors: node 2, 5, and 9. Node 2 is $idom(3)$, hence $GP(13) = P_t$. The path from 5 leads to node 3 itself indicating a loop with node 3 as the loop header. Therefore, $GP^*(3) = R_t$. The path from 9 evaluates to its sibling node 8 and the path is Q_t. Hence, Q_t is inserted into $ListP(3)$. In the case of node 6, it has two predecessors: node 3 and 14. The path from node 3 evaluates to its sibling node 3, which is R_f. The path from node 14 evaluates to its sibling node 8, which is Q_f (note that EVAL evaluates to Λ for the subpath from node 14 to node 11 because $R(14)$ is Λ (as explained in $UPDATE(14, GP(14))$). Hence, $ListP(6)$ contains R_f and Q_f. Node 8 has only node 2 as predecessor. Since $idom(8) = 2$, $GP(8)$ equals P_f. The toporder is 8, 3, 6 since node 3 has a path from 8 and node 6 has paths from both 8 and 3. Merge the paths for each node. $GP(8)$ remains P_f. $GP(3)$ becomes $P_t \cup (P_f \cdot Q_t) = \gamma(P, \Lambda^1, \gamma(Q, \Lambda^2, \emptyset))$. Since $GP^*(3) = R_t$, a μ function is also needed at node 3. The form of the μ function is $\mu(GP(3), GP^*(3)) = \mu(\gamma(P, \Lambda^1, \gamma(Q, \Lambda^2, \emptyset)), \gamma(R, \Lambda^3, \emptyset))$. For node 6, $GP(6)$ becomes $(P_t \cdot R_f) \cup (P_f \cdot Q_f) = \gamma(P, \gamma(R, \emptyset, \Lambda^1), \gamma(Q, \emptyset, \Lambda^2))$. It is easy to verify that $\Phi(3)$ and $\Phi(6)$ are set to be **true** by the algorithm indicating they need $\phi-$functions. The final result after renaming will be: for node 3, $A_3 = \mu(\gamma(P, A_0, \gamma(Q, A_9, \emptyset)), \gamma(R, A_4, \emptyset))$; and for node 6, $\gamma(P, \gamma(R, \emptyset, A_3), \gamma(Q, \emptyset, A_{14}))$.

6 Implementation and Measurement

We implemented the algorithm using path compression in the POLARIS restructuring compiler[BEF+94]. The simple algorithm uses only path compression and has a complexity of $O(E \log N)$. The following is the timing result for all the programs in the Perfect Benchmark[CKPK90]. The time given is the execution time on a SUN-10 workstation. Also plotted as a reference is the shape of the curve for $0.5 + 0.0015 \times (E \log(E))$ with the same E's as in the codes.

Figure 2. Timings of the algorithm with path compression.

7 Conclusions

In this paper, we present an almost linear algorithm to place and build gating functions in a single step for GSA construction. The algorithm is based on the well-known path compression technique[Tar79]. It is easy to implement and efficient for the programs in the Perfect Benchmark.

References

[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.

[BE94] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. In *Proceedings of Supercomputing '94*, November 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.

[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.

[Far77] R. Farrow. Efficient on-line evaluation of functions defined on paths in trees. Technical report, Rice University, Dept. Math. Sci., 1977.

[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.

[Har85] P. Harel. A linear time algorithm for finding dominators in flowgraphs and related problems. In *Proc. of the 17th ACM Symposium on Theory of Computing*, May 1985.

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

[JP93] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proc. the SIGPLAN '93 Conference on Program Language Design and Implementation*, June 1993.

[JPP94] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proc. the SIGPLAN '94 Conference on Program Language Design and Implementation*, June 1994.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[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.

[SG94] V.C. Sreedhar and G.R. Gao. Computing ϕ-nodes in linear time using dj-graph. Technical Report Technical Report,ACAPS Technical Memo 75, McGill University, School of Computer Science, January 1994.

[Tar79] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of ACM*, 26(4):690–715, October 1979.

[Tar81a] Robert Endre Tarjan. Fast algorithm for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[Tar81b] Robert Endre Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

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

[TP94] Peng Tu and David Padua. GSA based demand-driven symbolic analysis. Technical Report 1339, University of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, February 1994.

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

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