View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-93-24

1993-01-01

Efficiently Computing {phi}-Nodes On-The-Fly

Ron K. Cytron and Jeanne Ferrante

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 {phi}-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 paper we present an almost-linear algorithm for detemining exactly the same set of {phi}-nodes. ... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Cytron, Ron K. and Ferrante, Jeanne, "Efficiently Computing {phi}-Nodes On-The-Fly" Report Number: WUCS-93-24 (1993). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/312

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/312

Efficiently Computing {phi}-Nodes On-The-Fly

Ron K. Cytron and Jeanne Ferrante

Complete Abstract:

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 {phi}-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 paper we present an almost-linear algorithm for detemining exactly the same set of {phi}-nodes.

Efficiently Computing ϕ-Nodes On-The-Fly

Ron K. Cytron and Jeanne Ferrante

WUCS-93-24

October 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

Efficiently Computing ϕ-Nodes On-The-Fly

Ron K. Cytron[1] and Jeanne Ferrante[2]

[1] Department of Computer Science
Washington University Box 1045
St. Louis, MO 63130-4899
(Email: cytron@cs.wustl.edu)
[2] T. J. Watson Research Center
IBM Research PO Box 704
Yorktown Heights, NY 10598
(Email: ferrant@watson.ibm.com)

Abstract. 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 paper we present an *almost-linear* algorithm for determining exactly the same set of ϕ-nodes.

1 Motivation and Background

Static Single Assignment (SSA) form [9] and the more general Sparse Evaluation Graphs (SEG) [6] have emerged as an efficient mechanism for solving compile-time optimization problems via data flow analysis [18, 10, 3, 15, 7, 5]. 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 references that are relevant to solving certain data flow problems.

The SEG and SSA algorithms process and compute the following information:

Inputs:

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

Depth-first numbering where $dfn(X)$ is the number associated with X

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

in a depth-first search of $\mathcal{G}_f$.[3] We call the associated depth-first spanning tree $DFST$. Similarly, we define the inverse mapping

$$vertex(k) = X \mid dfn(X) = k$$

Dominator tree where $idom(X)$ is the immediate dominator of flow graph node X. We say that X *dominates* Y, written $X \gg Y$, 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 $X \ggeq Y$ and $X \neq Y$. Each node X has a unique *immediate dominator* $idom(X)$ such that

$$idom(X) \gg X \text{ and } \forall\ W \gg X,\ W \ggeq idom(X)$$

Node $idom(X)$ 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 non-identity transference in a data flow framework. For SSA, such nodes contain *definitions* of variables.

Outputs:

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:

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

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 $\mathcal{N}_\alpha \subseteq \mathcal{N}_f$.
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 $\mathcal{N}_\phi = \{\, W, X, Y, Z\,\}$.

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 $\mathcal{N}_\phi$.

[3] Here, $dfn(X)$ is assigned in order of nodes visited, starting with 1; in [2], depth first numbers are assigned starting from $|\mathcal{N}_f|$ down to 1.

5. In SEG, appropriate edges are then placed between nodes in

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

In SSA, 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 [9],

1. Constructing a single SEG (i.e., one data flow framework) by the usual algorithm [6] takes $O(E + N^2)$ time;
2. Where a data flow problem can be partitioned into V disjoint frameworks [14], 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 [8]. 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 [9], 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[4], 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 paper 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

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

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 describe an algorithm to construct an SSA-like representation [13] 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.

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 [9] is biased toward the *average* case, in which linear behavior for constructing or consulting dominance frontiers is expected. The algorithm we present in this paper, and the algorithm due to Johnson and Pingali [13], are $\Omega(EV)$, since each edge in the flow graph is examined for each variable.

We actually present two variations of the same algorithm in this paper. The first "simple" algorithm is presented for expository reasons; the second algorithm uses balanced path-compression to achieve our improved time bound. Preliminary experiments presented in Section 5 compare the simple algorithm presented in Section 2 with the usual dominance frontier-based algorithm [9]. 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

Definition 1. *The* equidominates *of a node X are 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.

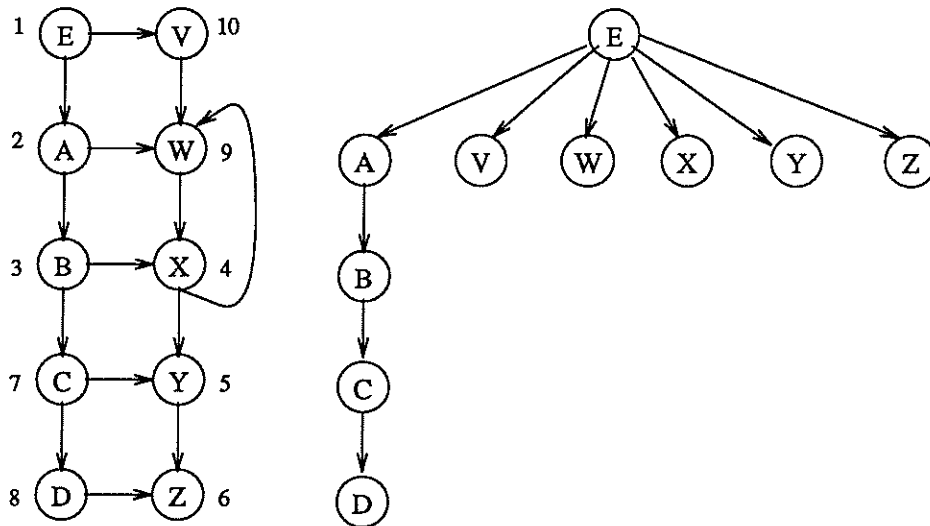

Fig. 1. Flow graph and its dominator tree.

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.

Our algorithm essentially partitions equidominates into blocks of nodes that are in each other's iterated dominance frontiers, but without the expense of explicitly computing dominance frontiers. The relevant portions of our algorithm refer to $Cnum(X)$, $LL(X)$, or $Map(X)$; these are paraphrases of a *strongly-connected component* (SCC) algorithm [1], which we use to find (some) strongly-connected components of the dominance frontier graph (in which an edge from Y to Z implies $Z \in DF(Y)$). While the details of $Cnum(X)$ and $LL(X)$ are beyond the scope of this paper [1], we note that $Map(X)$ is the representative node for

Algorithm[1] Sparse graph node determination (simple)

```
NodeCount ← 0
foreach (X ∈ N_f) do
    Υ(X) ← false
    Φ(X) ← false
    Map(X) ← X
od
Cnum ← 0
for n = N downto 1 do                                   ⇐ [1]
    foreach ({ Z | dfn(idom(Z)) = n }) do               ⇐ [2]
        if (Cnum(Z) = 0) then
            call Visit(Z)
        fi
    od
od
                                                        ⇐ [3]
for n = N downto 1 do                                   ⇐ [4]
    foreach ({ Z | dfn(idom(Z)) = n }) do               ⇐ [5]
        call Finish(Z)
    od
od
end
```

```
Function FindSnode(Y, P) : node
    for (X = Y) repeat(X = idom(X)) do
        if (Υ(Map(X)) or (idom(X) = P)) then
            return (Map(X))
        fi
    od
end
```

Fig. 3.

the SCC containing node X. Throughout this paper, "SCC" or "component" refers to nodes associated in this manner.

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

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

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

```
Procedure Visit(Z)
   LL(Z) ← Cnum(Z) ← ++NodeCount
   call push(Z)                                                  ⇐ [6]
   foreach ((Y, Z) ∈ E_f | Y ≠ idom(Z)) do                       ⇐ [7]
      s ← FindSnode(Y, idom(Z))                                  ⇐ [8]
      if (idom(s) ≠ idom(Z)) then
         call IncludeNode(Z)                                     ⇐ [9]
      else
         if (Cnum(s) = 0) then
            call Visit(s)                                        ⇐ [10]
            LL(Z) ← min(LL(Z), LL(s))
         else
            if ((Cnum(s) < Cnum(Z)) and OnStack(s)) then
               LL(Z) ← min(LL(Z), Cnum(s))
            fi
         fi
         if (Υ(s) and not OnStack(s)) then                       ⇐ [11]
            call IncludeNode(Z)                                  ⇐ [12]
         fi
      fi
   od
   if (LL(Z) = Cnum(Z)) then
      repeat                                                     ⇐ [13]
         Q ← pop()                                               ⇐ [14]
         Map(Q) ← Z
         if (Q ∈ N_α) then
            call IncludeNode(Q)                                  ⇐ [15]
         fi
         if (Υ(Q)) then
            call IncludeNode(Z)                                  ⇐ [16]
         fi
      until (Q = Z)
   fi
end
```

Fig. 4.

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, 4, and 5. 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 cor-

```
Procedure IncludeNode(X)
    ϒ(X) ← true                                          ⇐ [17]
end

Procedure finish(Z)
    foreach ((Y, Z) ∈ ℰ_f | Y ≠ idom(Z)) do
        s ← FindSnode(Y, idom(Z))
        if (ϒ(s)) then
            Φ(Z) ← true                                  ⇐ [18]
        fi
    od
    if (ϒ(Map(Z))) then
        call IncludeNode(Z)                              ⇐ [19]
    fi
end
```

Fig. 5.

rectly determined to be in $\mathcal{N}_\sigma$, the algorithm can correctly determine this property for nodes of lower depth first number.

The loop at [1] begins with V (depth-first numbered 10 in Figure 1), but since no nodes are dominated by V, no steps are taken by [2]; the same holds for nodes W and D. When [1] considers C, loop [2] calls *Visit* with D. The loop in *Visit* at [7] is empty, since the only predecessor of D immediately dominates D. Thus, $Map(D)$ is set to D in loop [13]. Loop [1] then considers in turn Z, Y, and X, each of which dominates no node, so [2] is empty. When [1] considers B, *Visit* is called on C, but since C's only predecessor immediately dominates C, no action is taken by [7], 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 [15] places B in $\mathcal{N}_\sigma$.

When [1] considers E, suppose loop [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 [7] 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 [9].
- For X, *FindSnode* returns X; since X and Y are equidominates, *Visit* is called recursively on X. In this invocation, loop [7] 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 [7] 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 [2] considers W, X, and V, each has already been visited. When *Visit* is called for Z, nodes D and Y are considered by [7].

- 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 [9, 6] 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

We begin with some preliminary lemmas.

Lemma 2.

$$(Y, Z) \in \mathcal{E}_f \implies idom(Z) \gg\!\!= Y$$

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 doesn't contain $idom(Z)$. Thus, $(Y, Z) \notin \mathcal{E}_f$
□

Corollary 3.

$$(Y, Z) \in \mathcal{E}_f \text{ and } Y \neq idom(Z) \implies$$
$$dfn(idom(Z)) \leq dfn(idom(Y)) < dfn(Y)$$

Corollary 4.

$$(Y, Z) \in \mathcal{E}_f \text{ and } Y \neq idom(Z) \implies$$
$$idom(Z) \gg\!\!= idom(Y) \gg Y$$

Lemma 5. *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 6. *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 $X \geqslant\!\!\geqslant Y$ 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 4, $idom(Z) \geqslant\!\!\geqslant idom(Y) \gg Y$. By Lemma 5, $X \gg Y \Rightarrow X \gg Z$. But since $X \gg Y$, we have $X \gg Z$, a contradiction. Hence $Z \notin DF(X)$.
□

Theorem 7.

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

Proof: Suppose $dfn(X) \leq dfn(idom(Z))$. In the DFST of $\mathcal{G}_f$, either

1. X is an ancestor of $idom(Z)$. By Theorem 6,

$$Z \notin DF(X)$$

   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 $Z \notin DF(X)$.

□

Theorem 8.

$$Z \in DF(X) \Longrightarrow idom(Z) \gg X$$

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 doesn't 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 $X \geqslant\!\!\geqslant Y$ and $X \not\gg Z$. With $X \geqslant\!\!\geqslant Y$, we can construct Q such that edges between X and Y are DFST edges. Since $idom(Z)$ doesn't 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 6, $Z \notin DF(X)$.

□

Corollary 9.

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

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

Lemma 10. *Visit is called exactly once for each node in Flowgraph.*

Lemma 11. *During all calls to $Visit$, each node is pushed and popped exactly once.*

Proof: By Lemma 10, $Visit(Z)$ is invoked exactly once, and 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 12.

$$Y = Map(X) \Longrightarrow idom(Y) = idom(X)$$

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 $\boxed{13}$, when equidominates are popped off the stack. In this case, we have $idom(Map(X)) = idom(X)$.
□

Lemma 13. *At step $\boxed{12}$, node s (referenced at step $\boxed{11}$) has already been pushed and popped.*

Proof: By the predicate of step $\boxed{11}$, node s cannot be on stack at step $\boxed{12}$. Thus, s either hasn't been pushed yet, or else s has been pushed and popped. If s hasn't been pushed then $Cnum(s) = 0$, but then step $\boxed{10}$ would have pushed s before step $\boxed{12}$ is reached. Therefore, s has been pushed and popped.
□

Corollary 14. *Any invocation of $IncludeNode(s)$ must already have occurred at step $\boxed{12}$.*

Lemma 15.

$$OnStack(X) \text{ and } OnStack(Y) \Longrightarrow idom(X) = idom(Y)$$

Lemma 16. *At step $\boxed{8}$, $FindSnode()$ returns*

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

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

Lemma 17.

$$Map(X) = Map(Y) \Longrightarrow Y \in DF^{+}(X) \text{ or } X = Y$$

Proof: Follows from initialization ($Map(X) = X$), Lemma 16, and the observation that $Map(X)$ represents the strongly-connected component containing X. □

Corollary 18.

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

Theorem 19. *As of step* [3],

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

Proof:

$\Longrightarrow$ We actually prove a stronger result:

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

To accommodate the iteration of step [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 [2] empty, this case is trivially satisfied.

Inductive step: Consider those steps that potentially set

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

Step [9]: With $idom(s) \neq idom(Z)$, step [8] must have returned

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

From Lemma 16 we obtain

$$Z \in DF^{+}(s)$$

From Lemma 12 and with $idom(s) \neq idom(Z)$, Corollary 9 implies

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

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

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

Thus,

$$Z \in \mathcal{N}_\sigma$$

Steps [12], [15], and [16]: 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 [6] and [14]). 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 15 ensures that steps [15] and [16] cannot affect any of the x_i. By Lemma 13, step [12] requires s to be an already popped x_i, so step [12] cannot affect any of the x_i.

Inductive step: We now prove $IH_B(n)$.

Step [12]: By Lemma 13, s has already been pushed and popped. By Corollary 14, and assuming

$$IH_B(k) \mid 1 \le k \le n-1$$

we obtain

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

Thus,

$$Z \in \mathcal{N}_\sigma$$

Step [15] By definition,

$$Q \in \mathcal{N}_\alpha \Longrightarrow Q \in \mathcal{N}_\sigma$$

Step [16] 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 17, $Z \in DF^+(Q)$, 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:

$$\mathcal{DF}^0(\mathcal{N}_\alpha) = Map(\mathcal{N}_\alpha)$$
$$\mathcal{DF}^i(\mathcal{N}_\alpha) = Map(DF(\mathcal{DF}^{i-1}(\mathcal{N}_\alpha)))$$

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 [6] and [14]. At step [15],

$$y \in \mathcal{N}_\alpha \Longrightarrow \Upsilon(y)$$

At step [16],

$$Y \in Map(\mathcal{N}_\alpha) \Longrightarrow \Upsilon(Y)$$

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 12

$$idom(Y) = idom(y)$$

and so Corollary 9 can be extended to:

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

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 [8] will return X. There are two cases:

$dfn(idom(Y)) < dfn(idom(X))$: Step [9] sets $\Upsilon(y)$ true. When y is popped at step [14], step [16] sets $\Upsilon(Y)$ true.

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

$X \neq Y$: Step [12] sets $\Upsilon(y)$ true. When y is popped at step [14], step [16] sets $\Upsilon(Y)$ true.
$X = Y$: then $\Upsilon(X) \Longrightarrow \Upsilon(Y)$.

□

Theorem 20. *After the call to* finish,

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

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 19. 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 21. *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 18). But

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

(by Theorem 19). 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 22. *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 [7] over predecessor edges into Z, and the loop starting at step [13] 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 [7]. 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 [8], and $O(E)$ other work. For the last loop starting at step [13] 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 [17], we rewrite certain parts of our algorithm (shown in Figures 6 and 7) 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 initially associated 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 [20] and [24].

```
Procedure Initialize
    NodeCount ← 0
    foreach (X ∈ N_f) do
        Υ(X) ← false
        Φ(X) ← false
        Map(X) ← X
    od
    foreach (X ∈ N_f) do                          ⇐ [20]
        link(X) ← ⊥
        Label(X) ← −dfn(X)
    od
    Cnum ← 0
end
Procedure IncludeNode(X)
    Υ(X) ← true
    call Update(X, dfn(X))                        ⇐ [21]
end
```

Fig. 6.

2. Links are inserted to extend the *Eval*() search at steps [22] and [25].
3. Path information is updated whenever a node X is added to the sparse graph, at step [21].
4. We redefine function *FindSnode*() by:

```
Function FindSnode(Y, P) : node
    return (Map(Eval(Y)))                         ⇐ [27]
end
```

We must now show:

1. We use the above instructions in a manner consistent with their definition [17]: operations *Link*() and *Update*() are applied to nodes at the end of a "link-path".

Lemma 23. *At steps [23] and [26], $link(Z) = \perp$ prior to invoking Link().*

Proof: Steps [20] and [24] 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 [22] and [25], $link(Z) = \perp$ just prior to applying *Link*() at steps [23] and [26]. □

Lemma 24. *When $Update(X, dfn(X))$ is invoked at step [21], $link(X) = \perp$.*

```
call Initialize
for n = N downto 1 do
    foreach ({ Z | dfn(idom(Z)) = n }) do
        if (Cnum(Z) = 0) then
            call Visit(Z)
        fi
    od
    foreach ({ Z | dfn(idom(Z)) = n }) do                    ⇐ [22]
        if (Z = Map(Z)) then                                 ⇐ [23]
            call Link(Z, idom(Z))
        else
            call Link(Z, Map(Z))
        fi
    od
od
foreach (X ∈ N_f) do                                         ⇐ [24]
    link(X) ← ⊥
    Label(X) ← − dfn(X)
od
for n = N downto 1 do
    foreach ({ Z | dfn(idom(Z)) = n }) do
        call Finish(Z)
    od
    foreach ({ Z | dfn(idom(Z)) = n }) do                    ⇐ [25]
        if (Z = Map(Z)) then                                 ⇐ [26]
            call Link(Z, idom(Z))
        else
            call Link(Z, Map(Z))
        fi
    od
od
```

Fig. 7.

Proof: The procedure *IncludeNode*() is invoked only from *Visit*(), which processes only equidominates. Since step [22] has not yet executed for any node considered by *Visit*(), each such node has $\perp$ for its link. □

2. Use of these instructions does not change the output of our algorithm.

Lemma 25. *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 [27] simulates exactly this behavior. First, notice that the path of links established at steps [23] and [26] 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 [21] to be $dfn(X)$. There are two cases to consider:
  (a) If $Map(X)$ is in the sparse graph, for any node X between Y and P (excluding P), then there is a link path to that node and 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.
  (b) 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.

□

Theorem 26. *Our faster algorithm takes $O(E\alpha(E))$ time.*

Proof: Follows from $O(E)$ calls to $FindSnode()$, Theorem 22, and Tarjan's result [17]. □

5 Preliminary Experiments

Although we have described flow graphs where the worst case, quadratic behavior of the standard algorithms does occur, previous experiments [9] have indicated this behavior is not expected in practice on real programs. We performed an experiment, wherein ϕ-functions were placed (toward construction of SSA form) in 139 Fortran procedures taken from the Perfect [4] (*Ocean*, *Spice*, *QCD*) benchmark suite and from the Eispack [16] and Linpack [12] subroutine library. In Figure 8 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 [9]; these execution times were obtained on a SparcStation 10, and they represent only the time necessary to compute the location of ϕ-functions.

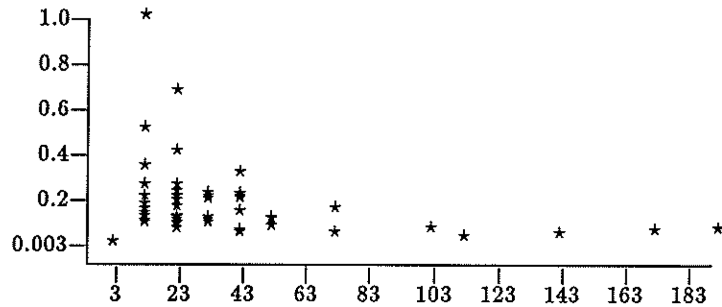

Fig. 8. Speedup of our algorithm vs. execution time (in milliseconds) of usual ϕ-placing algorithm

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 to also 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 8, 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 paper 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 SEG's 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 compare the results on real and artificially generated cases.

Acknowledgements

We thank Dirk Grunwald and Harini Srinivasan for their suggestions as we began work on this paper. We are grateful to those who reviewed a preliminary version of this paper [11], especially the reviewer who found a mistake in one of our proofs.

References

1. Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
3. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. *Fifteenth ACM Principles of Programming Languages Symposium,* pages 1–11, January 1988. San Diego, CA.
4. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The perfect club benchmarks: Effective performance evaluation of supercomputers the performance evaluation club (perfect). Technical report, U. of Ill–Center for Supercomputing Research and Development, November 1988.
5. Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages,* January 1993.
6. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages,* January 1991.
7. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient treatment of preserving definitions. Technical report, IBM Research, 1991. Research Report RC17065.

8. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.
9. 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,* October 1991.
10. Ron Cytron, Andy Lowry, and Ken Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. of the ACM Symp. on Principles of Compiler Construction,* 1986.
11. Ron K. Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly (extended abstract). *Proceedings of the 1993 Workshop on Languages and Compilers for Parallelism,* 1993.
12. J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *Linpack Users' Guide.* SIAM Press, 1979.
13. Richard Johnson and Keshav Pingali. Dependence-based program analysis. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Langauge Design and Implementation,* 1993. Published as SIGPLAN Notices Volume 28 Number 6.
14. Thomas J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, October 1989.
15. Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. *Fifteenth ACM Principles of Programming Languages Symposium,* pages 12–27, January 1988. San Diego, CA.
16. B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – Eispack Guide.* Springer-Verlag, 1976.
17. Robert Tarjan. Applications of path compression on balanced trees. *JACM,* 26(4):690–715, October 1979.
18. Mark Wegman and Ken Zadeck. Constant propagation with conditional branches. *Conf. Rec. Twelfth ACM Symposium on Principles of Programming Languages,* pages 291–299, January 1985.

This article was processed using the LaTeX macro package with LLNCS style