Efficiently Computing ϕ-Nodes On-The-Fly

RON K. CYTRON
Washington University in St. Louis
and
JEANNE FERRANTE
University of California at San Diego

Recently, Static Single-Assignment Form and Sparse Evaluation Graphs have been advanced for the efficient solution of program optimization problems. Each method is provided with an initial set of flow graph nodes that inherently affect a problem's solution. Other relevant nodes are those where potentially disparate solutions must combine. Previously, these so-called ϕ-nodes were found by computing the iterated dominance frontiers of the initial set of nodes, a process that could take worst-case quadratic time with respect to the input flow graph. In this article we present an almost-linear algorithm for determining exactly the same set of ϕ-nodes.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Algorithms, Languages, Theory, Performance

Additional Key Words and Phrases: Static Single-Assignment (SSA) form

1. MOTIVATION AND BACKGROUND

Static Single-Assignment (SSA) form [Cytron et al. 1991] and the more general Sparse Evaluation Graphs (SEG) [Choi et al. 1991] have emerged as an efficient mechanism for solving compile-time optimization problems via data flow analysis [Alpern et al. 1988; Choi et al. 1993; 1994; Cytron et al. 1986; Rosen et al. 1988; Wegman and Zadeck 1991]. Given an input data flow framework, the SEG construction algorithm distills the flow graph into a set of *relevant* nodes. These nodes are then interconnected with edges suitable for evaluating data flow solutions. Similarly, SSA form identifies and appropriately interconnects those variable

Ron K. Cytron is currently funded by the National Science Foundation, grant CCR–9402883.
A preliminary version of the ϕ-placing algorithm appeared in *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993. At that time, both authors were Research Staff Members at IBM T.J. Watson Reserach Labs, Yorktown Heights, New York.
Authors' addresses: Ron K. Cytron, Washington University, Department of Computer Science, St. Louis, MO 63130–4899; email: cytron@cs.wustl.edu; Jeanne Ferrante, University of California, San Diego, Department of Computer Science and Engineering, 9500 Gilman Drive, La Jolla, CA 92093–0114; email: ferrante@cs.ucsd.edu.

references that are relevant to solving certain data flow problems.

The SEG and SSA algorithms take the following structures as input:

Flowgraph. Graph $\mathcal{G}_f$ has $\mathcal{N}_f$, edges $\mathcal{E}_f$, and root node `Entry`. If $(X, Y) \in \mathcal{E}_f$, then we write $X \in \mathit{Pred}(Y)$ and $Y \in \mathit{Succ}(X)$. We assume $\mathcal{G}_f$ is connected.

Depth-first numbering. Let $\mathit{dfn}(X)$ be the number associated with X

$$1 \leq \mathit{dfn}(X) \leq |\mathcal{N}_f|$$

in a depth-first search of $\mathcal{G}_f$.[1] We call the associated depth-first spanning tree $DFST$. Similarly, we define the inverse mapping

$$\mathit{vertex}(k) = X \mid \mathit{dfn}(X) = k.$$

Dominator tree. Let $\mathit{idom}(X)$ be the immediate dominator of flow graph node X. We say that X *dominates* Y, written $X \underline{\gg} Y$, if X appears on every path from flow graph `Entry` to Y; domination is both reflexive and transitive. We say that X *strictly dominates* Y, written $X \gg Y$, if $X \underline{\gg} Y$ and $X \neq Y$. Each node X has a unique *immediate dominator* $\mathit{idom}(X)$ such that

$$\mathit{idom}(X) \gg X \text{ and } \forall\, W \gg X,\ W \underline{\gg}\, \mathit{idom}(X).$$

Node $\mathit{idom}(X)$ serves as the parent of X in a flow graph's dominator tree. (An example flow graph and its dominator tree are shown in Figure 1.)

Initial nodes. $\mathcal{N}_\alpha \subseteq \mathcal{N}_f$ is an initial subset of those nodes that must appear in the sparse representation. For an SEG, such nodes represent nonidentity transference in a data flow framework. For SSA, such nodes contain *definitions* of variables.

The SEG and SSA algorithms essentially produce the following structures as output:

Sparse nodes. $\mathcal{N}_\sigma$, which we compute as a property of each node:

$$\Upsilon(X) \Longleftrightarrow X \in \mathcal{N}_\sigma$$

ϕ-function nodes. $\mathcal{N}_\phi \subseteq \mathcal{N}_\sigma$, which we compute as a property of each node:

$$\Phi(X) \Longleftrightarrow X \in \mathcal{N}_\phi$$

Previous SEG and SSA construction algorithms operate as follows:

(1) The algorithm precomputes the *dominance frontier* $DF(X)$ for each node X:

$$DF(X) = \{\, Z \mid (\exists (Y, Z) \in \mathcal{E}_f)(X \underline{\gg} Y \text{ and } X \not\gg Z) \,\}$$

In other words, X dominates a predecessor of Z without strictly dominating Z. The dominance frontiers for the flow graph in Figure 1 are shown in Figure 2.

(2) The algorithm accepts as input $\mathcal{N}_\alpha \subseteq \mathcal{N}_f$.

(3) The algorithm then computes the set of nodes deserving ϕ-functions, $\mathcal{N}_\phi$, as the iterated dominance frontier of the initial set of nodes:

$$\mathcal{N}_\phi = DF^{+}(\mathcal{N}_\alpha)$$

In our example in Figures 1 and 2, if $\mathcal{N}_\alpha = \{\, D, W \,\}$, then $\mathcal{N}_\phi = \{\, W, X, Y, Z \,\}$.

[1]Here, $\mathit{dfn}(X)$ is assigned in order of nodes visited, starting with 1; in Aho et al. [1986], depth-first numbers are assigned starting from $|\mathcal{N}_f|$ down to 1.

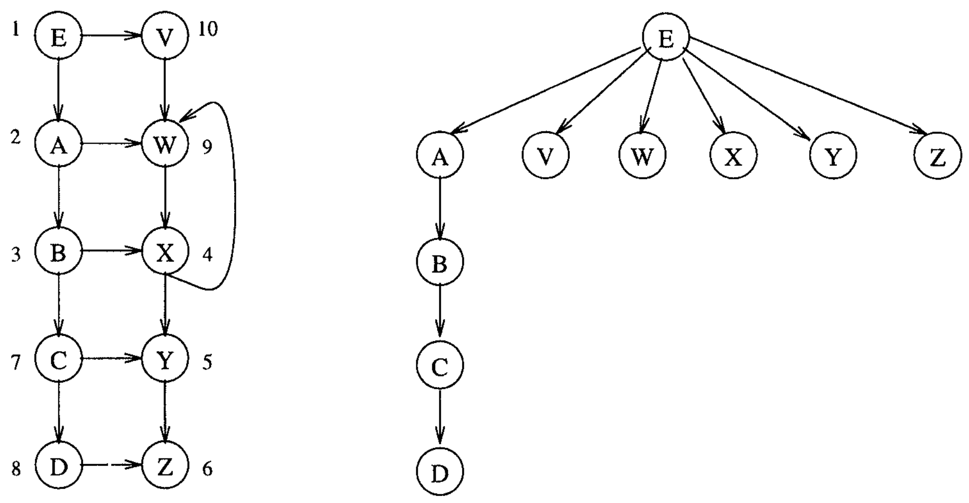

Fig. 1. Flow graph and its dominator tree.

X	$DF(X)$
A	$\{W,X,Y,Z\}$
B	$\{X,Y,Z\}$
C	$\{Y,Z\}$
D	$\{Z\}$
V	$\{W\}$
W	$\{X\}$
X	$\{W,Y\}$
Y	$\{Z\}$
Z	$\{\}$
E	$\{\}$

Fig. 2. Dominance frontiers for Figure 1.

(4) Steps (2) and (3) can be repeated to create a forest of (related) sparse evaluation graphs. In SSA form, these graphs are usually combined, with the appropriate increase in detail and size with respect to $\mathcal{N}_\phi$.

(5) In an SEG, appropriate edges are then placed between nodes in

$$\mathcal{N}_\sigma = \mathcal{N}_\alpha \cup \mathcal{N}_\phi.$$

In SSA form, variables are appropriately renamed, such that each used is reached by a single definition.

These two methods have one component in common, in name as well as function: the determination of $\mathcal{N}_\phi$, where potentially disparate information combines. Consider a flow graph $\mathcal{G}_f$ with N nodes (set $\mathcal{N}_f$) and E edges (set $\mathcal{E}_f$) for a program with V variables. While computing the so-called ϕ-nodes is efficient in practice [Cytron et al. 1991], the following observations are relevant:

(1) Constructing a single SEG (i.e., one data flow framework) by the usual algorithm [Choi et al. 1991] takes $O(E + N^2)$ time.

(2) Where a data flow problem can be partitioned into V disjoint frameworks [Marlowe 1989], constructing the associated V SEGs takes $O(EV + N^2)$ and $\Omega(EV)$ time.
(3) If we bound the number of variable references per node by some constant, then construction of SSA form takes $O(EV + N^2)$ and $\Omega(E + V + N)$ time.

In comparing (2) to (3), note that SEG provides a "solution" for each edge in the flow graph, while SSA form provides a "solution" only at a program's variable references.

Our algorithm for placing ϕ-functions avoids the the computation of dominance frontiers. In doing so, we reduce the time bound for (1) to $O(E\alpha(E))$, where $\alpha()$ is the slowly growing inverse-Ackermann function [Cormen et al. 1990]. The time bound for (2) is reduced to $O(V \times E\alpha(E))$. If our algorithm places ϕ-functions for SSA form, then the time bound for (3) becomes $O(V \times E\alpha(E))$ but $\Omega(EV)$.

To summarize the above discussion, computation of dominance frontiers and their use in placing ϕ-functions can take $O(E + N^2)$ time [Cytron et al. 1991], although such behavior is neither expected in general nor even possible for programs of certain structure. An example flow graph that exhibits the aforementioned worst-case behavior is shown in Figure 1. The flow graph's dominance frontiers are shown in Figure 2. As this graph structure grows,[2] the size of dominance frontiers of nodes along its left spine increases quadratically, while the size of the sparse data flow graph or SSA form is certainly linear in size. It is this worst-case behavior brought on by precomputing the dominance frontiers that we wish to avoid.

Since one reason for introducing ϕ-functions is to eliminate potentially quadratic behavior when solving actual data flow problems, such worst-case behavior during SEG or SSA construction could be problematic. Clearly, avoiding such behavior necessitates placing ϕ-functions without computing or using dominance frontiers.

In this article we present an algorithm that computes $\mathcal{N}_\sigma$ and $\mathcal{N}_\phi \subseteq \mathcal{N}_\sigma$ from the initially specified $\mathcal{N}_\alpha$. Where previous algorithms begin with a set of nodes $\mathcal{N}_\alpha$ and use dominance frontiers (iteratively) to induct other nodes into $\mathcal{N}_\sigma$, our algorithm does the reverse: we visit nodes in an order that allows us to determine conditions under which a given node must be in $\mathcal{N}_\sigma$. Using a similar approach, we can then determine $\mathcal{N}_\phi \subseteq \mathcal{N}_\sigma$.

In Section 2, we discuss a simple version of our algorithm and illustrate its application to the flow graph in Figure 1. The algorithm's correctness is shown in Section 3. In Section 4, we discuss how balanced path-compression can be used to make the algorithm more efficient; it is these techniques that allow us to achieve our almost-linear time bound. Section 5 gives some preliminary experiments, and Section 6 suggests future work.

In related work, Johnson and Pingali [1993] describe an algorithm to construct an SSA-like representation that takes $\Theta(EV)$ time. While their upper bound is slightly better than ours, our approach is more general: we construct sparse evaluation graphs for *arbitrary* data flow problems, while Johnson and Pingali construct def-use structures specific to the solution of SSA-based optimization problems such as constant propagation.

[2]by repeating the ladder structure; the back edge is unnecessary and was added to illustrate our algorithm

As we discuss further in Section 5, $O()$ asymptotic bounds in this area are deceptive, and one must take into account lower and expected bounds. The usual dominance-frontier-based algorithm [Cytron et al. 1991] is biased toward the *average* case, in which linear behavior for constructing or consulting dominance frontiers is expected. The algorithm we present in this article, as well as the algorithm due to Johnson and Pingali [1993], is $\Omega(EV)$, since each edge in the flow graph is examined for each variable.

We actually present two variations of the same algorithm in this article. The first "simple" algorithm is presented for expository reasons; the second algorithm uses balanced path-compression to achieve our improved time bound. The experiments presented in Section 5 compare the simple algorithm presented in Section 2 with the usual dominance-frontier-based algorithm [Cytron et al. 1991]. In fact, the usual algorithm is often faster, and so these experiments do not suggest blindly switching to the asymptotically faster algorithm. However, our algorithm does exhibit the same linear behavior as the usual algorithm. Moreover, we have not implemented the balanced path-compression presented in Section 4 which yields our better bound. These experiments give some evidence that our algorithm can yield comparable performance to the usual algorithm, while avoiding asymptotically poor efficiency.

2. ALGORITHM

To avoid computing all of the dominance frontier relation, our algorithm will use a specific *order* to determine elements of the relation. If this order is followed, the algorithm guarantees that no elements of the relation will be missed.

The algorithm splits into two cases. In the "typical" case, to determine if $Z \in DF_{n+1}(\mathcal{N}_\sigma)$ for some n, there must be an $X \in DF_n(\mathcal{N}_\sigma)$ such that

$$(\exists (Y, Z) \in \mathcal{E}_f) X \geqslant\!\!\!\!\!\geqslant Y \text{ and } X \not\gg Z.$$

Thus for any such edge (Y, Z), we need only check the dominator tree between $idom(Z)$ and Y for nodes already determined to be in $DF_n(\mathcal{N}_\sigma)$. We need an order that guarantees that $X \in DF_n(\mathcal{N}_\sigma)$ is already determined. Reverse depth-first numbering provides such an order. It turns out that $dfn(idom(X)) \geq dfn(idom(Z))$. Our algorithm thus examines nodes whose immediate dominators have decreasing depth-first numbers.

The second case is when X and Y are siblings in the dominator tree. No order is determined by the "typical case," since here both nodes have the same immediate dominator.

We define the *equidominates* of a node X as those nodes with the same immediate dominator as X:

$$equidom(X) = \{ Y \mid idom(Y) = idom(X) \}.$$

For example, in Figure 1,

$$equidom(A) = \{ A, V, W, X, Y, Z \}.$$

More generally, the noun *equidominates* refers to any such set of nodes.

Our algorithm essentially partitions equidominates into blocks of nodes that are in each other's iterated dominance frontiers, but without the expense of explicitly

```
Algorithm[1] Sparse graph node determination (simple)
    $NodeCount \leftarrow 0$
    foreach $(X \in \mathcal{N}_f)$ do
        $\Upsilon(X) \leftarrow$ false ;   $\Phi(X) \leftarrow$ false ;   $Map(X) \leftarrow X$
    od
    $Cnum \leftarrow 0$
    for $n = N$ downto 1 do                                                $\Leftarrow \boxed{1}$
        foreach $(\{ Z \mid dfn(idom(Z)) = n \})$ do                          $\Leftarrow \boxed{2}$
            if $(Cnum(Z) = 0)$ then   call $Visit(Z)$   fi
        od
    od
                                                                           $\Leftarrow \boxed{3}$
    for $n = N$ downto 1 do                                                $\Leftarrow \boxed{4}$
        foreach $(\{ Z \mid dfn(idom(Z)) = n \})$ do   call $Finish(Z)$   od    $\Leftarrow \boxed{5}$
    od
end
Function $FindSnode(Y, P)$ : $node$
    for $(X = Y)$ repeat$(X = idom(X))$ do
        if $(\Upsilon(Map(X))$ or $(idom(X) = P))$ then   return $(Map(X))$   fi
    od
end
Procedure $IncludeNode(X)$
    $\Upsilon(X) \leftarrow$ true                                                  $\Leftarrow \boxed{6}$
end
Procedure $Finish(Z)$
    foreach $((Y, Z) \in \mathcal{E}_f \mid Y \neq idom(Z))$ do
        $s \leftarrow FindSnode(Y, idom(Z))$
        if $(\Upsilon(s))$ then
            $\Phi(Z) \leftarrow$ true                                              $\Leftarrow \boxed{7}$
        fi
    od
    if $(\Upsilon(Map(Z)))$ then
        call $IncludeNode(Z)$                                              $\Leftarrow \boxed{8}$
    fi
end
```

Figure 3

computing dominance frontiers.

It is the control flow edges between equidominates that here determines the needed order. If the equidominates are in the same strongly connected component (SCC) of control flow edges, then any of the equidominates in the SCC is in $DF^+(\mathcal{N}_\sigma)$ iff one of the equidominates is. Our algorithm finds these SCCs and determines the dominance frontier relation for a representative node, $Map(X)$. The membership of this node is then used to determine the membership of all nodes in its SCC. If the equidominates are not in the same SCC, then the topological order determined by control flow edges between their respective SCCs is the correct order to follow.

Our algorithm uses a well-known algorithm [Aho et al. 1974] to find strongly connected components of the dominance frontier graph (in which an edge from

```
Procedure $Visit(Z)$
    $LL(Z) \leftarrow Cnum(Z) \leftarrow ++NodeCount$
    call $push(Z)$                                                    $\Leftarrow \boxed{9}$
    foreach $\big((Y, Z) \in \mathcal{E}_f \mid Y \neq idom(Z)\big)$ do                 $\Leftarrow \boxed{10}$
        $s \leftarrow FindSnode(Y, idom(Z))$                                   $\Leftarrow \boxed{11}$
        if $(idom(s) \neq idom(Z))$ then
            call $IncludeNode(Z)$                                          $\Leftarrow \boxed{12}$
        else
            if $(Cnum(s) = 0)$ then
                call $Visit(s)$                                            $\Leftarrow \boxed{13}$
                $LL(Z) \leftarrow min(LL(Z), LL(s))$
            else
                if $((Cnum(s) < Cnum(Z))$ and $OnStack(s))$ then
                    $LL(Z) \leftarrow min(LL(Z), Cnum(s))$
                fi
            fi
            if $(\Upsilon(s)$ and not $OnStack(s))$ then                          $\Leftarrow \boxed{14}$
                call $IncludeNode(Z)$                                      $\Leftarrow \boxed{15}$
            fi
        fi
    od
    if $(LL(Z) = Cnum(Z))$ then
        repeat                                                        $\Leftarrow \boxed{16}$
            $Q \leftarrow pop()$                                                $\Leftarrow \boxed{17}$
            $Map(Q) \leftarrow Z$
            if $(Q \in \mathcal{N}_\alpha)$ then
                call $IncludeNode(Q)$                                      $\Leftarrow \boxed{18}$
            fi
            if $(\Upsilon(Q))$ then
                call $IncludeNode(Z)$                                      $\Leftarrow \boxed{19}$
            fi
        until $(Q = Z)$
    fi
end
```

Figure 4

Y to Z implies $Z \in DF(Y)$). That algorithm finds the roots in the depth-first spanning tree containing the SCCs in the order of last visit by depth-first search. To do this, the algorithm uses auxiliary data structures $Cnum(X)$ and $LL(X)$ [Aho et al. 1974]. Briefly, $Cnum$ is a forward, depth-first number assigned by procedure $Visit$ (as opposed to the reverse depth-first number dfn assigned initially). LL is used to determine if a node is a root of a SCC. "SCC" or "component" refers to nodes associated in this manner.

We now give an overview of the algorithm. The main procedure initializes essential data structures and then visits each node not already visited (by calling the procedure $Visit$) in reverse depth-first order of immediate dominators. After all nodes have been visited, the main procedure calls $Finish$ to determine membership in $\mathcal{N}_\phi$ from $\mathcal{N}_\sigma$.

The procedure Visit forms the heart of the algorithm. Here, the input node Z is first put on a stack (used to keep track of SCCs). Then the dominator tree between $idom(Z)$ and Y is searched for each control flow edge (Y, Z) for a node X already determined to be in $DF^+(\mathcal{N}_\alpha)$ (by calling *FindSnode*). Depending on the success of the search, a further call to *Visit* may be necessary, or we may be able to determine directly that the node should be placed in $\mathcal{N}_\sigma$ (accomplished with a call to *IncludeNode*). The final code in *Visit* simultaneously determines the membership in $\mathcal{N}_\sigma$ of all nodes in the SCC (which are currently on the stack) and pops the entire stack.

Central to this algorithm is the function *FindSnode*(Y, P), which ascends the dominator tree from node Y, searching for a sparse graph node below P. Our proof of algorithmic correctness (Section 3) relies on the nature of *FindSnode*() rather than its actual implementation: the function always returns

$$X \mid Z \in DF^+(X),\ Z \in Succ(Y),\ P \gg X \underline{\gg} Y.$$

We present a straightforward, albeit inefficient, version of *FindSnode*(Y, P) in Figure 3. The correctness of our algorithm as presented in Section 3 is based on the behavior of *FindSnode*(). To obtain our almost-linear time bound, we modify our algorithm as discussed in Section 4.

The algorithm is shown in Figures 3 and 4. We now illustrate the application of the algorithm to the flow graph in Figure 1, assuming that $\mathcal{N}_\alpha = \{B\}$. Although the full proofs appear in Section 3, we mention here that

$$Z \in DF(X) \Rightarrow dfn(idom(X)) \geq dfn(idom(Z)).$$

By ensuring that nodes with higher depth-first number have already been correctly determined to be in $\mathcal{N}_\sigma$, the algorithm can correctly determine this property for nodes of lower depth-first number.

The loop at $\boxed{1}$ begins with V (depth-first numbered 10 in Figure 1). Since no nodes are dominated by V, no steps are taken by $\boxed{2}$; the same holds for nodes W and D. When $\boxed{1}$ considers C, loop $\boxed{2}$ calls *Visit* with D. The loop in *Visit* at $\boxed{10}$ is empty, since the only predecessor of D immediately dominates D. Thus, $Map(D)$ is set to D in loop $\boxed{16}$. Loop $\boxed{1}$ then considers in turn Z, Y, and X, each of which dominates no node, so $\boxed{2}$ is empty. When $\boxed{1}$ considers B, *Visit* is called on C; since C's only predecessor immediately dominates C, no action is taken by $\boxed{10}$, and $Map(C)$ is set to C. When *Visit* is next called on B, $Map(B)$ is set to B; also, since $B \in \mathcal{N}_\alpha$, step $\boxed{18}$ places B in $\mathcal{N}_\sigma$.

When $\boxed{1}$ considers E, suppose loop $\boxed{2}$ considers the nodes immediately dominated by E in order A, Y, W, X, V, and Z (although some of these will already have been processed by recursive calls to *Visit*). When *Visit* works on A, no steps of $\boxed{10}$ are taken, and $Map(A)$ becomes A. When *Visit* works on Y, *FindSnode* will be called on C and X.

—For C, *FindSnode* returns B; since B and Y are not equidominates, Y is placed in $\mathcal{N}_\sigma$ by $\boxed{12}$.

—For X, *FindSnode* returns X; since X and Y are equidominates, *Visit* is called recursively on X. In this invocation, loop $\boxed{10}$ considers nodes B and W.

—For B, *FindSnode* returns B, so X is placed in $\mathcal{N}_\sigma$.

—For W, *FindSnode* returns W, so *Visit* is called recursively on W. In this invocation, loop $\boxed{10}$ considers nodes A, V, and X.

—For A, *FindSnode* returns A. Since that node has already been visited, and since A is not in $\mathcal{N}_\sigma$, nothing happens to W because of A.

—For V, *FindSnode* returns V; *Visit* is then called recursively for V, where $Map(V)$ becomes V.

—For X, *FindSnode* returns X, which is already on stack, so nothing happens. In particular, $Map(X)$ is not set.

Now, $Map(W)$ and $Map(X)$ are both set to W. Since $X \in \mathcal{N}_\sigma$, W is placed in $\mathcal{N}_\sigma$.

When loop $\boxed{2}$ considers W, X, and V, each has already been visited. When *Visit* is called for Z, nodes D and Y are considered by $\boxed{10}$.

—For D, *FindSnode* returns B, so Z is placed in $\mathcal{N}_\sigma$.

—For Y, *FindSnode* returns Y, which has already been visited.

In contrast, previous methods [Choi et al. 1991; Cytron et al. 1991] would not only have constructed all dominance frontier sets in Figure 2, which are asymptotically quadratic in size, but would also have iterated through the dominance frontier sets of all nodes put on the worklist, namely $\{E, B, V, W, X, Y, Z\}$.

3. CORRECTNESS

The proofs contained in this section establish the following:

(1) The order in which nodes are considered by the algorithm presented in Section 2 suffices to identify the placement of ϕ-nodes. This property follows from general observations about flow graphs, dominator trees, and dominance frontiers, which are proven in Theorems 3.5, 3.6, and 3.7; these theorems are offered separately, as they may be useful in other contexts. The node-ordering property is itself presented as Corollary 3.8.

(2) The algorithm correctly identifies the placement of ϕ-nodes. Relying on the general results shown in previous theorems, Theorems 3.18, 3.19, and 3.20 prove that the algorithm presented in Section 2 computes the correct set of ϕ-nodes. The pitons of this proof sequence are as follows:

—Lemma 3.11: The *Map*() structure correctly relates equidominates.

—Lemma 3.16: Two different nodes have the same *Map*() value only if one is in the iterated dominance frontier of the other.

—Theorem 3.18: The node representing a set of equidominates is correctly identified as a sparse node.

—Theorem 3.19: When the algorithm terminates, ϕ-nodes have been correctly identified.

Theorem 3.20 is then easily established: the algorithm correctly identifies the sparse nodes.

LEMMA 3.1. $(Y, Z) \in \mathcal{E}_f \Longrightarrow idom(Z) \gg\!\!\!\!= Y$.

PROOF. Suppose not. Then there exists a path $P : Root \xrightarrow{+} Y$ that does not contain $idom(Z)$. If path P is extended by an edge (Y, Z), then we obtain a path to Z that does not contain $idom(Z)$. Thus, $(Y, Z) \notin \mathcal{E}_f$. □

COROLLARY 3.2.

$$\begin{aligned}(Y, Z) \in \mathcal{E}_f \textit{ and } Y \neq idom(Z) \Longrightarrow \\ dfn(idom(Z)) \leq dfn(idom(Y)) < dfn(Y).\end{aligned}$$

COROLLARY 3.3.

$$\begin{aligned}(Y, Z) \in \mathcal{E}_f \textit{ and } Y \neq idom(Z) \Longrightarrow \\ idom(Z) \underline{\gg} idom(Y) \gg Y.\end{aligned}$$

LEMMA 3.4. *If X is an ancestor of $idom(Z)$ in the depth-first spanning tree DFST of $\mathcal{G}_f$, and $idom(Z) \gg Y$, then $X \gg Y \implies X \gg Z$.*

PROOF. Suppose not. Then there exists a path $Root \xrightarrow{*} idom(Z) \xrightarrow{+} Z$ that excludes X. Since $idom(Z) \gg Y$, there is a path of DFST edges from $idom(Z)$ to Y that excludes X. Thus, X cannot dominate Y. □

THEOREM 3.5. *If X is an ancestor of $idom(Z)$ in the depth-first spanning tree DFST of $\mathcal{G}_f$, then Z cannot be in the dominance frontier of X.*

PROOF. Suppose $Z \in DF(X)$. Then $\exists (Y, Z) \in \mathcal{E}_f$ such that $X \underline{\gg} Y$ and $X \not\gg Z$. It cannot be the case that $idom(Z) = Y$, since if so $X \gg Y = idom(Z) \gg Z$, a contradiction. Therefore, $idom(Z) \neq Y$. By Corollary 3.3, $idom(Z) \underline{\gg} idom(Y) \gg Y$. By Lemma 3.4, $X \gg Y \Rightarrow X \gg Z$. But since $X \gg Y$, we have $X \gg Z$, a contradiction. Hence $Z \notin DF(X)$. □

THEOREM 3.6. $Z \in DF(X) \Longrightarrow dfn(X) > dfn(idom(Z))$.

PROOF. Suppose $dfn(X) \leq dfn(idom(Z))$, but $Z \in DF(X)$. In the DFST of $\mathcal{G}_f$, either

(1) X is an ancestor of $idom(Z)$. By Theorem 3.5, $Z \notin DF(X)$, or
(2) X is to the "left of" $idom(Z)$. Suppose there existed some node $Y \in Preds(Z)$ dominated by X. Since Y must be a descendant of X in DFST, Y is also to the left of $idom(Z)$, and therefore Y is to the left of Z. But Z cannot have a predecessor to its left in DFST. Therefore $Z \notin DF(X)$.

Either case reached a contradiction, thus proving the theorem. □

THEOREM 3.7. $Z \in DF(X) \Longrightarrow idom(Z) \gg X$.

PROOF. Suppose not. Then either

(1) $idom(Z) = X$, which implies $Z \notin DF(X)$; or,
(2) $idom(Z) \neq X$ and $idom(Z) \not\gg X$. Consider then the path

$$P : Root \xrightarrow{+} X$$

that does not contain $idom(Z)$. If $Z \in DF(X)$, then we can extend path P to

$$Q : Root \xrightarrow{+} X \xrightarrow{*} Y \rightarrow Z$$

such that $X \gg Y$ and $X \not\gg Z$. With $X \gg Y$, we can construct Q such that edges between X and Y are DFST edges. Since $idom(Z)$ does not occur on path P, $idom(Z)$ must occur on path Q after node X and before node Z. Thus, X is a DFST ancestor of $idom(Z)$. By Theorem 3.5, $Z \notin DF(X)$.

Either case reached a contradiction, thus proving the theorem. □

COROLLARY 3.8. $Z \in DF(X) \Longrightarrow dfn(idom(X)) \geq dfn(idom(Z))$.

The following lemmas formalize those properties of our algorithm that participate in our correctness proof. We omit proofs that directly follow from inspection of our algorithm.

LEMMA 3.9. *Visit is called exactly once for each node in Flowgraph.*

LEMMA 3.10. *During all calls to Visit, each node is pushed and popped exactly once.*

PROOF. By Lemma 3.9, $Visit(Z)$ is invoked exactly once; on this call, Z is pushed exactly once. We need to show that Z is popped exactly once. If $Z = Map(Z)$, then Z is popped by the invocation of $Visit$ in which Z was pushed. Otherwise, Z belongs to a strongly connected component represented by H, $H \neq Z$, in which case Z is popped by the iteration in which H is pushed. □

LEMMA 3.11. $Y = Map(X) \Longrightarrow idom(Y) = idom(X)$.

PROOF. Since initially $X = Map(X)$, the lemma holds at the start of the algorithm. Otherwise, $Map(X)$ is only set during the loop at step $\boxed{16}$, when equidominates are popped off the stack. In this case, we have $idom(Map(X)) = idom(X)$. □

LEMMA 3.12. *At step $\boxed{15}$, node s (referenced at step $\boxed{14}$) has already been pushed and popped.*

PROOF. By the predicate of step $\boxed{14}$, node s cannot be on stack at step $\boxed{15}$. Thus, s either has not been pushed yet, or else s has been pushed and popped. If s has not been pushed then $Cnum(s) = 0$, but then step $\boxed{13}$ would have pushed s before step $\boxed{15}$ is reached. Therefore, s has been pushed and popped. □

COROLLARY 3.13. *Any invocation of $IncludeNode(s)$ must already have occurred at step $\boxed{15}$.*

LEMMA 3.14. $OnStack(X)$ *and* $OnStack(Y) \Longrightarrow idom(X) = idom(Y)$.

LEMMA 3.15. *At step $\boxed{11}$, $FindSnode()$ returns $s \mid s = Map(S),\ Z \in DF(S)$.*

PROOF. Follows from inspection of $FindSnode()$ and the definition of dominance frontiers. □

LEMMA 3.16. $Map(X) = Map(Y) \Longrightarrow Y \in DF^+(X)$ *or* $X = Y$.

PROOF. Follows from initialization ($Map(X) = X$), Lemma 3.15, and the observation that $Map(X)$ represents the strongly connected component containing X. □

COROLLARY 3.17. $s \in \mathcal{N}_\sigma \iff Map(s) \in \mathcal{N}_\sigma$.

THEOREM 3.18. *As of step* $\boxed{3}$, $\Upsilon(Map(X)) \Longleftrightarrow Map(X) \in \mathcal{N}_\sigma$.

PROOF. We consider the two implications separately.

($\Longrightarrow$). We actually prove a stronger result: $\Upsilon(X) \Longrightarrow X \in \mathcal{N}_\sigma$. To accommodate the iteration of step $\boxed{1}$, we prove the induction hypothesis

$$IH_A(n) \equiv (dfn(idom(X)) = n) \wedge \Upsilon(X) \implies X \in \mathcal{N}_\sigma$$

by backward induction on n (following the progression of our algorithm), noting that $\Upsilon(X)$ can only be set in procedure $Visit(Z)$ when $idom(Z) = idom(X)$.

Base Case. Node $vertex(N)$ is childless in its depth-first spanning tree, and so cannot immediately dominate any node. With loop at step $\boxed{2}$ empty, this case is trivially satisfied.

Inductive Step. Consider those steps that potentially set

$$\Upsilon(X) \mid dfn(idom(X)) = n.$$

Step $\boxed{12}$. With $idom(s) \neq idom(Z)$, step $\boxed{11}$ must have returned

$$s \mid \Upsilon(s), s = Map(S), Z \in DF(S).$$

From Lemma 3.15 we obtain $Z \in DF^+(s)$. From Lemma 3.11 and with $idom(s) \neq idom(Z)$, Corollary 3.8 implies

$$dfn(idom(s)) > dfn(idom(Z)).$$

Applying $IH_A(k) \mid N \geq k \geq n+1$, we obtain

$$\Upsilon(s) \Longrightarrow s \in \mathcal{N}_\sigma.$$

Thus, $Z \in \mathcal{N}_\sigma$.

Steps $\boxed{15}$, $\boxed{18}$, *and* $\boxed{19}$. Each of these steps determines $\Upsilon(X)$ by consulting nodes whose immediate dominator is $idom(X)$. Each node in the set

$$\{\, X \mid dfn(idom(X)) = n \,\}$$

gets pushed and popped exactly once (by steps $\boxed{9}$ and $\boxed{17}$). We name such nodes $\{\, x_1, x_2, \ldots, x_L \,\}$ according to the order in which they emerge from the stack: node x_1 is the first such node popped; node x_i is popped before node x_{i+1}: and node x_L is the last such node popped. Accordingly, we define the predicate

$$Popped(k) \equiv \bigwedge_{i=k+1}^{L} OnStack(x_i)$$

which is true when exactly k such nodes have been popped. We now prove the following induction hypothesis:

$$IH_B(n) \equiv (Popped(n)) \wedge \Upsilon(X) \Longrightarrow X \in \mathcal{N}_\sigma.$$

Base Case. Prior to popping x_1, Lemma 3.14 ensures that steps $\boxed{18}$ and $\boxed{19}$ cannot affect any of the x_i. By Lemma 3.12, step $\boxed{15}$ requires s to be an already popped x_i, so step $\boxed{15}$ cannot affect any of the x_i.

Inductive Step. We now prove $IH_B(n)$.

Step $\boxed{15}$. By Lemma 3.12, s has already been pushed and popped. By Corollary 3.13, and assuming $IH_B(k) \mid 1 \le k \le n-1$ we obtain

$$\begin{array}{rcl} s & \in & \mathcal{N}_\sigma \\ Z & \in & DF^+(s) \end{array}$$

Thus, $Z \in \mathcal{N}_\sigma$.

Step $\boxed{18}$. By definition, $Q \in \mathcal{N}_\alpha \Longrightarrow Q \in \mathcal{N}_\sigma$.

Step $\boxed{19}$. Two cases:

$Q \neq Z$. This statement cannot affect $\Upsilon(Q)$, whose membership in $\mathcal{N}_\sigma$ is then covered by the other cases in this proof. From Lemma 3.16, $Z \in DF^+(Q)$, and so $Z \in \mathcal{N}_\sigma$.

$Q = Z$. This statement becomes tautologous.

($\Longleftarrow$). To prove $Map(Y) \in \mathcal{N}_\sigma \Longrightarrow \Upsilon(Map(Y))$ it is sufficient to show $Y \in Map(\mathcal{N}_\sigma) \Longrightarrow \Upsilon(Y)$. We formalize iterative dominance frontiers by:

$$\begin{array}{rcl} \mathcal{DF}^0(\mathcal{N}_\alpha) & = & Map(\mathcal{N}_\alpha) \\ \mathcal{DF}^i(\mathcal{N}_\alpha) & = & Map(DF(\mathcal{DF}^{i-1}(\mathcal{N}_\alpha))) \end{array}$$

so that

$$\bigcup_{i=0}^{\infty} \mathcal{DF}^i(\mathcal{N}_\alpha) = Map(\mathcal{N}_\sigma).$$

We now prove the following induction hypothesis:

$$IH_C(n) \equiv Y \in \mathcal{DF}^n(\mathcal{N}_\alpha) \Longrightarrow \Upsilon(Y).$$

Base Case. Every node in the flow graph is pushed and popped by steps $\boxed{9}$ and $\boxed{17}$. At step $\boxed{18}$, $y \in \mathcal{N}_\alpha \Longrightarrow \Upsilon(y)$. At step $\boxed{19}$, $Y \in Map(\mathcal{N}_\alpha) \Longrightarrow \Upsilon(Y)$ where $Y = Map(y)$.

Inductive Step. We now prove $IH_C(n)$ assuming $IH_C(n-1)$. Consider any

$$Y \in \mathcal{DF}^n(\mathcal{N}_\alpha), n > 0.$$

Since $Y \in Map(DF(\mathcal{DF}^{n-1}(\mathcal{N}_\alpha)(\mathcal{N}_\alpha)))$ we have

$$Y = Map(y) \mid y \in DF(X), X \in \mathcal{DF}^{n-1}(\mathcal{N}_\alpha).$$

By Lemma 3.11, $idom(Y) = idom(y)$, so Corollary 3.8 can be extended to

$$y \in DF(X) \Longrightarrow dfn(idom(X)) \ge dfn(idom(Y)).$$

We show that $Visit(y)$ will set $\Upsilon(Y)$ true. By $IH_C(n-1)$, we have $\Upsilon(X)$ true. Since $X \in \mathcal{DF}^{n-1}(\mathcal{N}_\alpha)$, the call to $FindSnode()$ at step $\boxed{11}$ will return X. There are two cases:

$dfn(idom(Y)) < dfn(idom(X))$. Step $\boxed{12}$ sets $\Upsilon(y)$ true. When y is popped at step $\boxed{17}$, step $\boxed{19}$ sets $\Upsilon(Y)$ true.

$dfn(idom(Y)) = dfn(idom(X))$. Two cases:

—$X \neq Y$. Step $\boxed{15}$ sets $\Upsilon(y)$ true. When y is popped at step $\boxed{17}$, step $\boxed{19}$ sets $\Upsilon(Y)$ true.

—$X = Y$. then $\Upsilon(X) \Longrightarrow \Upsilon(Y)$.

Thus, both implications hold. □

THEOREM 3.19. *After the call to Finish(Z),* $\Phi(Z) \Longleftrightarrow Z \in \mathcal{N}_\phi$.

PROOF.

$$\begin{aligned} \Phi(Z) &\Longleftrightarrow \exists (Y, Z) \in \mathcal{E}_f \mid \\ Y &\neq idom(Z), \\ s &= FindSnode(Y, idom(Z)), \\ s &\in \mathcal{N}_\sigma \end{aligned}$$

by construction in *Finish*() and Theorem 3.18. But this holds

$$\Longleftrightarrow Z \in DF^+(s), s \in \mathcal{N}_\sigma$$

by construction in *FindSnode*(), and this holds

$$\Longleftrightarrow Z \in \mathcal{N}_\phi$$

by definition of $\mathcal{N}_\phi$. □

THEOREM 3.20. *After calls to Finish(),* $\Upsilon(X) \Longleftrightarrow X \in \mathcal{N}_\sigma$.

PROOF. $X \in \mathcal{N}_\sigma \Longleftrightarrow Map(x) \in \mathcal{N}_\sigma$ (by Corollary 3.17). But $Map(x) \in \mathcal{N}_\sigma \Longleftrightarrow \Upsilon(Map(X))$ (by Theorem 3.18). But in *Finish*, $\Upsilon(Map(X)) \Longleftrightarrow \Upsilon(X)$. □

4. COMPLEXITY

In this section, we first show that our algorithm is $O(N + E + T)$, where N is the number of nodes and E the number of edges in the input flowgraph, and T is the total time for all calls to *FindSnode*. Unfortunately, T is not linear using *FindSnode* as written. We then provide a faster version of our algorithm and show that our correctness results still hold. In our faster algorithm, T is $O(E\alpha(E))$, obtaining our desired almost-linear complexity bound.

4.1 Analysis of Initial Algorithm

THEOREM 4.1.1. *The algorithm of Figure 3 is $O(N + E + T)$, where N is the number of nodes and E the number of edges in the input flowgraph, and T is the total time for all calls to FindSnode.*

PROOF. The algorithm consists of

—an initialization phase,

—a phase where *Visit* is called recursively once for each node, and

—a call to *Finish*.

We analyze the complexity of each of these phases. The initialization phase is $O(N)$. For each call *Visit*(Z), there is a constant amount of work not inside any loop, the loop starting at step $\boxed{10}$ over predecessor edges into Z, and the loop starting at step $\boxed{16}$ where the contents of the stack are popped. The constant amount of work can be ignored in determining the bound. Consider the loop starting at step $\boxed{10}$. Since *Visit* is called only once for each node, over all calls to *Visit*, this loop is executed $O(E)$ times. Thus over all calls to *Visit*, the loop will execute at most $O(E)$ calls to *FindSnode* in step $\boxed{11}$, and $O(E)$ other work. For the last loop starting at step $\boxed{16}$ in *Visit*, all nodes are pushed and popped exactly once, so this loop is $O(N)$. Finally, consider the call to *Finish*. It consists of $O(E)$ work plus at most $O(E)$ calls to *FindSnode*. Summing all of this work, we obtain the desired result. □

We now analyze the asymptotic behavior of the function *FindSnode*(Y, P) as shown in Figure 3. Each invocation could require visiting each node on a dominator tree path from `Entry` to Y. The cost of applying *FindSnode*() to each of N flow graph nodes is then $O(N^2)$. As such, the overall asymptotic behavior of the simple version of our algorithm is $O(E \times N)$.

4.2 Faster Algorithm

Using a *path-compression* result due to Tarjan [1979], we rewrite certain parts of our algorithm to use the instructions:

Eval(Y). Using the links established by the *Link*() instruction, *Eval*() ascends the dominator tree from Y, returning the node of maximum label. The label associated initially with each node X is $-dfn(X)$; the link of each node is initially $\perp$.

Link$(Y, idom(Y))$. Sets $link(Y) = idom(Y)$. Any *Eval*() search that includes node Y now also includes the immediate dominator of Y.

Update$(X, dfn(X))$. Changes the label associated with X to $dfn(X)$. This instruction is issued when node X becomes included in set $\mathcal{N}_\sigma$.

The path-compressing version of algorithm is obtained as follows:

(1) We initialize the path-compression at steps $\boxed{20}$ and $\boxed{23}$.
(2) Links are inserted to extend the *Eval*() search at steps $\boxed{21}$ and $\boxed{24}$.
(3) Path information is updated whenever a node X is added to the sparse graph, at step $\boxed{26}$.
(4) We redefine function *FindSnode*() by:

```
Function $FindSnode(Y, P)$ : $node$
    return ($Map(Eval(Y))$)                    $\Leftarrow \boxed{27}$
end
```

We now state and prove the required properties of the path-compressing version of our algorithm.

Proper Use of Path-Compressing Instructions. We first show that the above instructions are used in a manner consistent with their definition [Tarjan 1979]: operations *Link*() and *Update*() are applied to nodes at the end of a "link-path."

LEMMA 4.2.1. *At steps* $\boxed{22}$ *and* $\boxed{25}$, *link*$(Z) = \perp$ *prior to invoking Link*().

PROOF. Steps $\boxed{20}$ and $\boxed{23}$ initialize $link(X) = \perp$ for each node $X \in \mathcal{N}_f$. Since each node Z has a unique immediate dominator in $idom(Z)$, and a unique map representative in $Map(Z)$, and since n is a strictly decreasing sequence at steps $\boxed{21}$ and $\boxed{24}$, $link(Z) = \perp$ just prior to applying *Link*() at steps $\boxed{22}$ and $\boxed{25}$. □

```
    $NodeCount \leftarrow 0$
    foreach $(X \in \mathcal{N}_f)$ do
        $\Upsilon(X) \leftarrow$ false ;   $\Phi(X) \leftarrow$ false ;   $Map(X) \leftarrow X$
    od
    foreach $(X \in \mathcal{N}_f)$ do                                  $\Leftarrow \boxed{20}$
        $link(X) \leftarrow \perp$ ;   $Label(X) \leftarrow -dfn(X)$
    od
    $Cnum \leftarrow 0$
    for $n = N$ downto 1 do
        foreach $(\{ Z \mid dfn(idom(Z)) = n \})$ do
            if $(Cnum(Z) = 0)$ then   call $Visit(Z)$   fi
        od
        foreach $(\{ Z \mid dfn(idom(Z)) = n \})$ do                   $\Leftarrow \boxed{21}$
            if $(Z = Map(Z))$ then                                      $\Leftarrow \boxed{22}$
                call $Link(Z, idom(Z))$
            else
                call $Link(Z, Map(Z))$
            fi
        od
    od
    foreach $(X \in \mathcal{N}_f)$ do                                  $\Leftarrow \boxed{23}$
        $link(X) \leftarrow \perp$ ;   $Label(X) \leftarrow -dfn(X)$
    od
    for $n = N$ downto 1 do
        foreach $(\{ Z \mid dfn(idom(Z)) = n \})$ do
            call $Finish(Z)$
        od
        foreach $(\{ Z \mid dfn(idom(Z)) = n \})$ do                   $\Leftarrow \boxed{24}$
            if $(Z = Map(Z))$ then                                      $\Leftarrow \boxed{25}$
                call $Link(Z, idom(Z))$
            else
                call $Link(Z, Map(Z))$
            fi
        od
    od

Procedure $IncludeNode(X)$
    $\Upsilon(X) \leftarrow$ true
    call $Update(X, dfn(X))$                                            $\Leftarrow \boxed{26}$
end
```

Fig. 5. Faster algorithm.

LEMMA 4.2.2. *When $Update(X, dfn(X))$ is invoked at step $\boxed{26}$, $link(X) = \perp$.*

PROOF. The procedure *IncludeNode*() is invoked only from *Visit*(), which processes only equidominates. Since step $\boxed{21}$ has not yet executed for any node considered by *Visit*(), each such node has $\perp$ for its link. □

Correctness of the Faster Algorithm. We now show that the path-compressing version produces the same output as its slower version.

LEMMA 4.2.3. *As invoked during the course of their respective algorithms, each implementation of $FindSnode(Y, P)$ returns the same answers.*

PROOF.

—By inspection, $FindSnode(Y, P)$ of Figure 3 begins at node Y and considers each ancestor X of Y, up to but excluding node P. As each node X is considered, the function returns $Map(X)$ if $Map(X)$ is already included in the sparse graph. If no $Map(X)$ is already in the sparse graph, then the function returns $Map(X)$, where X is ancestor of Y just prior to P.

—We now argue that $FindSnode(Y, P)$ at $\boxed{27}$ simulates exactly this behavior. First, notice that the path of links established at steps $\boxed{22}$ and $\boxed{25}$ link each node X to $Map(X)$ if $X \neq Map(X)$, and otherwise link each node X to $idom(X)$. Thus, strongly connected nodes are linked to their representative member, while that member is linked to its dominator. Each node X's label begins as $-dfn(X)$, but can be changed by step $\boxed{26}$ to be $dfn(X)$. There are two cases to consider:

(1) If $Map(X)$ is in the sparse graph, for any node X between Y and P (excluding P), then
   (a) there is a link path to that node, and
   (b) its label has been changed to $dfn(Map(X))$.

   $Eval(Y)$ returns the node of maximum label on the link path from Y, up to but excluding node P, since P has not been linked in yet. Since any node is depth-first numbered higher than its immediate dominator, $Eval(Y)$ returns some node in the strongly connected component closest to Y whose representative node is already in the sparse graph. Applying $Map()$ once again ensures that the representative node is returned.

(2) If no node on the link path is included in the sparse graph, then each such node must be labeled by its negative depth-first number. Thus, when $Eval(Y)$ returns the node of maximum label, this will be the node of minimum negative label, which will be some node in the strongly connected component containing the ancestor of Y just below P. Applying $Map()$ once again ensures that the representative node is returned.

Thus, the two versions of $FindSnode(Y, P)$ compute the same results. □

Performance. The asymptotic complexity of the path-compressing version of our algorithm is almost linear.

THEOREM 4.2.4. *Our faster algorithm takes $O(E\alpha(E))$ time.*

PROOF. There are $O(E)$ calls to $FindSnode()$. The proof thus follows from Theorem 4.1.1 and Tarjan [1979]. □

5. EXPERIMENTS

Although we have described flow graphs where the worst-case quadratic behavior of the standard algorithms does occur, previous experiments [Cytron et al. 1991] have indicated that this behavior is not expected in practice on real programs. In this section, we present results from experiments intended to answer the following questions:

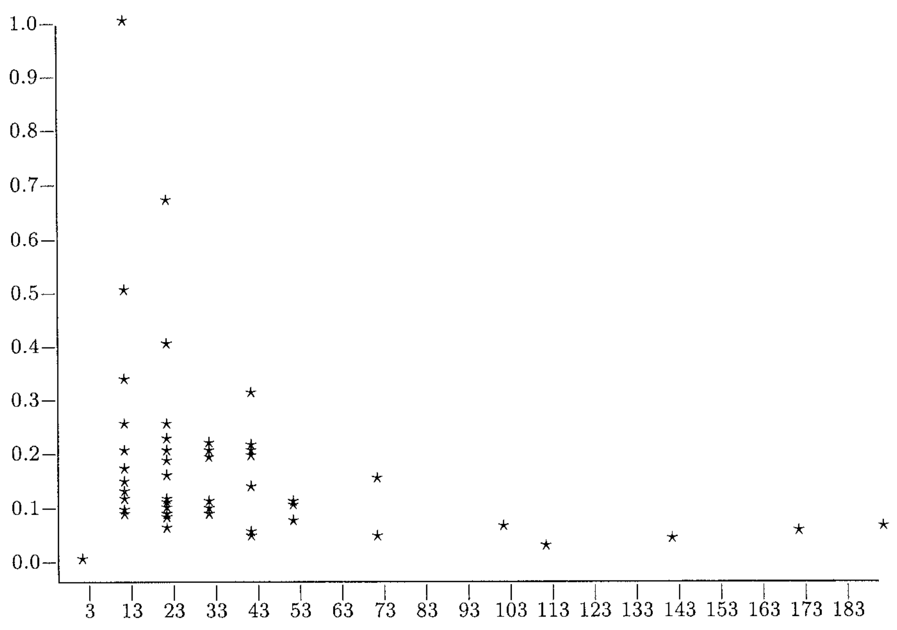

Fig. 6. Speedup of our algorithm vs. execution time (in milliseconds) of the usual algorithm

(1) How does the performance of our algorithm compare with the performance of the potentially quadratic algorithm [Cytron et al. 1991] on typical programs?

(2) How big must a contrived example become before our algorithm beats the quadratic algorithm?

We performed an experiment, wherein ϕ-functions were placed (toward construction of SSA form) in 139 Fortran procedures taken from the Perfect [Berry et al. 1988] (*Ocean*, *Spice*, *QCD*) benchmark suite and from the Eispack [Smith et al. 1976] and Linpack [Dongarra et al. 1979] subroutine library; these are the same procedures that participated in the experiments reported in Cytron et al. [1991]. In Figure 6 we compare the speed of our simple (i.e., without balanced path-compression) but asymptotically faster algorithm given in Section 2 with the speed of the usual algorithm [Cytron et al. 1991]; these execution times were obtained on a SparcStation 10, and they represent only the time necessary to compute the location of ϕ-functions.

Most of the runs show that the execution time of our algorithm is linear (with a small constant) in the execution time of the usual algorithm, and so can be expected also to exhibit linear behavior in practice on the same programs. Because each of these runs took under 1 second, both algorithms are fast on this collection of programs. We have not implemented the balanced path-compression, and so these experiments did not reflect any improvements that might be gained by this theoretically more efficient algorithm.

Though not represented in Figure 6, we also experimented with a series of increasingly taller "ladder" graphs of the form shown in Figure 1, where worst-case behavior is expected. Our algorithm demonstrated better performance when these ladder graphs had 10 or more rungs, though smaller graphs take scant execution time anyway. Our algorithm is twice as fast as the usual algorithm for a ladder graph of 75 rungs, taking 20 milliseconds while the usual algorithm took 40 milliseconds.

In summary, comparison of our simple algorithm to the usual algorithm shows

—linear performance for the same cases as the usual algorithm, although our median test case exhibited performance degradation of a factor of 3;

—a factor-of-2 better performance for some artificially generated cases.

Thus, preliminary evidence indicates comparable expected performance using our simple algorithm.

6. CONCLUSIONS AND FUTURE WORK

In this article we have shown how to eliminate the potentially costly step of computing dominance frontiers when constructing Sparse Evaluation Graphs or SSA form. By directly determining the conditions under which a node is a ϕ-node, rather than by iterating through the dominance frontiers, we obtain a worst-case almost-linear bound for constructing SEGs and a worst-case almost-quadratic bound for constructing SSA form. In both cases, we have eliminated the $O(N^2)$ behavior associated with computing and using dominance frontiers. We have also given preliminary experimental evidence that our simple algorithm's behavior, though slower in many cases, is comparable in practice to the usual algorithm.

Future work will incorporate balanced path-compression into our simple algorithm and will compare the results on real and artificially generated cases.

Recently, Sreedhar and Gao [1994] have developed an elegant algorithm that determines ϕ-nodes in linear time. Their algorithm uses some of the graph properties developed in this article, but introduces a new structure (the "DJ-graph") that assists in determining ϕ nodes in the "forward" manner (given node X, include nodes in $DF(X)$) of the original ϕ-node placement algorithm [Cytron et al. 1991]. Our algorithm instead computes whether a given node Y is in the dominance frontier of any node X already included in the graph. The Sreedhar and Gao algorithm is asymptotically faster and remarkably simple; however, our backward style of determining ϕ-nodes may be more useful in some situations, as when SSA form is updated incrementally in response to inclusion of *may-alias* information [Cytron and Gershbein 1993].

Acknowledgements

We thank Dirk Grunwald and Harini Srinivasan for their suggestions with a preliminary version. We are grateful to our referees, especially the one who found a mistake in our proofs.

REFERENCES

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

ALPERN, B., WEGMAN, M. N., AND ZADECK, F. K. 1988. Detecting equality of variables in programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 1–11.

BERRY, M., CHEN, D., KOSS, P., KUCK, D., LO, S., PANG, Y., ROLOFF, R., SAMEH, A., CLEMENTI, E., CHIN, S., SCHNEIDER, D., FOX, G., MESSINA, P., WALKER, D., HSIUNG, C., SCHWARZMEIER, J., LUE, K., ORSZAG, S., SEIDL, F., JOHNSON, O., SWANSON, G., GOODRUM, R., AND MARTIN, J. 1988. The perfect club benchmarks: Effective performance evaluation of supercomputers. Tech. Rep. 827, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, Ill.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 232–245.

CHOI, J.-D., CYTRON, R., AND FERRANTE, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 55–66.

CHOI, J.-D., CYTRON, R., AND FERRANTE, J. 1994. On the efficient engineering of ambitious program analysis. *IEEE Trans. Softw. Eng. 20,* 2, 105–114.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass.

CYTRON, R. AND GERSHBEIN, R. 1993. Efficiently accommodating may-alias information in SSA form. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* ACM, New York, 36–45.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4 (Oct.), 451–490.

CYTRON, R., LOWRY, A., AND ZADECK, K. 1986. Code motion of control structures in high-level languages. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 70–85.

DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., AND STEWART, G. W. 1979. *Linpack Users' Guide.* SIAM Press, Philadelphia, Pa.

JOHNSON, R. AND PINGALI, K. 1993. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 78–89.

MARLOWE, T. J. 1989. Data flow analysis and incremental iteration. Ph. D. thesis, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J.

ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1988. Global value numbers and redundant computations. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 12–27.

SMITH, B. T., BOYLE, J. M., DONGARRA, J. J., GARBOW, B. S., IKEBE, Y., KLEMA, V. C., AND MOLER, C. B. 1976. *Matrix Eigensystem Routines—Eispack Guide.* Springer-Verlag, Berlin.

SREEDHAR, V. AND GAO, G. 1994. Computing ϕ-nodes in linear time using DJ-graphs. Tech. Rep. ACAPS Memo 75, School of Computer Science, McGill Univ., Montreal, Quebec, Canada.

TARJAN, R. 1979. Applications of path compression on balanced trees. *J. ACM 26,* 4 (Oct.), 690–715.

WEGMAN, M. N. AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13,* 2 (Apr.), 181–210.

Received October 1993; revised October 1994; accepted November 1994