Algorithms for Computing the Static Single Assignment Form

Gianfranco Bilardi
Dipartimento di Ingegneria dell'Informazione
Università di Padova, 35131 Padova, Italy

Keshav Pingali
Department of Computer Science
Cornell University, Ithaca, NY 14853

The Static Single Assignment (SSA) form is a program representation used in many optimizing compilers. The key step in converting a program to SSA form is called ϕ-placement. Many algorithms for ϕ-placement have been proposed in the literature, but the relationships between these algorithms are not well understood.

In this paper, we propose a framework within which we systematically derive (i) properties of the SSA form and (ii) ϕ-placement algorithms. This framework is based on a new relation called *merge* which captures succinctly the structure of a program's control flow graph that is relevant to its SSA form. The ϕ-placement algorithms we derive include most of the ones described in the literature, as well as several new ones. We also evaluate experimentally the performance of some of these algorithms on the SPEC95 benchmarks.

Some of the algorithms described here are optimal for a single variable. However, their repeated application is not necessarily optimal for multiple variables. We conclude the paper by describing such an optimal algorithm, based on the transitive reduction of the merge relation, for multi-variable ϕ-placement in structured programs. The problem for general programs remains open.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers and optimization*; I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Control dependence, optimizing compilers, program optimization, program transformation, static single assignment form

1. INTRODUCTION

Many program optimization algorithms become simpler and faster if programs are first transformed to *Static Single Assignment* (SSA) form [SS70; CFR$^+$91] in which

Gianfranco Bilardi (bilardi@dei.unipd.it) was supported in part by the Italian Ministry of University and Research and by the Italian National Research Council. Keshav Pingali (pingali@cs.cornell.edu) was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, and ACI-0121401.
Section 6 of this paper contains an extended and revised version of an algorithm that appeared in a PLDI'95 paper [PB95].

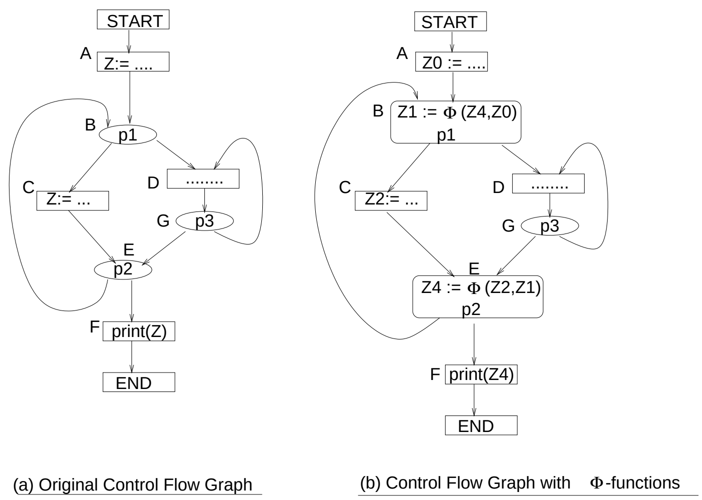


Fig. 1. A program and its SSA form

each use[1] of a variable is reached by a single definition of that variable. The conversion of a program to SSA form is accomplished by introducing *pseudo-assignments* at confluence points, *i.e.*, points with multiple predecessors, in the control flow graph (CFG) of the program. A pseudo-assignment for a variable Z is a statement of the form $Z = \phi(Z, Z, .., Z)$ where the ϕ-function on the right hand side has one argument for each incoming CFG edge at that confluence point. Intuitively, a ϕ-function at a confluence point in the *CFG merges* multiple definitions that reach that point. Each occurrence of Z on the right hand side of a ϕ-function is called a *pseudo-use* of Z. A convenient way to represent reaching definitions information after ϕ-placement is to rename the left hand side of every assignment and pseudo-assignment of Z to a unique variable, and use the new name at all uses and pseudo-uses reached by that assignment or pseudo-assignment. In the CFG of Figure 1(a), ϕ-functions for Z are placed at nodes B and E; the program after conversion to SSA form is shown in Figure 1(b). Note that no ϕ-function is needed at D, since the pseudo-assignment at B is the only assignment or pseudo-assignment of Z that reaches node D in the transformed program.

An SSA can be easily obtained by placing ϕ-functions for all variables at every confluence point in the CFG. In general, this approach introduces more ϕ-functions than necessary. For example, in Figure 1, an unnecessary ϕ-function for Z would be introduced at node D.

In this paper, we study the problem of transforming an arbitrary program into an equivalent SSA form by inserting ϕ-functions only where they are needed. A ϕ-function for variable Z is certainly required at a node v if assignments to variable Z occur along two non-empty paths $u \xrightarrow{+} v$ and $w \xrightarrow{+} v$ intersecting only at v. This observation suggests the following definition [CFR+91].

[1]Standard definitions of concepts like control flow graph, dominance, defs, uses, etc. can be found in the Appendix.

Definition 1.1. Given a CFG G=(V,E) and a set $S \subseteq V$ of its nodes such that $\texttt{START} \in S$, $J(S)$ is the set of all nodes v for which there are distinct nodes $u, w \in S$ such that there is a pair of paths $u \stackrel{+}{\to} v$ and $w \stackrel{+}{\to} v$, intersecting only at v. The set $J(S)$ is called the *join set of* S.

If S is the set of assignments to a variable Z, we see that we need pseudo-assignments to Z at least in the set of nodes $J(S)$. By considering the assignments in S and these pseudo-assignments in $J(S)$, we see that we might also need further pseudo-assignments in the nodes $J(S \cup J(S))$. However, as shown by Weiss [Weiss92] and proved in Section 2.3, $J(S \cup J(S)) = J(S)$. Hence, the ϕ-assignments in the nodes $J(S)$ are sufficient[2].

The need for J sets arises also in the computation of the *weak control dependence* relation [PC90], as shown in [BP96] and briefly reviewed in Section 5.1.1.

If several variables have to be processed, it may be efficient to preprocess the CFG and obtain a data structure that facilitates the construction of $J(S)$ for any given S. Therefore, the performance of a ϕ-placement algorithm is appropriately measured by the *preprocessing time* T_p and *preprocessing space* S_p used to build and store the data structure corresponding to G, and by the *query time* T_q used to obtain $J(S)$ from S, given the data structure. Then, the total time spent for ϕ-placement of all the variables is

$$T_{\phi-placement} = O(T_p + \sum_Z T_q(S_Z)). \tag{1}$$

Once the set $J(S_Z)$ has been determined for each variable Z of the program, the following *renaming* steps are necessary to achieve the desired SSA form. (i) For each $v \in S_Z \cup J(S_Z)$, rename the assignment to Z as an assignment to Z_v. (ii) For each $v \in J(S_Z)$, determine the arguments of the ϕ-assignment $Z_v = \phi(Z_{x_1}, \ldots, Z_{x_q})$. (iii) For each node $u \in U_Z$ where Z is used in the original program, replace Z by the appropriate Z_v. The above steps can be performed efficiently by an algorithm proposed in [CFR$^+$91]. This algorithm visits the CFG according to a top-down ordering of its dominator tree, and works in time

$$T_{renaming} = O(|V| + |E| + \sum_Z (|S_Z| + |J(S_Z)| + |U_Z|)). \tag{2}$$

Preprocessing time T_p is at least linear in the size $|V|+|E|$ of the program and query time $T_q(S_Z)$ is at least linear in the size of its input and output sets $(|S_Z|+|J(S_Z)|)$. Hence, assuming the number of uses $\sum_Z |U_Z|$ to be comparable with the number of definitions $\sum_Z |S_Z|$, we see that the main cost of SSA conversion is that of ϕ-placement. Therefore, the present paper focuses on ϕ-placement algorithms.

1.1 Summary of Prior Work

A number of algorithms for ϕ-placement have been proposed in the literature. An outline of an algorithm was given by Shapiro and Saint [SS70]. Reif and Tar-

[2]Formally, we are looking for the least set $\phi(S)$ (where pseudo-assignments must be placed) such that $J(S \cup \phi(S)) \subseteq \phi(S)$. If subsets of V are ordered by inclusion, the function J is monotonic. Therefore, $\phi(S)$ is the largest element of the sequence $\{\}, J(S), J(S \cup J(S)), \ldots$. Since $J(S \cup J(S)) = J(S)$, $\phi(S) = J(S)$.

jan extended the Lengauer and Tarjan dominator algorithm [LT79] to compute ϕ-placement for all variables in a bottom-up walk of the dominator tree [RT81]. Their algorithm takes $O(|E|\alpha(|E|))$ time per variable, but it is complicated because dominator computation is folded into ϕ-placement. Since dominator information is required for many compiler optimizations, it is worth separating its computation from ϕ-placement. Cytron *et al.* showed how this could be done using the idea of *dominance frontiers* [CFR+91]. Since the collective size of dominance frontier sets can grow as $\theta(|V|^2)$ even for structured programs, numerous attempts were made to improve this algorithm. An *on-the-fly* algorithm computing J sets in $O(|E|\alpha(|E|))$ time per variable was described by Cytron and Ferrante [CF93]; however, path compression and other complications made this procedure not competitive with the Cytron *et al.* algorithm, in practice. An algorithm by Johnson and Pingali, based on the dependence flow graph [PBJ+91] and working in $O(|E|)$ time per variable, was not competitive in practice either [JP93]. Sreedhar and Gao described another approach which traversed the dominator tree of the program to compute J sets on demand [SG95]. This algorithm requires $O(|E|)$ preprocessing time, preprocessing space, and query time, and it is easy to implement, but it is not competitive with the Cytron *et al.* algorithm in practice, as we discuss in Section 7. The first algorithm with this asymptotic performance that is competitive in practice with the Cytron *et al.* algorithm was described by us in an earlier paper on optimal control dependence computation [PB95], and is named *lazy pushing* in this paper. Lazy pushing uses a data structure called the *augmented dominator tree* $\mathcal{ADT}$ with a parameter β that controls a particular space-time trade-off. The algorithms of Cytron *et al.* and of Sreedhar and Gao can be essentially viewed as special cases of lazy pushing, obtained for particular values of β.

1.2 Overview of Paper

This paper presents algorithms for ϕ-placement, some from the literature and some new ones, placing them in a framework where they can be compared, based both on the structural properties of the SSA form and on the algorithmic techniques being exploited [3].

In Section 2, we introduce a new relation called the *merge* relation M that holds between nodes v and w of the CFG whenever $v \in J(\{\texttt{START}, w\})$; that is, v is a ϕ-node for a variable assigned only at w and START. This is written as $(w, v) \in M$, or as $v \in M(w)$. Three key properties make M the cornerstone of SSA computation:

1. If $\{\texttt{START}\} \subseteq S \subseteq V$, then $J(S) = \cup_{w \in S} M(w)$.
2. $v \in M(w)$ if and only if there is a so-called M-path from w to v in the CFG (as defined later, an M-path from w to v is a path that does not contain any strict dominator of v).
3. M is a transitive relation.

Property 1 reduces the computation of J to that of M. Conversely, M can be uniquely reconstructed from the J sets, since $M(w) = J(\{\texttt{START}, w\})$. Hence, *the*

[3]Ramalingam [Ram00] has proposed a variant of the SSA form which may place ϕ-functions at nodes other than those of the SSA form as defined by Cytron *et al.* [CFR+91]; thus, it is outside the scope of this paper.

merge relation summarizes the information necessary and sufficient to obtain any J set for a given CGF.

Property 2 provides a handle for efficient computation of M by linking the merge relation to the extensively studied *dominance* relation. A first step in this direction is taken in Section 2.2, which presents two simple but inefficient algorithms for computing the M relation, one based on graph reachability and the other on dataflow analysis.

Property 3, established in Section 2.3, opens the door to efficient preprocessing techniques based on any partial transitive reduction R of M ($R^+ = M$). In fact, $J(S) = \cup_{x \in S} M(x) = \cup_{x \in S} R^+(x)$. Hence, *for any partial reduction R of M, $J(S)$ equals the set $R^+(S)$ of nodes reachable from some $x \in S$ in graph $G_R = (V, R)$, via a non trivial path (a path with at least one edge).*

As long as relations are represented *element-wise* by explicitly storing each element (pair of CFG nodes), any SSA technique based on constructing relation R leads to preprocessing space $S_p = O(|V| + |R|)$ and to query time $T_q = O(|V| + |R|)$; these two costs are clearly minimized when $R = M_r$, the (total) *transitive reduction* of M. However, the preprocessing time T_p to obtain R from the CFG $G = (V, E)$ is not necessarily minimized by the choice $R = M_r$. Since there are CFGs for which the size of any reduction of M is quadratic in the size of the CFG itself, working with the element-wise representations might be greatly inefficient. This motivates the search for a partial reduction of M for which there are representations that (i) have small size, (ii) can be efficiently computed from the CFG, and (iii) support efficient computation of the reachability information needed to obtain J sets.

A candidate reduction of M is identified in Section 3. There, we observe that any M-path can be uniquely expressed as the concatenation of *prime* M-paths that are not themselves expressible as the concatenation of smaller M-paths. It turns out that there is a prime M-path from w to v if and only if v is in the *dominance frontier* of w, where dominance frontier DF is the relation defined in [CFR$^+$91]. As a consequence, DF is a partial reduction of M; that is, $DF^+ = M$. This is a remarkable characterization of the iterated dominance frontiers DF^+ since the definition of M makes no appeal to the notion of dominance.

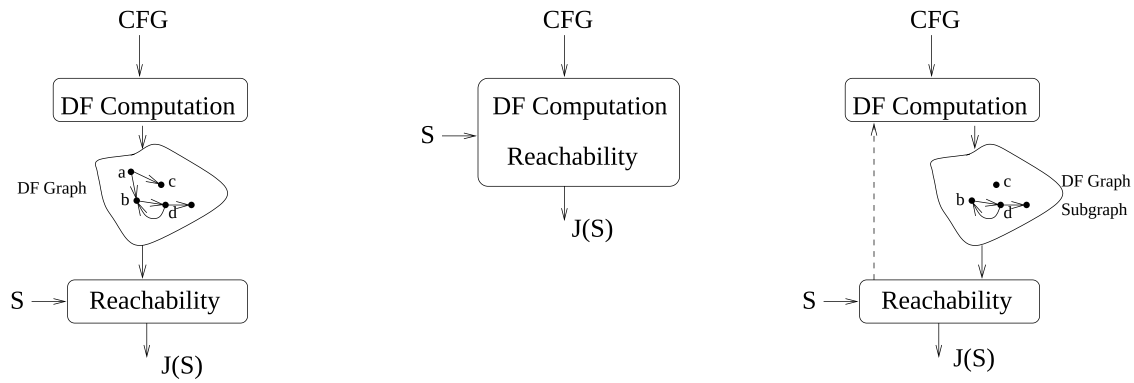
Thus, we arrive at the following characterization of the J sets:

1. $G_{DF} = f(G)$, where f is the function that maps a control flow graph G into the corresponding dominance frontier graph;
2. $J(S) = g(S, G_{DF})$, where g is the function that, given a set S of nodes and the dominance frontier graph G_{DF} of G, outputs $DF^+(S)$.

The algorithms described in this paper are produced by choosing (a) a specific way of representing and computing G_{DF}, and (b) a specific way of combining steps 1 and 2.

Algorithms for computing G_{DF} can be classified broadly into *predecessor-oriented* algorithms, which work with the set $DF^{-1}(v)$ of the predecessors in G_{DF} of each node v, and *successor-oriented* algorithms, which work with the set $DF(w)$ of the successors in G_{DF} of each node w. Section 3.2 develops the key expressions for these two approaches.

The strategies by which the DF and the reachability computations are combined are shown pictorially in Figure 2 and discussed next.



(1) Two-phase algorithms (2) Lock-step algorithms (3) Lazy algorithms

Fig. 2. Three strategies for computing ϕ-placement

Two-phase algorithms. The entire DF graph is constructed, and then the nodes reachable from input set S are determined. With the notation introduced above, this corresponds to computing $g(f(x))$, by computing $f(x)$ first and passing its output to g.

The main virtue of two-phase algorithms is simplicity. In Section 4, we describe two such algorithms: *edge-scan*, a predecessor-oriented algorithm first proposed here, and *node-scan*, a successor-oriented algorithm due to Cytron et al. [CFR+91]. Both algorithms use preprocessing time $T_p = O(|V|+|E|+|DF|)$ and preprocessing space $S_p = O(|V| + |DF|)$. To compute a set $J(S)$, they visit the portion of G_{DF} reachable from S, in time $T_q = O(|V| + |DF|)$.

Lock-step algorithms. A potential drawback of two-phase algorithms is that the size of the DF relation can be quite large (for example $|DF| = \Omega(|V|^2)$, even for some very sparse ($|E| = O(|V|)$), structured CFGs) [CFR+91]. A lock-step algorithm interleaves the computation of the reachable set $DF^+(S)$ with that of the DF relation. Once a node is reached, further paths leading to it do not add useful information, which ultimately makes it possible to construct only a subgraph $G'_{DF} = f'(G, S)$ of the DF graph that is sufficient to determine $J(S) = g'(S, G'_{DF})$.

The idea of simplifying the computation of $f(g(x))$ by interleaving the computations of f and g is quite general. In the context of loop optimizations, this is similar to *loop jamming* [Wolfe95] which may permit optimizations such as scalarization. Frontal algorithms for out-of-core sparse matrix factorizations [GL81] exploit similar ideas.

In Section 5, we discuss two lock-step algorithms, a predecessor-oriented *pulling* algorithm and a successor-oriented *pushing* algorithm; for both, $T_p, S_p, T_q = O(|V|+|E|)$. A number of structural properties of the merge and dominance frontier relations, established in this section, are exploited by the pulling and pushing algorithms. In particular, we exploit a result which permits us to topologically sort a suitable acyclic condensate of the dominance frontier graph without actually constructing this graph.

Lazy algorithms. A potential source of inefficiency of lock-step algorithms is that they perform computations at all nodes of the graph, even though only a

Approach	Order	T_p	S_p	T_q
M relation (Section 2):				
Reachability	pred.	$\|V\|\|E\|$	$\|V\|^2$	$\sum_{v \in S} \|M(v)\|$
Backward dataflow	succ.	$\|V\|\|E\|^2$	$\|V\|\|E\|$	$\sum_{v \in S} \|M(v)\|$
DF relation (Section 3):				
Two phase (Section 4):				
Edge scan	pred.	$\|V\| + \|DF\|$	$\|V\| + \|DF\|$	$\sum_{v \in S \cup J(S)} \|DF(v)\|$
Node scan [CFR+91]	succ.	$\|V\| + \|DF\|$	$\|V\| + \|DF\|$	$\sum_{v \in S \cup J(S)} \|DF(v)\|$
Lock-step (Section 5):				
Pulling	pred.	$\|V\| + \|E\|$	$\|V\| + \|E\|$	$\|V\| + \|E\|$
Pushing	succ.	$\|V\| + \|E\|$	$\|V\| + \|E\|$	$\|V\| + \|E\|$
Lazy (Section 6):				
Fully lazy [SG95]	succ.	$\|V\| + \|E\|$	$\|V\| + \|E\|$	$\|V\| + \|E\|$
Lazy pulling [PB95]	succ.	$h_\beta(\|V\|, \|E_{up}\|)$	$h_\beta(\|V\|, \|E_{up}\|)$	$h_\beta(\|V\|, \|E_{up}\|)$
			$h_\beta(\|V\|, \|E_{up}\|) = \|E_{up}\| + (1 + 1/\beta)\|V\|$	
M_r relation (Section 8):				
Two phase for structured programs (Section 8):				
Forest	succ.	$\|V\| + \|E\|$	$\|V\| + \|E\|$	$\|S\| + \|J(S)\|$

Fig. 3. Overview of ϕ-placement algorithms. $O()$ estimates are reported for preprocessing time T_p, preprocessing space S_p, and query time T_q.

small subset of these nodes may be relevant for computing $M(S)$ for a given S. A second source of inefficiency in lock-step algorithms arises when several sets $J(S_1)$, $J(S_2) \ldots$ have to be computed, since the DF information is derived from scratch for each query.

Both issues are addressed in Section 6 with the introduction of the *augmented dominator tree*, a data structure similar to the augmented postdominator tree [PB97]. The first issue is addressed by constructing the DF graph lazily as needed by the reachability computation. The idea of lazy algorithms is quite general and involves computing $f(g(x))$ by computing only that portion of $g(x)$ that is required to produce the output of f [Haskell]. In our context, this means that we compute only that portion of the DF relation that is required to perform the reachability computation. The second issue is addressed by precomputing and caching DF sets for certain carefully chosen nodes in the dominator tree. *Two-phase algorithms can be viewed as one extreme of this approach in which the entire DF computation is performed eagerly.*

In Section 7, lazy algorithms are evaluated experimentally, both on a micro-benchmark and on the SPEC benchmarks.

Although these ϕ-placement algorithms are efficient in practice, a query time of $O(|V| + |E|)$ is not asymptotically optimal when ϕ sets have to be found for several variables in the same program. In Section 8, for the special case of structured programs, we achieve $T_q = O(|S| + |J(S)|)$, which is asymptotically optimal because it takes at least this much time to read the input (set S) and write the output (set $J(S)$). We follow the two-phase approach; however, the total transitive reduction M_r of M is computed instead of DF. This is because M_r for a structured program is a forest which can be constructed, stored, and searched very efficiently. Achieving query time $T_q = O(|S| + |J(S)|)$ for general programs remains an open problem.

In summary, the main contributions of this paper are the following.

1. We define the *merge* relation on nodes of a CFG and use it to derive systematically all known properties of the SSA form.
2. We place existing ϕ-placement algorithms into a simple framework (Figure 3).
3. We present two new $O(|V| + |E|)$ algorithms for ϕ-placement, *pushing* and *pulling*, which emerged from considerations of this framework.
4. For the special case of structured programs, we present the first approach to answer ϕ-placement queries in optimal time $O(|S| + |J(S)|)$.

2. THE MERGE RELATION AND ITS USE IN ϕ-PLACEMENT

In this section, we reduce ϕ-placement to the computation of a binary relation M on nodes called the *merge* relation. We begin by establishing a link between the merge and the dominance relations. Based on this link, we derive two algorithms to compute M and show how these provide simple but inefficient solutions to the ϕ-placement problem. We conclude the section by showing that the merge relation is transitive but that it might prove difficult to compute its transitive reduction efficiently. This motivates the search for partial reductions and leads to the introduction of the DF relation in Section 3.

2.1 The Merge Relation

Definition 2.1. **Merge** is a binary relation $M \subseteq V \times V$ defined as follows:

$$M = \{(w, v) | v \in J(\{\texttt{START}, w\})\}.$$

For any node w, the *merge set* of node w, denoted by $M(w)$, is the set $\{v|(w, v) \in M\}$. Similarly, we let $M^{-1}(v) = \{w|(w, v) \in M\}$.

Intuitively, $M(w)$ is the set of the nodes where ϕ-functions must be placed if the only assignments to the variable are at `START` and w; conversely, a ϕ-function is needed at v if the variable is assigned in any node of $M^{-1}(v)$. Trivially, $M(\texttt{START}) = \{\}$. Next, we shows that if S contains `START`, then $J(S)$ is the union of the merge sets of the elements of S.

THEOREM 2.2. *Let* $G = (V, E)$ *and* $\{\texttt{START}\} \subseteq S \subseteq V$. *Then,* $J(S) = \cup_{w \in S} M(w)$.

PROOF. It is easy to see from the definitions of J and M that $\cup_{w \in S} M(w) \subseteq J(S)$. To show that $J(S) \subseteq \cup_{w \in S} M(w)$, consider a node $v \in J(S)$. By Definition 1.1, there are paths $a \overset{+}{\to} v$ and $b \overset{+}{\to} v$, with $a, b \in S$, intersecting only at v. By Definition A.1, there is also a path $\texttt{START} \overset{+}{\to} v$. There are two cases:

1. Path $\texttt{START} \overset{+}{\to} v$ intersects path $a \overset{+}{\to} v$ only at v. Then, $v \in M(a)$, hence $v \in \cup_{w \in S} M(w)$.
2. Path $\texttt{START} \overset{+}{\to} v$ intersects path $a \overset{+}{\to} v$ at some node different from v. Then, let z be the first node on path $\texttt{START} \overset{+}{\to} v$ occurring on either $a \overset{+}{\to} v$ or $b \overset{+}{\to} v$. Without loss of generality, let z be on $a \overset{+}{\to} v$. Then, there is clearly a path $\texttt{START} \overset{+}{\to} z \overset{+}{\to} v$ intersecting with $b \overset{+}{\to} v$ only at v, so that $v \in M(b)$, hence $v \in \cup_{w \in S} M(w)$.

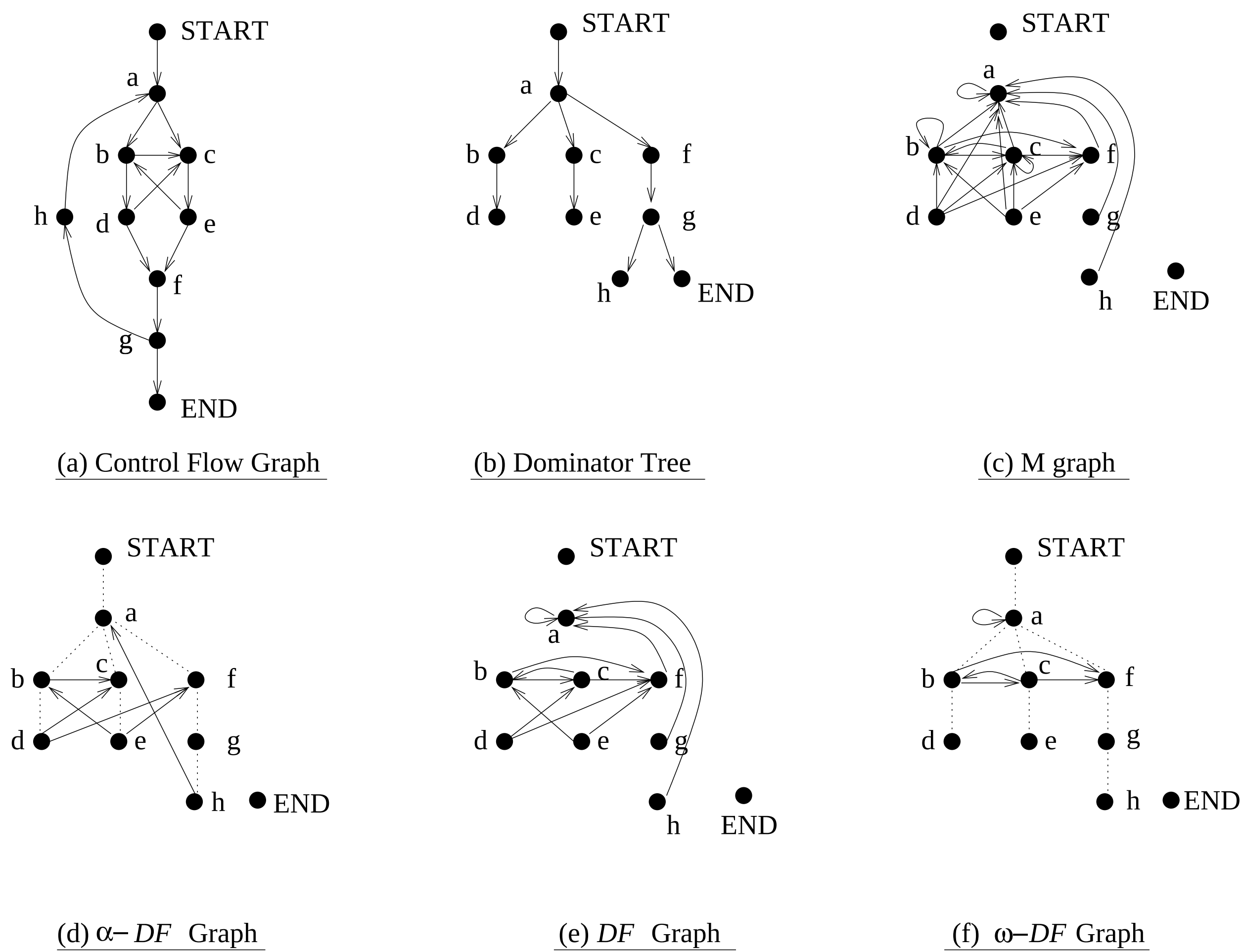


Fig. 4. A control flow graph and its associated graphs

□

The control flow graph in Figure 4(a) is the running example used in this paper. Relation M defines a graph $G_M = (V, M)$. The M graph for the running example is shown in Figure 4(c). Theorem 2.2 can be interpreted graphically as follows: *for any subset S of the nodes in a CFG, $J(S)$ is the set of neighbors of these nodes in the corresponding M graph.* For example, $J(\{b, c\}) = \{b, c, f, a\}$.

There are deep connections between merge sets and the standard notion of *dominance* (reviewed in the Appendix), rooted in the following result:

THEOREM 2.3. *For any $w \in V$, $v \in M(w)$ iff there is a path $w \xrightarrow{+} v$ not containing $idom(v)$.*

PROOF. ($\Rightarrow$) If $v \in M(w)$, Definition 2.1 asserts that there are paths $P1 = \texttt{START} \xrightarrow{+} v$ and $P2 = w \xrightarrow{+} v$ which intersect only at v. Since, by Definition A.3,

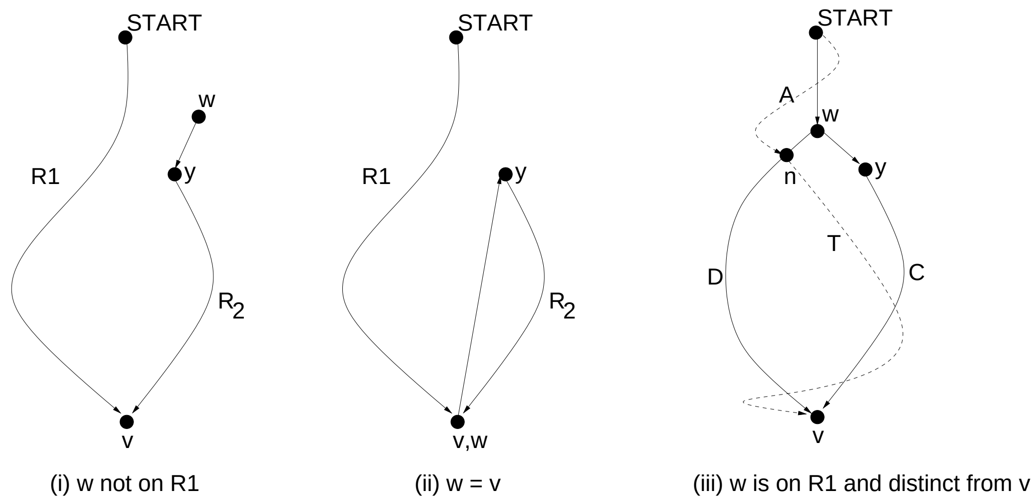


Fig. 5. Case analysis for Theorem 2.3

every dominator of v must occur on $P1$, no strict dominator of v can occur on $P2$. Hence, $P2$ does not contain $idom(v)$.

($\Leftarrow$) Assume now the existence of a path $P = w \xrightarrow{+} v$ that does not contain $idom(v)$. By induction on the length (number of arcs) of path P, we argue that there exists paths $P1 = \texttt{START} \xrightarrow{+} v$ and $P2 = w \xrightarrow{+} v$ which intersect only at v, *i.e.*, $w \in M(v)$.

Base case: Let the length of P be 1, *i.e.*, P consists only of edge $w \rightarrow v$. If $v = w$, let $P2 = P$ and let $P1$ be any simple path from START to v, and the result is obtained. Otherwise, v and w are distinct. There must be a path $T = \texttt{START} \xrightarrow{+} v$ that does not contain w, since otherwise, w would dominate v, contradicting Lemma 2.5(ii). The required result follows by setting $P2 = P$ and $P1 = T$.

Inductive step: Let the length of P be at least two so that $P = w \rightarrow y \xrightarrow{+} v$. By the inductive assumption, there are paths $R1 = \texttt{START} \xrightarrow{+} v$ and $R2 = y \xrightarrow{+} v$ intersecting only at v. Let C be the path obtained by concatenating the edge $w \rightarrow y$ to the path $R2$ and consider the following two cases:

- $w \notin (R1 - \{v\})$. Then, let $P1 = R1$ and $P2 = C$. Figures 5(i) and (ii) illustrate the sub-cases $w \neq v$ and $w = v$, respectively.
- $w \in (R1 - \{v\})$. Let D be the suffix $w \xrightarrow{+} v$ of $R1$ and observe that C and D intersect only at their endpoints w and v (see Figure 5(iii)). Let also $T = \texttt{START} \xrightarrow{+} v$ be a path that does not contain w (the existence of T was established earlier). Let n be the first node on T that is contained in either C or D (such a node must exist since all three paths terminate at v). Consider the following cases:

  1. $n = v$. Then, we let $P1 = T$, and $P2 = C$.
  2. $n \in (D - C)$. Referring to Figure 5, let $P1$ be the concatenation of the

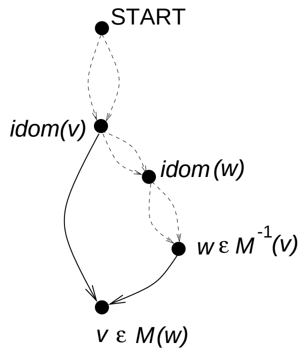


Fig. 6. A pictorial representation of Lemma 2.5

prefix $\texttt{START} \xrightarrow{+} n$ of T with the suffix $n \xrightarrow{+} v$ of D, which is disjoint from $P2 = C$ except for v.

3. $n \in (C - D)$. The proof is analogous to the previous case and is omitted.

□

The dominator tree for the running example of Figure 4(a) is shown in Figure 4(b). Consider the path $P = e \to b \to d \to f$ in Figure 4(a). This path does not contain $idom(f) = a$. As required by the theorem, there are paths $P_1 = \texttt{START} \to a \to b \to d \to f$ and $P_2 = e \to f$ with only f in common, *i.e.*, $f \in M(e)$.

The preceding result motivates the following definition of M-paths.

Definition 2.4. Given a CFG $G = (V, E)$, an M-path is a path $w \xrightarrow{+} v$ that does not contain $idom(v)$.

Note that M-paths are paths in the CFG, not in the graph of the M relation. They enjoy the following important property, illustrated in Figure 6.

LEMMA 2.5. *If $P = w \xrightarrow{+} v$ is an M-path, then (i) $idom(v)$ strictly dominates all nodes on P, hence (ii) no strict dominator of v occurs on P.*

PROOF. (i) (By contradiction.) Let n be a node on P that is not strictly dominated by $idom(v)$. Then, there is a path $Q = \texttt{START} \to n$ that does not contain $idom(v)$; concatenating Q with the suffix $n \to v$ of P, we get a path from START to v that does not contain $idom(v)$, a contradiction.

(ii) Since dominance is tree-structured, any strict dominator d of v dominates $idom(v)$, hence d is not strictly dominated by $idom(v)$ and, by (i), can not occur on P. □

We note that in Figure 6, $idom(v)$ strictly dominates w (Lemma 2.5(i)); so from the definition of $idom$, it follows that $idom(v)$ also dominates $idom(w)$.

2.2 Computing the Merge Relation

Approaches to computing M can be naturally classified as being *successor oriented* (for each w, $M(w)$ is determined) or *predecessor oriented* (for each v, $M^{-1}(v)$

Procedure Merge(CFG);
{
Assume CFG $= (V, E)$;
$M = \{\}$;
for $v \in V$ **do**
  Let $G' = (G - idom(v))^R$;
  Traverse G' from v, appending (w, v) to M for each visited w.
od
return M;
}
Procedure ϕ-placement(M, S);
{
$J = \{\}$;
for each $v \in S$
  for each $(v, w) \in M$ append w to J;
return J;
}

Fig. 7. Reachability algorithm

is determined). Next, based on Theorem 2.3, we describe a predecessor-oriented algorithm which uses graph reachability and a successor-oriented algorithm which solves a backward dataflow problem.

2.2.1 *Reachability Algorithm.* The *reachability algorithm* shown in Figure 7 computes the set $M^{-1}(y)$ for any node y in the CFG by finding the the set of nodes reachable from y in the graph obtained by deleting $idom(y)$ from the CFG and reversing all edges in the remaining graph (we call this graph $(G - idom(y))^R$. The correctness of this algorithm follows immediately from Theorem 2.3.

PROPOSITION 2.6. *The reachability algorithm for SSA has preprocessing time $T_p = O(|V||E|)$, preprocessing space $S_p = O(|V| + |M|) \leq O(|V|^2)$, and query time $T_q = O(\sum_{v \in S} |M(v)|)$.*

PROOF. The bound on preprocessing time comes from the fact that there are $|V|$ visits each to a subgraph of $G = (V, E)$, hence taking time $O(|E|)$. The bound on preprocessing space comes from the need to store $|V|$ nodes and $|M|$ arcs to represent the M relation. The bound on query time comes from the fact that each $M(v)$ for $v \in S$ is obtained in time proportional to its own size. The bound on T_p also subsumes the time to construct the dominator tree, which is $O(|E|)$, (cf. Appendix). □

2.2.2 *Dataflow Algorithm.* We now show that the structure of the M-paths leads to an expression for set $M(w)$ in terms of the sets $M(u)$ for successors u of w in the CFG. This yields a system of backward dataflow equations which can be solved by any one of the numerous methods in the literature [ASU86].

Here and in several subsequent discussions, it is convenient to partition the edges of the control flow graph $G = (V, E)$ as $E = E_{tree} + E_{up}$, where $(u \to v) \in E_{tree}$ (a tree edge of the dominator tree of the graph) if $u = idom(v)$, and $(u \to v) \in E_{up}$ (an *up-edge*) otherwise. Figure 4(a,b) shows a CFG and its dominator tree. In Figure 4(a), $a \to b$ and $g \to h$ are tree edges, while $h \to a$ and $e \to b$ are up-edges.

For future reference, we introduce the following definition.

Definition 2.7. Given a CFG $G = (V, E)$, $(u \to v) \in E$ is an up-edge if $u \neq idom(v)$. The subgraph (V, E_{up}) of G containing only the up-edges is called the α-DFgraph.

Figure 4(d) shows the α-DF graph for the CFG of Figure 4(a). Since an up-edge $(u \to v)$ is a path from u to v that does not contain $idom(v)$, its existence implies $v \in M(u)$ (from Theorem 2.3); then, from the transitivity of M, $E_{up}^{+} \subseteq M$. In general, the latter relation does not hold with equality (for example, in Figure 4, $a \in M(g)$ but a is not reachable from g in the α-DF graph). Fortunately, the set $M(w)$ can be expressed as a function of α-$DF(w)$ and the sets $M(u)$ for all CFG successors u of w as follows. We let $children(w)$ represent the set of children of w in the dominator tree.

THEOREM 2.8. *The merge sets of the nodes of a CFG satisfy the following set of relations, for $w \in V$:*

$$M(w) = \alpha{-}DF(w) \cup (\cup_{u \in succ(w)} M(u) - children(w)). \tag{3}$$

PROOF. (a) We first prove that $M(w) \subseteq \alpha$-$DF(w) \cup (\cup_{u \in succ(w)} M(u) - children(w))$.
If $v \in M(w)$, Theorem 2.3 implies that there is a path $P = w \overset{+}{\to} v$ that does not contain $idom(v)$; therefore, $w \neq idom(v)$. If the length of P is 1, then $v \in succ(w)$ and $w \neq idom(v)$, so $v \in \alpha$-$DF(w)$. Otherwise P can be written as $w \to u \overset{+}{\to} v$. Since $idom(v)$ does not occur on the sub-path $u \overset{+}{\to} v$, $v \in M(u)$; furthermore, since $w \neq idom(v)$, $v \in M(u) - children(w)$.
(b) We now show that $M(w) \supseteq \alpha$-$DF(w) \cup (\cup_{u \in succ(w)} M(u) - children(w))$.
If $v \in \alpha$-$DF(w)$, the CFG edge $w \to v$ is an M-path from w to v; so $v \in M(w)$ from Theorem 2.3. If $v \in (\cup_{u \in succ(w)} M(u) - children(w))$, (i) there is a CFG edge $w \to u$, (ii) $v \in M(u)$ and (iii) $w \neq idom(v)$. From Theorem 2.3, there is an M-path $P_1 = u \overset{+}{\to} v$. The path obtained by prepending edge $w \to u$ to path P_1 is an M-path; therefore, $v \in M(w)$. □

We observe that since $\alpha{-}DF(w)$ and $children(w)$ are disjoint, no parentheses are needed in Equation 3, if set union is given precedence over set difference. For the CFG of Figure 4(a), the $M(w)$ sets are related as shown in Figure 8. For an acyclic CFG, the system of equations (3) can be solved for $M(w)$ in a single pass, by processing the nodes w's in reversal topological order of the CFG. For a CFG with cycles, one has to resort to the more general, well-established framework of equations over lattices [ASU86], as outlined next.

THEOREM 2.9. *The M relation is the* least solution *of the dataflow equations (3), where the unknowns $\{M(w) : w \in V\}$ range over the lattice $\mathcal{L}$ of all subsets of V, ordered by inclusion.*

PROOF. Let L be the least solution of the dataflow equations. Clearly, $L \subseteq M$, since M is also a solution. To conclude that $M = L$ it remains to prove that $M \subseteq L$. We establish this by induction on the length of shortest (minimal length) M-paths.

$$\begin{aligned}
M(\texttt{START}) &= M(a) - \{a\} \\
M(a) &= M(b) \cup M(c) - \{b, c, f\} \\
M(b) &= \{c\} \cup M(c) \cup M(d) - \{d\} \\
M(c) &= M(e) - \{e\} \\
M(d) &= \{c, f\} \cup M(c) \cup M(f) \\
M(e) &= \{f\} \cup M(b) \cup M(f) \\
M(f) &= M(g) - \{g, h, \texttt{END}\} \\
M(g) &= M(h) \cup M(\texttt{END}) - \{h, \texttt{END}\} \\
M(h) &= \{a\} \cup M(a) \\
M(\texttt{END}) &= \{\}
\end{aligned}$$

(a) Dataflow equations

$$\begin{aligned}
M(\texttt{START}) &= \{\} \\
M(a) &= \{a\} \\
M(b) &= \{b, c, f, a\} \\
M(c) &= \{b, c, f, a\} \\
M(d) &= \{b, c, f, a\} \\
M(e) &= \{b, c, f, a\} \\
M(f) &= \{a\} \\
M(g) &= \{a\} \\
M(h) &= \{a\} \\
M(\texttt{END}) &= \{\}
\end{aligned}$$

(b) Solution of dataflow equations

Fig. 8. Equations set up and solved by the dataflow algorithm, for the CFG in Figure 4(a)

Consider any pair $(w, v) \in M$ such that there is an M-path of length 1 from w to v. This means that $v \in \alpha\text{-}DF(w)$, so from Equation 3, $(w, v) \in L$.

Inductively, assume that if $(u, v) \in M$ and the minimal length M-path from u to v has length n, then $(u, v) \in L$. Consider a pair $(w, v) \in M$ for which there is a minimal length M-path $w \rightarrow u \xrightarrow{+} v$ of length $(n + 1)$. The sub-path $u \xrightarrow{+} v$ is itself an M-path and is of length n; therefore, by inductive assumption, $(u, v) \in L$. Since $w \neq idom(v)$, it follows from Equation 3 that $(w, v) \in L$. $\square$

The least solution of dataflow equations (3) can be determined by any of the techniques in the literature [ASU86]. A straightforward iterative algorithm operates in space $O(|V|^2)$ and time $O(|V|^2|E|^2)$, charging time $O(|V|)$ for bit-vector operations. The above considerations, together with arguments already developed in the proof of Proposition 2.6, lead to the following result:

PROPOSITION 2.10. *There is a dataflow algorithm for SSA with preprocessing time $T_p = O(|V|^2|E|^2)$, preprocessing space $S_p = O(|V| + |M|) \leq O(|V|^2)$, and query time $T_q = O(\sum_{v \in S} |M(v)|)$.*

In Section 5, as a result of a deeper analysis of the structure of the M relation, we shall show that a topological ordering of the (acyclic condensate) of the M graph can be constructed in time $O(|E|)$, directly from the CFG. Using this ordering, a single-pass over the dataflow equations becomes sufficient for their solution, yielding $T_p = O(|V||E|)$ for the computation of M.

2.3 M is Transitive

In general, the merge relation of a CFG can be quite large, so it is natural to explore ways to avoid computing and storing the entire relation. As a first step in this direction, we show that the fact that M-paths are closed under concatenation leads immediately to a proof that M is transitive.

THEOREM 2.11. *If $P_1 = x \xrightarrow{+} y$ and $P_2 = y \xrightarrow{+} z$ are M-paths, then so is their concatenation $P = P_1 P_2 = x \xrightarrow{+} z$. Hence, M is transitive.*

PROOF. By Definition 2.4, P_1 does not contain $idom(y)$ and P_2 does not contain $idom(z)$. We will show that $idom(z)$ cannot occur in P_1, so concatenating P_1 and P_2 gives a path P from x to z that does not contain $idom(z)$, as claimed. We note that $idom(z)$ is distinct from y since it does not occur on path P_2. Furthermore, from Lemma 2.5(i), $idom(z)$ must strictly dominate y. If $idom(z) = idom(y)$, then this node does not occur on P, and the required result is proved. Otherwise, $idom(z)$ strictly dominates $idom(y)$, so we conclude from Lemma 2.5(ii) that $idom(z)$ does not occur on P_1.

From Theorem 2.3, it follows that P is an M-path. □

As an illustration of the above theorem, with reference to Figure 4(a), consider the M-paths $P_1 = b \to d \to f$ (which does not contain $idom(f) = a$) and $P_2 = f \to g \to h \to a$ (which does not contain $idom(a) = \texttt{START}$). Their concatenation $P = P_1P_2 = b \to d \to f \to g \to h \to a$ does not contain $idom(a) = \texttt{START}$; hence it is an M-path.

Combining Theorems 2.2 and 2.11, we obtain another graph-theoretic interpretation of *a join set $J(S)$ as the set of nodes reachable in the M graph by non-empty paths originating at some node in S.* It follows trivially that $J(S \cup J(S)) = J(S)$, as first shown by Weiss [Weiss92].

2.4 Transitive Reductions of M

We observe that if R is a relation such that $M = R^+$, the set of nodes reachable from any node by non-empty paths is the same in the two graphs $G_R = (V, R)$ and $G_M = (V, M)$. Since $|R|$ can be considerably smaller than $|M|$, using G_R instead of G_M as the data structure to support queries could lead to considerable savings in space. The query time can also decrease substantially. Essentially, a query requires a visit to the subgraph $G_R(S) = (S \cup M(S), R_S)$ containing all the nodes and arcs reachable from S in G_R. Therefore, since the visit will spend constant time per node and per edge, query time is $T_q = O(|S| + |M(S)| + |R_S|)$.

Determining a relation R such that $R^+ = M$ for a given transitive M is a well-known problem. Usually, an R of minimum size, called the *transitive reduction* of M is the goal. Unless M is acyclic (*i.e.*, the graph G_M is a dag), R is not necessarily unique. However, if the strongly connected components of M are collapsed into single vertices, the resulting acyclic condensate (call it M_c) has a unique transitive reduction M_r which can be computed in time $O(|V||M_c|)$ [CLR92] or $O(|V|^\gamma)$ by using an $O(n^\gamma)$ matrix multiplication algorithm[4]. In summary:

PROPOSITION 2.12. *The reachability algorithm for ϕ-placement (with transitive reduction preprocessing) has preprocessing time $T_p = O(|V|(|E| + \min(|M|, |V|^{\gamma-1}))$, preprocessing space $S_p = O(|V| + |M_r|)$, and query time $T_q = O(|V| + |M_r^+(S)|)$.*

Clearly, preprocessing time is too high for this algorithm to be of much practical interest. It is natural to ask whether the merge relation M has any special structure that could facilitate the transitive reduction computation. Unfortunately, for general programs, the answer is negative. Given an arbitrary relation $R \subseteq (V - \texttt{START}) \times (V - \texttt{START})$, it can be easily shown that the CFG

[4]For instance, $\gamma = 3$ for the standard algorithm and $\gamma = \log_2 7 \approx 2.81$ for Strassen's algorithm [CLR92].

$G = (V, R \cup (\{\texttt{START}\} \times (V - \texttt{START})))$ has exactly R^+ as its own merge relation M. In particular, if R is transitive to start with, then $M = R$.

Rather than pursuing the *total* transitive reduction of M, we investigate *partial* reductions next.

3. THE DOMINANCE FRONTIER RELATION

We have seen that the M relation is uniquely determined by the set of M-paths (Theorem 2.3), which is closed under concatenation (Theorem 2.11). We can therefore ask the question: "what is the smallest subset of M-paths by concatenating which one obtains all M-paths?" We will characterize this subset in Section 3.1 and discover that it is intimately related to the well-known dominance frontier relation [CFR+91]. Subsequent subsections explore a number of properties of dominance frontier, as a basis for the development of SSA algorithms.

3.1 Prime Factorization of M-paths Leads to Dominance Frontier

We begin by defining the key notion needed for our analysis of M.

Definition 3.1. Given a graph $G = (V, E)$ and a set $\mathcal{M}$ of paths closed under concatenation, a path $P \in \mathcal{M}$ is *prime* whenever there is no pair of non empty paths P_1 and P_2 such that $P = P_1 P_2$.

With reference to the example immediately following Theorem 2.11 and letting $\mathcal{M}$ denote the set of M-paths, we can see that P is not prime while P_1 and P_2 are prime. Our interest in prime paths stems from the following fact, whose straightforward proof is omitted.

PROPOSITION 3.2. *With the notation of Definition 3.1, path P can be expressed as the concatenation of one or more prime paths if and only if $P \in \mathcal{M}$.*

Next, we develop a characterization of the prime paths for the set of M-paths.

PROPOSITION 3.3. *Let $\mathcal{M}$ be the set of M-paths in a CFG and let $P = w \to x_1 \to \ldots \to x_{n-1} \to v$ be a CFG path. Then, P is prime if and only if*

1. *w strictly dominates nodes $x_1, x_2, \ldots, x_{n-1}$, and*
2. *w does not strictly dominate v.*

PROOF. Assume P to be a prime path. Since P in an M-path, by Lemma 2.5, w does not strictly dominate v. Then, let P_1 be the shortest, non empty prefix of P terminating at a vertex x_i that is not strictly dominated by w. Clearly, P_1 satisfies properties 1 and 2. We claim that $P_1 = P$. Otherwise, the primality of P would be contradicted by the factorization $P = P_1 P_2$ where (i) P_1 is an M-path, since by construction $idom(x_i)$ is not dominated by w, hence does not occur on P_1, and (ii) P_2 is an M-path since $idom(v)$ does not occur on P (an M-path ending at v) and a fortiori on P_2.

Assume now that P is a path satisfying properties 1 and 2. We show that P is prime, *i.e.*, it is in $\mathcal{M}$ and it is not factorable.
(a)P is an M-path. In fact, if $idom(v)$ were to occur on P, then by property 1, w would dominate $idom(v)$ and, by transitivity of dominance, it would strictly dominate v, contradicting property 2. Thus P does not contain $idom(v)$ hence, by

Theorem 2.3 it is an M-path.
(b)P can not be factored as $P = P_1P_2$ where P_1 and P_2 are both non-empty M-paths. In fact, for any proper prefix $P_1 = w \xrightarrow{+} x_i$, x_i is strictly dominated by w. Then, by Lemma 2.5, $idom(x_i)$ occurs on P_1, which therefore is not an M-path. □

The reader familiar with the notion of dominance frontier will quickly recognize that properties 1 and 2 of Proposition 3.3 imply that v belongs to the dominance frontier of w. Before exploring this interesting connection, let us recall the relevant definitions:

Definition 3.4. A CFG edge $(u \to v)$ is in the *edge dominance frontier* $EDF(w)$ of node w if

1. w dominates u, and
2. w does not strictly dominate v.

If $(u \to v) \in EDF(w)$, then v is said to be in the *dominance frontier* $DF(w)$ of node w and the dominance frontier relation is said to hold between w and v, written $(w, v) \in DF$.

It is often useful to consider the DF graph $G_{DF} = (V, DF)$ associated with binary relation DF, which is illustrated in Figure 4(e) for the running example. We are now ready to link the merge relation to dominance frontier.

PROPOSITION 3.5. *There exists a prime M-path from w to v if and only if $(w, v) \in DF$.*

PROOF. Assume first that P is a prime M-path from w to v. Then, P satisfies properties 1 and 2 of Proposition 3.3, which straightforwardly imply, according to Definition 3.4, that $(x_{n-1} \to v) \in EDF(w)$, hence $(w, v) \in DF$.

Assume now that $(v, w) \in DF$. Then, by Definition 3.4, there is in the CFG an edge $u \to v$ such that (i) w dominates u and (ii) w does not strictly dominate v. By (i) and Lemma A.4, there is a path $Q = w \xrightarrow{*} u$ on which each node is dominated by w. If we let $R = w \xrightarrow{*} u$ be the smallest suffix of Q whose first node equals w, then each node on R except for the first one is strictly dominated by w. This fact together with (ii) implies that the path $P = R(u \to v)$ satisfies properties 1 and 2 of Proposition 3.3, hence it is a prime M-path from w to v. □

The developments of this subsection lead to the sought partial reduction of M.

THEOREM 3.6. $M = DF^+$.

PROOF. The stated equality follows from the equivalence of the sequence of statements listed below, where the reason for the equivalence of a statement to its predecessor in the list is in parenthesis.

—$(w, v) \in M$;

—there exists an M-path P from w to v, (by Theorem 2.3);

—for some $k \geq 1$, $P = P_1P_2 \ldots P_k$ where $P_i = w_i \xrightarrow{+} v_i$ are prime M-paths such that $w_1 = w$, $v_k = v$, and for $i = 2, \ldots, k$, $w_i = v_{i-1}$, (by Proposition 3.2 and Theorem 2.11);

—for some $k \geq 1$, for $i = 1, \ldots, k$, $(w_i, v_i) \in DF$, with $w_1 = w$, $v_k = v$, and for $i = 2, \ldots, k$, $w_i = v_{i-1}$, (by Proposition 3.5);
—$(w, v) \in DF^+$, (by definition of transitive closure).

□

In general, DF is neither transitively closed nor transitively reduced, as can be seen in Figure 4(e). The presence of $c \rightarrow f$ and $f \rightarrow a$ and the absence of $c \rightarrow a$ in the DF graph show that it is not transitively closed. The presence of edges $d \rightarrow c$, $c \rightarrow f$, and $d \rightarrow f$ shows that it is not transitively reduced.

Combining Theorems 2.2 and 3.6, we obtain a simple graph-theoretic interpretation of *a join set $J(S) = g(S, G_{DF})$ as the set of nodes reachable in the DF graph by non-empty paths originating at some node in S.*

3.2 Two Identities for the DF Relation

Most of the algorithms described in the rest of this paper are based on the computation of all or part of the DF graph $G_{DF} = f(G)$ corresponding to the given CFG G. We now discuss two identities for the DF relation, the first one enabling efficient computation of $DF^{-1}(v)$ sets (a predecessor-oriented approach), and the second one enabling efficient computation of $DF(w)$ sets (a successor-oriented approach).

Definition 3.7. Let $T =< V, F >$ be a tree. For $x, y \in V$, let $[x, y]$ denote the set of vertices on the simple path connecting x and y in T, and let $[x, y)$ denote $[x, y] - \{y\}$. In particular, $[x, x)$ is empty.

For example, in the dominator tree of Figure 4(b), $[d, a] = \{d, b, a\}$, $[d, a) = \{d, b\}$, and $[d, g] = \{d, b, a, f, g\}$.

THEOREM 3.8. $EDF = \bigcup_{(u \rightarrow v) \in E} [u, idom(v)) \times \{u \rightarrow v\}$, *where* $[u, idom(v)) \times \{u \rightarrow v\} = \{(w, u \rightarrow v) | \ w \in [u, idom(v))\}$.

PROOF. $\supseteq$: Suppose $(w, a \rightarrow b) \in \bigcup_{(u \rightarrow v) \in E} [u, idom(v)) \times u \rightarrow v$. Therefore, $[a, idom(b))$ is non-empty which means that $(a \rightarrow b)$ is an up-edge. Applying Lemma 2.5 to this edge, we see that $idom(b)$ strictly dominates a. Therefore, w dominates a but does not strictly dominate b, which implies that $(w, v) \in DF$ from Definition 3.4.

$\subseteq$: If $(w, v) \in DF$, there is an edge $(u \rightarrow v)$ such that w dominates u but does not strictly dominate v. Therefore $w \in [u, \texttt{START}] - [idom(v), \texttt{START}]$, which implies $u \neq idom(v)$. From Lemma 2.5, this means that $idom(v)$ dominates u. Therefore, the expression $[u, \texttt{START}] - [idom(v), \texttt{START}]$ can be written as $[u, idom(v))$, and the required result follows. □

Based on Theorem 3.8, $DF^{-1}(v)$ can be computed as the union of the sets $[u, idom(v))$ for all incoming edges $(u \rightarrow v)$. Theorem 3.8 can be viewed as the DF analog of the reachability algorithm of Figure 7 for the M relation: to find $DF^{-1}(v)$, we overlay on the dominator tree all edges $(u \rightarrow v)$ whose destination is v and find all nodes reachable from v without going through $idom(v)$ in the reverse graph.

The next result [CFR+91] provides a recursive characterization of the $DF(w)$ in terms of DF sets of the children of w in the dominator tree. There is a striking analogy with the expression for $M(w)$ in Theorem 2.8. However, the dependence of the DF expression on the dominator-tree children (rather than on the CFG successors needed for M) is a great simplification, since it enables solution in a single pass, made according to any bottom-up ordering of the dominator tree.

THEOREM 3.9. *Let $G = (V, E)$ be a CFG. For any node $w \in V$,*

$$DF(w) \;=\; \alpha\text{-}DF(w) \cup (\cup_{c \in children(w)} DF(c) - children(w)).$$

For example, consider nodes d and b in Figure 4(a). By definition, $\alpha\text{-}DF(d) = \{c, f\}$. Since this node has no children in the dominator tree, $DF(d) = \{c, f\}$. For node b, $\alpha\text{-}DF(b) = \{c\}$. Applying Theorem 3.9, we see that $DF(b) = \{c\} \cup (\{c, f\} - \{d\}) = \{c, f\}$, as required.

PROOF. ($\subseteq$) We show that if $v \in$DF(w), then v is contained in the set described by the r.h.s. expression. Applying Definition 3.4, we see that there must be an edge $(u \rightarrow v)$ such that w dominates u but does not strictly dominate v. There are two cases to consider.

1. If $w = u$, then $v \in \alpha{-}DF(w)$, so v is contained in the set described by the r.h.s. expression.
2. Otherwise, w has a child c such that c dominates u. Moreover, since w does not strictly dominate v, c (a descendant of d) cannot strictly dominate v either. Therefore, $v \in DF(c)$. Furthermore, v is not a child of w (otherwise w would strictly dominate v). Therefore, v is contained in the set described by the r.h.s. expression.

($\supseteq$) We show that if v is contained in the set described by the r.h.s. expression, then $v \in DF(w)$. There are two cases to consider.

1. If $v \in \alpha{-}DF(w)$, there is a CFG edge $(w \rightarrow v)$ such that w does not strictly dominate v. Applying Definition 3.4 with $u = w$, we see that $v \in DF(w)$.
2. If $v \in (\cup_{c \in children(w)} DF(c) - children(w))$, there is a child c of w and an edge $(u \rightarrow v)$ such that (i) c dominates u, (ii) c does not strictly dominate v, and (iii) v is not a child of w. From (i) and the fact that w is the parent of c, it follows that w dominates u.
   Furthermore, if w were to strictly dominate v, then either (a) v would be a child of w, or (b) v would be a proper descendant of some child of w. Possibility (a) is ruled out by fact (iii). Fact (ii) means that v cannot be a proper descendant of c. Finally, if v were a proper descendant of some child l of w other than c, then $idom(v)$ would not dominate u, which contradicts Lemma 2.5. Therefore, w cannot strictly dominate v. This means that $v \in DF(w)$, as required.

□

3.3 Strongly Connected Components of the DF and M Graphs

There is an immediate and important consequence of Theorem 3.8 which is useful in proving many results about the DF and M relations. The *level* of a node in the dominator tree can be defined in the usual way: the root has a level of 0; the level of any other node is 1 more than the level of its parent. From Theorem 3.8, it follows that if $(w, v) \in DF$, then there is an edge $(u \to v) \in E$ such that $w \in [u, idom(v))$; therefore, $level(w) \geq level(v)$. Intuitively, this means that DF (and M) edges are oriented in a special way with respect to the dominator tree: a DF or M edge overlayed on the dominator tree is always directed "upwards" or "sideways" in this tree, as can be seen in Figure 4. Furthermore, if $(w, v) \in DF$, then $idom(v)$ dominates w (this is a special case of Lemma 2.5). For future reference, we state these facts explicitly.

LEMMA 3.10. *Given a CFG = (V, E) and its dominator tree D, let $level(v)$ be the length of the shortest path in D from* `START` *to v. If $(w, v) \in DF$, then $level(w) \geq level(v)$ and $idom(v)$ dominates w. In particular, if $level(w) = level(v)$, then w and v are siblings in D.*

This result leads to an important property of strongly connected components (scc's) in the DF graph. If x and y are two nodes in the same scc, every node reachable from x is reachable from y and vice versa; furthermore, if x is reachable from a node, y is reachable from that node too, and vice versa. In terms of the M relation, this means that $M(x) = M(y)$ and $M^{-1}(x) = M^{-1}(y)$. The following lemma states that the scc's have a special structure with respect to the dominator tree.

LEMMA 3.11. *Given a CFG = (V, E) and its dominator tree D, all nodes in a strongly connected component of the DF relation (equivalently, the M relation) of this graph are siblings in D.*

PROOF. Consider any cycle $n_1 \to n_2 \to n_3 \ldots \to n_1$ in the scc. From Lemma 3.10, it follows that $level(n_1) \geq level(n_2) \geq level(n_3) ... \geq level(n_1)$; therefore, it must be true that $level(n_1) = level(n_2) = level(n_3)....$ From Lemma 3.10, it also follows that n_1, n_2, etc. must be siblings in D. $\square$

In Section 5, we will show how the strongly connected components of the DF graph of a CFG (V, E) can be identified in $O(|E|)$ time.

3.3.1 *Self-loops in the M Graph.* In general, relation M is not reflexive. However, for some nodes w, $(w, w) \in M$ and the merge graph (V, M) has a self-loop at w. As a corollary of Theorem 2.3 and of Lemma 2.5, such nodes are exactly those w's contained in some cycle whose nodes are all strictly dominated by $idom(w)$. An interesting application of self-loops will be discussed in Subsection 5.1.1.

3.3.2 *Irreducible Programs.* There is a close connection between the existence of *non-trivial* cycles in the DF (or M) graph and the standard notion of *irreducible* control flow graph [ASU86].

PROPOSITION 3.12. *A CFG $G = (V, E)$ is irreducible if and only if its M graph has a non-trivial cycle.*

PROOF. ($\Rightarrow$) Assume G is irreducible. Then, G has a cycle C on which no node dominates all other nodes on C. Therefore, there must be two nodes a and b for which neither $idom(a)$ nor $idom(b)$ is contained in C. Cycle C obviously contains two paths $P_1 = a \xrightarrow{+} b$ and $P_2 = b \xrightarrow{+} a$. Since C does not contain $idom(b)$, neither does P_1 which is therefore is an M-path, implying that $b \in M(a)$. Symmetrically,

$a \in M(b)$. Therefore, there is a non-trivial cycle containing nodes a and b in the M graph.

($\Leftarrow$) Assume the M graph has a non-trivial cycle. Let a and b be any two nodes on this cycle. From Lemma 3.11, $idom(a) = idom(b)$. By Theorem 2.3, there are non-trivial CFG paths $P_1 = a \xrightarrow{+} b$ which does not contain $idom(b)$ (equivalently, $idom(a)$), and $P_2 = b \xrightarrow{+} a$ which does not contain $idom(a)$ (equivalently, $idom(b)$). Therefore, the concatenation $C = P_1P_2$ is a CFG cycle containing a and b but not containing $idom(a)$ or $idom(b)$. Clearly, no node in C dominates all other nodes, so that CFG G is irreducible. $\square$

It can also be easily seen that the absence from M of self loops (which implies the absence of non-trivial cycles) characterizes acyclic programs.

3.4 Size of DF Relation

How large is DF? Since $DF \subseteq V \times V$, clearly $|DF| \leq |V|^2$. From Theorem 3.8, we see that an up-edge of the CFG generates a number of DF edges equal to one plus the difference between the levels of its endpoints in the dominator tree. If the dominator tree is deep and up-edges span many levels, then $|DF|$ can be considerably larger than $|E|$. In fact, it is not difficult to construct examples of sparse (*i.e.*, $|E| = O(|V|)$), structured CFGs, for which $|DF| = \Omega(|V|^2)$, proportional to the worst case. For example, it is easy to see that a program with a repeat-until loop nest with n loops such as the program shown in Figure 18 has a DF relation of size $n(n+1)/2$.

It follows that an algorithm that builds the entire DF graph to do ϕ-placement must take $\Omega(|V|^2)$ time, in the worst case. As we will see, it is possible to do better than this by building only those portions of the DF graph that are required to answer a ϕ-placement query.

4. TWO-PHASE ALGORITHMS

Two-phase algorithms compute the entire DF graph $G_{DF} = f(G)$ in a preprocessing phase before doing reachability computations $J(S) = g(S, G_{DF})$ to answer queries.

4.1 Edge scan algorithm

The edge scan algorithm (Figure 9) is essentially a direct translation of the expression for DF given by Theorem 3.8. A little care is required to achieve the time complexity of $T_p = O(|V| + |DF|)$ given in Proposition 4.1. Let v be the destination of a number of up-edges (say $u_1 \to v$, $u_2 \to v, \ldots$). A naive algorithm would first visit all the nodes in the interval $[u_1, idom(v))$ adding v to the DF set of each node in this interval, then visit all nodes in the interval $[u_2, idom(v))$ adding v to the DF sets of each node in this interval, etc. However, these intervals in general are not disjoint; if l is the least common ancestor of $u_1, u_2, ...$, nodes in the interval $[l, idom(v))$ will in general be visited once for each up-edge terminating at v, but only the first visit would do useful work. To make the preprocessing time proportional to the size of the DF sets, all up-edges that terminate at a given CFG node v are considered together. The DF sets at each node are maintained essentially as a stack in the sense that the first node of a (ordered) DF set is the one that was added most recently. The traversal of the nodes in interval $[u_k \to idom(v))$ checks each node to see if v is already in the DF set of that node by examining the first element of that DF set in constant time; if that element is v, the traversal is terminated.

Once the DF relation is constructed, procedure **ϕ-placement** is executed for each variable Z to determine, given the set S where Z is assigned, all nodes where ϕ-functions for Z are to be placed.

PROPOSITION 4.1. *The edge scan algorithm for SSA in Figure 9 has preprocessing time $T_p = O(|V| + |DF|)$, preprocessing space $S_p = O(|V| + |DF|)$, and query time $T_q = O(\sum_{v \in (S \cup M(S))} |DF(v)|)$.*

PROOF. In the preprocessing stage, time $O(|V| + |E|)$ is spent to visit the CFG, and additional constant time is spent for each of the $|DF|$ entries of (V, DF), for a total preprocessing time $T_p = O(|V| + |E| + |DF|)$ as described above. The term $|E|$ can be dropped from the last expression since $|E| = |E_{tree}| + |E_{up}| \leq |V| + |DF|$. The preprocessing space is that needed to store (V, DF). Query is performed by procedure ϕ-placement of Figure 9. Query time is proportional to the size of the portion of (V, DF) reachable from S. □

4.2 Node scan algorithm

The node scan algorithm (Figure 9) scans the nodes according to a bottom-up walk in the dominator tree and constructs the entire set $DF(w)$ when visiting w, following the approach in Theorem 3.9. The DF sets can be represented, *e.g.*, as linked lists of nodes; then, union and difference operations can be done in time proportional to the size of the operand sets, exploiting the fact that they are subsets of V. Specifically, we make use of an auxiliary Boolean array B, indexed by the elements of V and initialized to 0. To obtain the union of two or more sets, we scan the corresponding lists. When a node v is first encountered ($B[v] = 0$), it is added to the output list and then $B[v]$ is set to 1. Further occurrences of v are then detected ($B[v] = 1$) and are not appended to the output. Finally, for each v in the output list, $B[v]$ is reset to 0, to leave B properly initialized for further operations. Set difference can be handled by similar techniques.

PROPOSITION 4.2. *The node scan algorithm for SSA in Figure 9 has preprocessing time $T_p = O(|V| + |DF|)$, preprocessing space $S_p = O(|V| + |DF|)$, and query time $T_q = O(\sum_{v \in (S \cup M(S))} |DF(v)|)$.*

PROOF. Time $O(|V| + |E|)$ is required to walk over CFG edges and compute the α-DFsets for all nodes. In the bottom-up walk, the work performed at node w is bounded as follows:

$$work(w) \propto |\alpha(w)| + \sum_{c \in children(w)} |DF(c)| + |children(w)|.$$

Therefore, the total work for preprocessing is bounded by $O(|V| + |E| + |DF|)$ which, as before, is $O(|V| + |DF|)$. The preprocessing space is the space needed to store (V, DF). Query time is proportional to the size of the subgraph of (V, DF) that is reachable from S. □

```
Procedure EdgeScanDF(CFG, DominatorTree D):returns DF;
{
    Assume CFG = $(V, E)$;
    $DF = \{\}$;
    for each node $v$
        for each edge $e = (u \to v) \in E$ do
            if $u \neq idom(v)$ then
                $w = u$;
                while $(w \neq idom(v)) \& (v \notin DF(w))$ do
                    $DF(w) = DF(w) \cup \{v\}$;
                    $w = idom(w)$
            od
            endif
        od
    od
    return DF;
}
Procedure NodeScanDF(CFG,DominatorTree D):returns DF;
{
    Assume CFG = $(V, E)$;
    Initialize $DF(w) = \{\}$ for all nodes $w$;
    for each CFG edge $(u \to v)$ do
        if $(u \neq idom(v))$ $DF(u) = DF(u) \cup \{v\}$
    od
    for each node $w \in D$ in bottom-up order do
        $DF(w) = DF(w) \cup (\cup_{c \in children(w)} DF(c) - children(w))$;
    od
    return DF;
}
Procedure $\phi$-placement(DF,S):returns set of nodes where $\phi$-functions are needed;
{
    In $DF$, mark all nodes in set $S$;
    $M(S) = \{\}$;
    Enter all nodes in $S$ onto work-list $M$;
    while work-list $M$ is not empty do
        Remove node $w$ from $M$;
        for each node $v$ in $DF(w)$ do
            $M(S) = M(S) \cup \{v\}$;
            if $v$ is not marked then
                Mark $v$;
                Enter $v$ into work-list $M$;
            endif
        od
    od
    return $M(S)$;
}
```

Fig. 9. Edge scan and node scan algorithms

4.3 Discussion

Node scan is similar to the algorithm given by Cytron *et al.* [CFR+91]. As we can see from Propositions 4.1 and 4.2, the performance of two-phase algorithms is very sensitive to the size of the DF relation. We have seen in Section 3 that the size of the DF graph can be much larger than that of the CFG. However, real programs often have shallow dominator trees, hence their DF graph is comparable in size to the CFG; thus, two-phase algorithms may be quite efficient.

5. LOCK-STEP ALGORITHMS

In this section, we describe two *lock-step* algorithms that visit *all* the nodes of the CFG but compute only a subgraph $G'_{DF} = f'(G, S)$ of the DF graph that is sufficient to determine $J(S) = g'(S, G'_{DF})$. Specifically, the set reachable by non-empty paths that start at a node in S in G'_{DF} is the same as in G_{DF}. The f' and g' computations are interleaved: when a node v is reached through the portion of the DF graph already built, there is no further need to examine other DF edges pointing to v.

The set $DF^+(S)$ of nodes reachable from an input set S via non-empty paths can be computed efficiently in an acyclic DF graph, by processing nodes in topological order. At each step, a *pulling* algorithm would add the current node to $DF^+(S)$ if any of its predecessors in the DF graph belongs to S or has already been reached, *i.e.*, already inserted in $DF^+(S)$. A *pushing* algorithm would add the successors of current node to $DF^+(S)$ if it belongs to S or has already been reached.

The class of programs with an acyclic DF graph is quite extensive since it is identical to the class of reducible programs (Proposition 3.12). However, irreducible programs have DF graphs with non-trivial cycles, such as the one between nodes b and c in Figure 4(e). A graph with cycles can be conveniently preprocessed by collapsing into a "supernode" all nodes in the same strongly connected component, as they are equivalent as far as reachability is concerned [CLR92]. We show in Subsection 5.1 that it is possible to exploit Lemma 3.11 to compute a topological ordering of (the acyclic condensate of) the DF graph in $O(|E|)$ time, directly from the CFG, *without actually constructing the DF graph.* This ordering is exploited by the pulling and the pushing algorithms presented in subsequent subsections.

5.1 Topological Sorting of the DF and M Graphs

It is convenient to introduce the M-reduced CFG, obtained from a CFG G by collapsing nodes that are part of the same scc in the M graph of G. Figure 10 shows the M-reduced CFG corresponding to the CFG of Figure 4(a). The only non-trivial scc in the M graph (equivalently, in the DF graph) of the CFG in Figure 4(a) contains nodes b and c, and these are collapsed into a single node named bc in the M-reduced graph. The dominator tree for the M-reduced graph can be obtained by collapsing these nodes in the dominator tree of the original CFG.

Definition 5.1. Given a CFG $G = (V, E)$, the corresponding *M-reduced CFG* is the graph $\tilde{G} = (\tilde{V}, \tilde{E})$ where $\tilde{V}$ is the set of strongly connected components of M, and $(a \to b) \in \tilde{E}$ if and only if there is an edge $(u \to v) \in E$ such that $u \in a$ and $v \in b$.

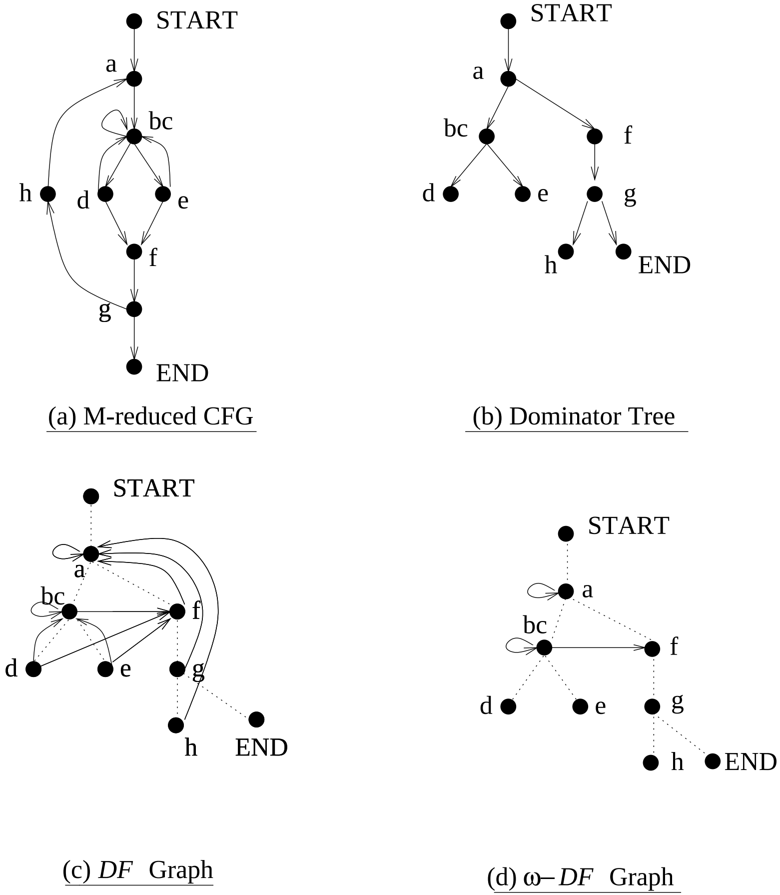


Fig. 10. M-reduced CFG corresponding to CFG of Figure 4(a)

Without loss of generality, the ϕ-placement problem can be solved on the reduced CFG. In fact, if $\tilde{M}$ denotes the merge relation in $\tilde{G}$, and $\tilde{w} \in \tilde{V}$ denotes the component to which w belongs, then $M(w) = \cup_{\tilde{x} \in \tilde{M}(\tilde{w})} \tilde{x}$ is the union of all the scc's $\tilde{x}$ reachable via $\tilde{M}$-paths from the scc $\tilde{w}$ containing w. The key observation permitting the efficient computation of scc's in the DF graph is Lemma 3.11, which states that all the nodes in a single scc of the DF graph are siblings in the dominator tree. Therefore, to determine scc's, it is sufficient to consider the subset of the DF graph, called the *ω-DF graph*, that is defined next.

Definition 5.2. The ω-DF relation of a CFG is the sub-relation of its DF relation that contains only those pairs (w, v) for which w and v are siblings in the dominator tree of that CFG.

Figure 4(f) shows the ω-DF graph for the running example. Figure 11 shows an algorithm for computing this graph.

```
Procedure ω-DF(CFG, DominatorTree);
{
    Assume CFG = (V, E);
    DF_ω = {};
    Stack = {};
    Visit(Root of DominatorTree);
    return G_ω = (V, DF_ω);

    Procedure Visit(u);
      Push u on Stack;
      for each edge e = (u → v) ∈ E do
        if u ≠ idom(v) then
          let c = node pushed after idom(v) on Stack;
          Append edge c → v to DF_ω;
        endif
      od
      for each child d of u do
        Visit(d); od
      Pop u from Stack;
}
```

Fig. 11. Building the ω-DF graph

LEMMA 5.3. *The ω-DF graph for CFG $G = (V, E)$ is constructed in $O(|E|)$ time by the algorithm in Figure 11.*

PROOF. From Theorem 3.8, we see that each CFG up-edge generates one edge in the ω-DF graph. Therefore, for each CFG up-edge $u \to v$, we must identify the child c of $idom(v)$ that is an ancestor of u, and introduce the edge $(c \to v)$ in the ω-DF graph. To do this in constant time per edge, we build the ω-DF graph while performing a depth-first walk of the dominator tree, as shown in Figure 11. This walk maintains a stack of nodes; a node is pushed on the stack when it is first encountered by the walk, and is popped from the stack when it is exited by the walk for the last time. When the walk reaches a node u, we examine all up-edges $u \to v$; the child of $idom(v)$ that is an ancestor of u is simply the node pushed after $idom(v)$ on the node stack. □

PROPOSITION 5.4. *Given the CFG $G = (V, E)$, its M-reduced version $\tilde{G} = (\tilde{V}, \tilde{E})$ can be constructed in time $O(|V| + |E|)$.*

PROOF. The steps involved are the following, each taking linear time:

1. Construct the dominator tree [BKRW98].
2. Construct the ω-DF graph (V, DF_ω) as shown in Figure 11.
3. Compute strongly connected components of (V, DF_ω) [CLR92].
4. Collapse each scc into one vertex and eliminate duplicate edges.

□

It is easy to see that the dominator tree of the M-reduced CFG can be obtained by collapsing the scc's of the ω-DF graph in the dominator tree of the original

CFG. For the CFG in Figure 4(a), the only non-trivial *scc* in the ω-DF graph is $\{b, c\}$, as is seen in Figure 4(f). By collapsing this scc, we get the M-reduced CFG and its dominator tree shown in Figure 10(a,b).

It remains to compute a topological sort of the DF graph of the M-reduced CFG (without building the DF graph explicitly). Intuitively, this is accomplished by topologically sorting the children of each node according to the ω-DF graph of the M-reduced CFG and concatenating these sets in some bottom-up order such as post-order in the dominator tree. We can describe this more formally as follows.

Definition 5.5. Given a M-reduced CFG $G = (V, E)$, let the children of each node in the dominator tree be ordered left to right according to a topological sorting of the ω-DF graph. A postorder visit of the dominator tree is said to yield an ω-**ordering** of G.

The ω-DF graph of the M-reduced CFG of the running example is shown in Figure 10(d). Note that the children of each node in the dominator tree are ordered so that the left to right ordering of the children of each node is consistent with a topological sorting of these nodes in the ω-DF graph. In particular, node *bc* is ordered before its sibling f. The postorder visit yields the sequence $< d, e, bc, h, g, f, a >$ which is a topological sort of the acyclic condensate of the DF graph of the original CFG in Figure 4(a).

THEOREM 5.6. *An ω-ordering of an M-reduced CFG $G = (V, E)$ is a topological sorting of the corresponding dominance frontier graph (V, DF) and merge graph (V, M) and it can be computed in time $O(|E|)$.*

PROOF. Consider an edge $(w \to v) \in DF$. We want to show that, in the ω-ordering, w precedes v.

From Theorem 3.8, it follows that there is a sibling s of v such that (i) s is an ancestor of w and (ii) there is an edge $(s \to v)$ in the DF (and ω-DF) graph. Since the ω-ordering is generated by a postorder walk of the dominator tree, w precedes s in this order; furthermore, s precedes v because an ω-ordering is a topological sorting of the ω-DF graph. Since $M = DF^+$, an ω-ordering is a topological sorting of the merge graphs as well. The time bound follows from Lemma 5.3, Proposition 5.4, Definition 5.5, and the fact that a postorder visit of a tree takes linear time. $\square$

From Proposition 3.12, it follows that for reducible CFG's, there is no need to determine the scc's of the ω-DF graph in order to compute ω-orderings.

5.1.1 *An Application to Weak Control Dependence.* In this subsection, we take a short detour to illustrate the power of the techniques just developed by applying these techniques to the computation of *weak control dependence*. This relation, introduced in [PC90], extends standard control dependence to include nonterminating program executions. We have shown in [BP96] that, in this context, the standard notion of postdominance must be replaced with the notion of loop postdominance. Furthermore, loop postdominance is transitive and its transitive reduction is a forest which can be obtained from the postdominator tree by disconnecting each node in a suitable set B from its parent. As it turns out, $B = J(K \cup \{\texttt{START}\})$, where K is the set of self-loops of the merge relation of the *reverse CFG*, which are called the *crowns*. The following proposition is concerned with the efficient computation of the self-loops of M.

PROPOSITION 5.7. *The self-loops of the M-graph for CFG $G = (V, E)$ can be found in $O(|V| + |E|)$.*

PROOF. It is easy to see that there is a self-loop for M at a node $w \in V$ if and only if there is a self-loop at $\tilde{w}$ (the scc containing w) in the M-reduced graph $\tilde{G} = (\tilde{V}, \tilde{E})$. By Proposition 5.4, $\tilde{G}$ can be constructed in time $O(|V| + |E|)$ and its self-loops can be easily identified in the same amount of time. □

When applied to the reverse CFG, Proposition 5.7 yields the set of crowns K. Then, $J(K \cup \{\texttt{START}\})$ can be obtained from $K \cup \{\texttt{START}\}$ by using any of the ϕ-placement algorithms presented in this paper, several of which also run in time $O(|V| + |E|)$. In conclusion, the loop postdominance forest can be obtained from the postdominator tree in time proportional to the size of the CFG. As shown in [BP96], once the loop postdominance forest is available, weak control dependence sets can be computed optimally by the algorithms of [PB97].

In the reminder of this section, we assume that the CFG is M-reduced.

5.2 Pulling Algorithm

The pulling algorithm (Figure 12) is a variation of the edge scan algorithm of Section 4.1. A bit-map representation is kept for the input set S and for the output set $J(S) = DF^+(S)$, which is built incrementally. We process nodes in ω-ordering and maintain, for each node u, an *off/on* binary tag, initially off and turned on when processing the first dominator of u which is $S \cup DF^+(S)$, denoted w_u. Specifically, when a node v is processed, either if it belongs to S or if it is found to belong to $DF^+(S)$, a top-down walk of the dominator subtree rooted at v is performed turning on all visited nodes. If we visit a node x already turned on, clearly the subtree rooted at x must already be entirely on, making it unnecessary to visit that subtree again. Therefore, the overall overhead to maintain the off/on tags is $O(|V|)$.

To determine whether to add a node v to $DF^+(S)$, each up-edge $u \to v$ incoming into v is examined: if u is turned on, then v is added and its processing can stop. Let **TurnOn**(D,w_u) be the call that has switched u on. Clearly, w_u belongs to the set $[u, idom(v))$ of the ancestors of u that precede v in ω-ordering which, by Theorem 3.8, is a subset of $DF^{-1}(v)$. Hence, v is correctly added to $DF^+(S)$ if and only if one of its DF predecessors (w_u) is in $S \cup DF^+(S)$. Such predecessor could be v itself, if $v \in S$ and there is a self-loop at v; for this reason, when $v \in S$, the call **TurnOn**(D,v) (Line 4) is made before processing the incoming edges. Clearly, the overall work to examine and process the up-edges is $O(|E_{up}|) = O(|E|)$. In summary, we have:

PROPOSITION 5.8. *The pulling algorithm for SSA of Figure 12 has preprocessing time $T_p = O(|V| + |E|)$, preprocessing space $S_p = O(|V| + |E|)$, and query time $T_q = O(|V| + |E|)$.*

Which subgraph $G'_{DF} = f'(G, S)$ of the DF graph gets (implicitly) built by the pulling algorithm? The answer is that, for each $v \in DF^+(S)$, G'_{DF} contains edge

```
Procedure Pulling(D,S); //D is dominator tree,S is set of assignment nodes
{
    Initialize DF+(S) to {};
    Initialize all nodes in dominator tree as off;

    for each node v in ω-ordering do
        if v ∈ S then TurnOn(D,v) endif ;
        for each up-edge u → v do
            if u is on then
                Add v to DF+(S);
                if v is off then TurnOn(D,v) endif ;
                break //exit inner loop
            endif
    od
}
ProcedureTurnOn(D, x);
{
    Switch x on;
    for each c ∈ children(x) in D do
        if c is off then TurnOn(D,c)
}
```

Fig. 12. Pulling algorithm

$(w_u \rightarrow v)$, where u is the first predecessor in the CFG adjacency list of node v that has been turned on when v is processed, and w_u is the ancestor that turned it on. As a corollary, G'_{DF} contains exactly $|DF^+(S)|$ edges.

5.3 Pushing Algorithm

The pushing algorithm (Figure 13) is a variation of the node scan algorithm in Section 4.2. It processes nodes in ω-ordering and builds $DF^+(S)$ incrementally; when a node $w \in S \cup DF^+(S)$ is processed, nodes in $DF(w)$ that are not already in set $DF^+(S)$ are added to it. A set $PDF(S, w)$, called the *pseudo-dominance frontier*, is constructed with the property that any node in $DF(w) - PDF(w)$ has already been added to $DF^+(S)$ by the time w is processed. Hence, it is sufficient to add to $DF^+(w)$ the nodes in $PDF(S, w) \cap DF(w)$, which are characterized by being after w in the ω-ordering. Specifically, $PDF(S, w)$ is defined (and computed) as the union of α-$DF(w)$ with the PDFs of those children of w that are not in $S \cup DF^+(S)$.

It is efficient to represent each PDF set as a singly linked list with a header that has a pointer to the start and one at the end of the list, enabling constant time concatenations. The union at Line 7 of procedure **Pushing** is implemented as list concatenation, hence in constant time per child for a global $O(|V|)$ contribution. The resulting list may have several entries for a given node, but each entry corresponds to a unique up-edge pointing at that node. If $w \in S \cup DF^+(S)$, then each node v in the list is examined and possibly added to $DF^+(S)$. Examination of each list entry takes constant time. Once examined, a list no longer contributes to the PDF set of any ancestor; hence the global work to examine lists is $O(|E|)$. In conclusion, the complexity bounds are as follows.

Procedure Pushing(S); S is set of assignment nodes

{

1: Initialize $DF^+(S)$ set to $\{\}$;

2: Initialize $\alpha\text{-}DF$ and PDF sets of all nodes to $\{\}$;

3: **for** each CFG edge $(u \to v)$ **do**

4: if $(u \neq idom(v))$ $\alpha\text{-}DF(u) = \alpha\text{-}DF(u) \cup \{v\}$

5: **od**

6: **for** each node w in ω-ordering **do**

7: $PDF(S,w) = \alpha\text{-}DF(w) \cup (\cup_{c \in (children(w) - S - DF^+(S))} PDF(c))$;

8: **if** $w \in S \cup DF^+(S)$ **then**

9: **for** each node v in $PDF(w)$ **do**

10: **if** $v >_\omega w$ and $v \notin DF^+(S)$ **then** Add v to $DF^+(S)$ **endif**

11: **endif** ;

12: **od**

}

Fig. 13. Pushing algorithm

PROPOSITION 5.9. *The pushing algorithm for ϕ-placement of Figure 13 is correct and has preprocessing time $T_p = O(|V|+|E|)$, preprocessing space $S_p = O(|V|+|E|)$, and query time $T_q = O(|V| + |E|)$.*

PROOF. Theorem 3.9 implies that a node the set $PDF(S,w)$ computed in Line 7 either belongs to $DF(w)$ or is dominated by w. Therefore, every node that is added to $DF^+(S)$ by Line 10, belongs to it (since $v <_\omega w$ implies that v is not dominated by w). We must also show that every node in $DF^+(S)$ gets added by this procedure. We proceed by induction on the length of the ω-ordering. The first node in such an ordering must be a leaf and, for a leaf w, $PDF(S,w) = DF(w)$. Assume inductively that for all nodes n before w in the ω-ordering, those in $DF(n) - PDF(S,n)$ are added. Since all the children of w precede it in the ω-ordering, it is easy to see that all nodes in $DF(w) - PDF(S,w)$ are added after w has been visited, satisfying the inductive hypothesis. □

The DF subgraph $G'_{DF} = f'(G,S)$ implicitly built by the pushing algorithm contains, for each $v \in DF^+(S)$, the DF edge $(w \to v)$ where w is the first node of $DF^{-1}(v) \cap (S \cup DF^+(S))$ occurring in ω-ordering. In general, this is a different subgraph from the one built by the pulling algorithm, except when the latter works on a CFG representation where the predecessors of each node are listed in ω-ordering.

5.4 Discussion

The ω-DF graph was introduced in [BP96] under the name of *sibling connectivity graph* to solve the problem of optimal computation of *weak control dependence* [PC90].

The pulling algorithm can be viewed as an efficient version of the reachability algorithm of Figure 7. At any node v, the reachability algorithm visits all nodes that are reachable from v in the reverse CFG along *paths* that do not contain $idom(v)$, while the pulling algorithm visits all nodes that are reachable from v in the reverse CFG along a *single edge* that does not contain (*i.e.*, originate from) $idom(v)$. The pulling algorithm achieves efficiency by processing nodes in ω-order, which ensures that information relevant to v can be found by traversing single edges rather than entire paths. It is the simplest ϕ-placement algorithm that achieves linear worst-case bounds for all three measures T_p, S_p and T_q.

For the pushing algorithm, the computation of the M-reduced graph can be eliminated and nodes can simply be considered in bottom-up order in the dominator tree, at the cost of having to revisit a node if it gets marked after it has been visited for computing its PDF set.

Reif and Tarjan [RT81] proposed a lock-step algorithm that combined ϕ-placement with the computation of the dominator tree. Their algorithm is a modification of the Lengauer and Tarjan algorithm which computes the dominator tree in a bottom-up fashion [LT79]. Since the pushing algorithm traverses the dominator tree in bottom-up order, it is possible to combine the computation of the dominator tree with pushing to obtain ϕ-placement in $O(|E|\alpha(|E|))$ time per variable. Cytron and Ferrante have described a lock-step algorithm which they call *on-the-fly* computation of merge sets [CF93], with $O(|E|\alpha(|E|))$ query time. Their algorithm is considerably more complicated than the pushing and pulling algorithms described here, in part because it does not use ω-ordering.

6. LAZY ALGORITHMS

A drawback of lock-step algorithms is that they visit all the nodes in the CFG, including those that are not in $M(S)$. In this section, we discuss algorithms that compute sets $EDF(w)$ *lazily*, *i.e.*, only if w belongs to $M(S)$, potentially saving the effort to process irrelevant parts of the DF graph. Lazy algorithms have the same the asymptotic complexity as lock-step algorithms, but outperform them in practice (Section 7).

We first discuss a lazy algorithm that is optimal for computing EDF sets, based on the approach of [PB95; PB97] to compute the control dependence relation of a CFG. Then, we apply these results to ϕ-placement. The lazy algorithm works for arbitrary CFGs (*i.e.*, M-reduction is not necessary).

6.1 $\mathcal{ADT}$: The Augmented Dominator Tree

One way to compute $EDF(w)$ is to appeal directly to Definition 3.4: traverse the dominator subtree rooted at w and for each visited node u and edge $(u \to v)$, output edge $(u \to v)$ if w does not strictly dominate v. Pseudocode for this query procedure, called **TopDownEDF**, is shown in Figure 14. Here, each node u is assumed to have a node list L containing all the targets of up-edges whose source is u (*i.e.*, α-$DF(u)$). The **Visit** procedure calls itself recursively, and the recursion terminates when it encounters a *boundary node. For now, boundary nodes coincide with the leaves of tree. However, we shall soon generalize the notion of boundary node in a critical way.* For the running example of Figure 4, the call $EDF(a)$ would visit nodes $\{a, b, d, c, e, f, g, h, \texttt{END}\}$ and output edge $(h \to a)$ to answer the EDF query.

This approach is *lazy* because the EDF computation is done only when it is required to answer the query. The **TopDownEDF** procedure takes time $O(|E|)$ since, in the worst case, the entire dominator tree has to be visited and all the edges in the CFG have to be examined. To decrease query time, one can take an

```
Procedure TopDownEDF(QueryNode);
{
1:    EDF = {};
2:    Visit(QueryNode, QueryNode);
3:    return EDF;
}
Procedure Visit(QueryNode, VisitNode);
{
1:    for each edge (u → v) ∈ L[VisitNode] do
2:      if idom(v) is a proper ancestor of QueryNode
3:        then EDF = EDF ∪ {(u → v)}; endif
4:      od ;
5:    if VisitNode is not a boundary node
6:      then
7:        for each child C of VisitNode
8:          do
9:            Visit(QueryNode,C)
10:         od ;
11:   endif ;
}
```

Fig. 14. Top-down query procedure for EDF

eager approach by precomputing the entire EDF graph, storing each $EDF(w)$ in list $L(w)$, and letting *every* node be a boundary node. We still use **TopDownEDF** to answer a query. The query would visit only the queried node w and complete in time $O(|EDF(w)|)$. This is essentially the two-phase approach of Section 4 — the query time is excellent but the preprocessing time and space requirements are $O(|V| + |EDF|)$.

As a tradeoff between fully eager and fully lazy evaluation, we can arbitrarily partition V into boundary and interior nodes; **TopDownEDF** will work correctly if $L(w)$ is initialized as follows:

Definition 6.1. $L[w] = EDF(w)$ if w is a *boundary* node and $L[w] = \alpha\text{-}EDF(w)$ if w is an *interior* node.

In general, we will assume that leaves are boundary nodes, to ensure proper termination of recursion (this choice has no consequence on $L[w]$ since, for a leaf, $EDF(w) = \alpha\text{-}DF(w)$.) The correctness of **TopDownEDF** is argued next. It is easy to see that if edge $(u \to v)$ is added to EDF by Line 3 of **Visit**, then it does belong to $EDF(w)$. Conversely, let $(u \to v) \in EDF(w)$. Consider the dominator tree path from w to u. If there is no boundary node on this path, then procedure **TopDownEDF** outputs $(u \to v)$ when it visits u. Else, let b be the first boundary node on this path: then $(u \to v) \in EDF(b)$ and it will be output when the procedure visits b.

So far, no specific order has been assumed for the edges $(u_1 \to v_1), (u_2 \to v_2), \ldots$ in list $L[w]$. We note from Lemma 3.10 that $idom(v_1), idom(v_2), \ldots$ dominate w and are therefore totally ordered by dominance. To improve efficiency, the edges in $L[w]$ are ordered so that, in the sequence $idom(v_1), idom(v_2), \ldots$, a node appears after its ancestors. Then, the examination loop of Line 1 in procedure **TopDownEDF**

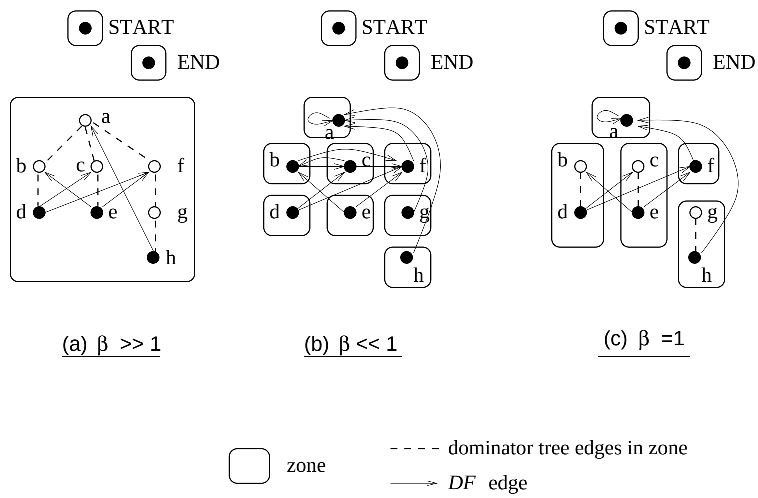


Fig. 15. Zone structure for different values of β

can terminate as soon as a node v is encountered where $idom(v)$ does not strictly dominate the query node.

Different choices of boundary nodes (solid dots) and interior nodes (hollow dots) are illustrated in Figure 15. Figure 15(a) shows one extreme in which only START and the leaves are boundary nodes. Since $EDF(\mathtt{START}) = \emptyset$ and $EDF(w) = \alpha\text{-}DF(w)$ for any leaf w, by Definition 6.1, only α-EDF edges are stored explicitly, in this case. Figure 15(b) shows the other extreme in which all nodes are boundary nodes, hence all EDF edges are stored explicitly. Figure 15(c) shows an intermediate point where the boundary nodes are $\mathtt{START}, \mathtt{END}, a, d, e, f$, and h.

If the edges from a boundary node to any of its children, which are never traversed by procedure **TopDownEDF**, are deleted, the dominator tree becomes partitioned into smaller trees called *zones*. For example, in Figure 15(c), there are seven zones, with node sets : $\{\mathtt{START}\}$, $\{\mathtt{END}\}$, $\{a\}$, $\{b, d\}$, $\{c, e\}$, $\{f\}$, $\{g, h\}$. A query **TopDownEDF**(q) visits the portion of a zone below node q, which we call the *sub-zone* associated with q. Formally:

Definition 6.2. A node w is said to be in the *sub-zone* Z_q associated with a node q if (i) w is a descendant of q, and (ii) the path [q,w) does not contain any boundary nodes. A *zone* is a maximal sub-zone; that is, a sub-zone that is not strictly contained in any other sub-zone.

In the implementation, we assume that for each node there is a boolean variable $Bndry?$ set to true for boundary nodes and set to false for interior nodes. In Line 2 of Procedure **Visit**, testing whether $idom(v)$ is a proper ancestor of $QueryNode$ can be done in constant time by comparing their *dfs* (depth-first search) number or their *level* number. (Both numbers are easily obtained by preprocessing; the dfs number is usually already available as a byproduct of dominator tree construction.) It follows immediately that the query time Q_q is proportional to the sum of the number of visited nodes and the number of reported edges:

$$Q_q = O(|Z_q| + |EDF(q)|). \tag{4}$$

To limit query time, we shall define zones so that, in terms of a design parameter β (a positive real number), for every node q we have:

$$|Z_q| \leq \beta |EDF(q)| + 1 \ . \tag{5}$$

Intuitively, the number of nodes visited when q is queried is at most one more than some constant proportion of the answer size. We observe that, when $EDF(q)$ is empty (*e.g.*, when $q =$ `START` or when $q =$ `END`), Condition (5) forces $Z_q = \{q\}$, for any β.

By combining Equations (4) and (5), we obtain

$$Q_q = O((\beta + 1)|EDF(q)|). \tag{6}$$

Thus, for constant β, query time is linear in the output size, hence asymptotically optimal. Next, we consider space requirements.

6.1.1 *Defining Zones.* Can we define zones so as to satisfy Inequality (5) and simultaneously limit the extra space needed to store an up-edge $(u \to v)$ at each boundary node w dominating u and properly dominated by v? A positive answer is provided by a simple bottom-up, greedy algorithm that makes zones as large as possible subject to Inequality (5) and to the condition that the children of a given node are either all in separate zones or all in the same zone as their parent [5]. More formally:

Definition 6.3. If node v is a leaf or $(1 + \sum_{u \in children(v)} |Z_u|) > (\beta |EDF(v)| + 1)$, then v is a *boundary* node and Z_v is $\{v\}$. Else, v is an *interior* node and Z_v is $\{v\} \cup_{u \in children(v)} Z_u$.

The term $(1 + \sum_{u \in children(v)} |Z_u|)$ is the number of nodes that would be visited by a query at node v if v were made an interior node. If this quantity is larger than $(\beta |EDF(v)| + 1)$, Inequality (5) fails, so we make v a boundary node.

To analyze the resulting storage requirements, let X denote the set of boundary nodes that are not leaves. If $w \in (V - X)$, then only α-DF edges out of w are listed in $L[w]$. Each up-edge in E_{up} appears in the list of its bottom node and, possibly, in the list of some other node in X. For a boundary node w, $|L[w]| = |EDF(w)|$. Hence, we have:

$$\sum_{w \in V} |L[w]| = \sum_{w \in (V-X)} |L[w]| + \sum_{w \in X} |L[w]| \leq |E_{up}| + \sum_{w \in X} |EDF(w)|. \tag{7}$$

From Definition 6.3, if $w \in X$, then

$$|EDF(w)| < \sum_{u \in children(w)} |Z_u| / \beta. \tag{8}$$

When we sum over $w \in X$ both sides of Inequality (8), we see that the right hand side evaluates at most to $|V|/\beta$, since all sub-zones Z_u's involved in the resulting

[5] The removal of this simplifying condition might lead to further storage reductions.

double summation are disjoint. Hence, $\sum_{w \in X} |EDF(w)| \leq |V|/\beta$, which, used in Relation (7) yields:

$$\sum_{w \in V} |L[w]| \leq |E_{up}| + |V|/\beta. \tag{9}$$

Therefore, to store this data structure, we need $O(|V|)$ space for the dominator tree, $O(|V|)$ further space for the *Bndry*? bit and for list headers, and finally, from Inequality (9), $O(|E_{up}| + |V|/\beta)$ for the list elements. All together, we have $S_p = O(|E_{up}| + (1 + 1/\beta)|V|)$.

We summarize the *Augmented Dominator Tree* $\mathcal{ADT}$ for answering EDF queries:

1. T: dominator tree that permits top-down and bottom-up traversals.
2. $dfs[v]$: dfs number of node v.
3. $Bndry?[v]$: boolean. Set to true if v is a boundary node, and set to false otherwise.
4. $L[v]$: list of CFG edges. If v is a boundary node, $L[v]$ is $EDF(v)$; otherwise, it is α-$DF(v)$.

6.1.2 *$\mathcal{ADT}$ Construction.* The preprocessing algorithm that constructs the search structure $\mathcal{ADT}$ takes three inputs:

- The dominator tree T, for which we assume that the relative order of two nodes one of which is an ancestor of the other can be determined in constant time.
- The set E_{up} of up-edges $(u \to v)$ ordered by $idom(v)$.
- Real parameter $\beta > 0$, which controls the space/query-time tradeoff.

The stages of the algorithm are explained below and translated into pseudocode in Figure 16.

1. *For each node x, compute the number $b[x]$ (respectively, $t[x]$) of up-edges $(u \to v)$ with $u = x$ (respectively, $idom(v) = x$).* Set up two counters initialized to zero and, for each $(u \to v) \in E_{up}$, increment the appropriate counters of its endpoints. This stage takes time $O(|V| + |E_{up}|)$, for the initialization of the $2|V|$ counters and for the $2|E_{up}|$ increments of such counters.
2. *For each node x, compute $|EDF(x)|$.* It is easy to see that $|EDF(x)| = b[x] - t[x] + \sum_{y \in children(x)} |EDF(y)|$. Based on this relation, the $|EDF(x)|$ values can be computed in bottom-up order, using the values of $b[x]$ and $t[x]$ computed in Step 1, in time $O(|V|)$.
3. *Determine boundary nodes,* by appropriate setting of a boolean variable $Bndry?[x]$ for each node x. Letting $z[x] = |Z_x|$, Definition 6.3 becomes:
   If x is a leaf or $(1 + \sum_{y \in children(x)} z[y]) > (\beta|EDF(x)| + 1)$, then x is a boundary node, and $z[x]$ is set to 1. Otherwise, x is an interior node, and $z[x] = (1 + \sum_{y \in children(x)} z[y])$.
   Again, $z[x]$ and $Bndry?[x]$ are easily computed in bottom-up order, taking time $O(|V|)$.
4. *Determine, for each node x, the next boundary node $NxtBndry[x]$ in the path from x to the root.* If the parent of x is a boundary node, then it is the next boundary for x. Otherwise, x has the same next boundary as its parent. Thus, $NxtBndry[x]$ is easily computed in top-down order, taking $O(|V|)$ time. The

next boundary for root of T set to a conventional value $-\infty$, considered as a proper ancestor of any node in the tree.

5. *Construct list $L[x]$ for each node x.* By Definition 6.1, given an up-edge $(u \to v)$, v appears in list $L[x]$ for $x \in W_{uv}\{w_0 = u, w_1, ..., w_k\}$, where W_{uv} contains u as well as all boundary nodes contained in the dominator-tree path $[u, idom(v))$ from u (included) to $idom(v)$ (excluded). Specifically, $w_i = NxtBndry[w_{i-1}]$, for $i = 1, 2, ..., k$ and w_k is the proper descendant of $idom(v)$ such that $idom(v)$ is a descendant of $NxtBndry[w_k]$.
Lists $L[x]$'s are formed by scanning the edges $(u \to v)$ in E_{up} in decreasing order of $idom(v)$. Each node v is appended at the end of (the constructed portion of) $L[x]$ for each x in W_{uv}. This procedure ensures that, in each list $L[x]$, nodes appear in decreasing order of $idom(v)$.
This stage takes time proportional to the number of append operations, which is $\sum_{x \in V} |L[x]| = O(|E_{up}| + |V|/\beta)$.

In conclusion, the preprocessing time is $T = O(|E_{up}| + (1 + 1/\beta)|V|)$. The developments of the present subsection are summarized in the following theorem.

THEOREM 6.4. *Given a CFG, the corresponding augmented dominator tree can be constructed in time $T_p = O(|E_{up}| + (1 + 1/\beta)|V|)$ and stored in space $S_p = O(|E_{up}| + (1 + 1/\beta)|V|)$. A query to the edge dominance frontier of a node q can be answered in time $Q_q = O((\beta + 1)|EDF(q)|)$.*

6.1.3 *The Role of β.* Parameter β essentially controls the degree of caching of EDF information. For a given CFG, as β increases, the degree of caching and space requirements decrease while query time increases. However, for a fixed β, the degree of caching adapts to the CFG being processed in a way that guarantees linear performance bounds. To take a closer look at the role of β, it is convenient to consider two distinguished values associated with each CFG G.

Definition 6.5. Given a CFG $G = (V, E)$, let Y be the set of nodes q such that (i) q is not a leaf of the dominator tree, and (ii) $EDF(q) \neq \emptyset$. Let D_q be the set of nodes dominated by q.

We define two quantities $\beta_1(G)$ and $\beta_2(G)$ as follows. [6] :

$$\beta_1(G) = 1/\max_{q \in Y} |EDF(q)| \tag{10}$$

and

$$\beta_2(G) = \max_{q \in Y}(|D_q| - 1)/|EDF(q)|. \tag{11}$$

Since, for $q \in Y$, $1 \leq |EDF(q)| < |E|$ and $2 \leq D_q < |V|$, it is straightforward to show that

$$1/|E| \; < \; \beta_1(G) \; \leq \; 1, \tag{12}$$
$$1/|E| \; < \; \beta_2(G) \; \leq \; |V|, \tag{13}$$
$$\beta_1(G) \; \leq \; \beta_2(G). \tag{14}$$

[6] Technically, we assume Y is not empty, a trivial case that, under Definition A.1, arises only when the CFG consists of a single path from `START` to `END`.

```
Procedure BuildADT(T: dominator tree, E_up: array of up-edges, β: real);
{
// b[x]/t[x]: number of up-edges u → v with u/idom(v) equal x
for each node x in T do
   b[x] := t[x] := 0; od
for each up-edge u → v in E_up do
   Increment b[u];
   Increment t[idom(v)];
od ;
//Determine boundary nodes.
for each node x in T in bottom-up order do
   //Compute output size when x is queried.
   a[x] := b[x] - t[x] + Σ_{y∈children(x)} a[y];
   z[x] := 1 + Σ_{y∈children(x)} z[y]; //Tentative zone size.
   if (x is a leaf) or (z[x] > β * a[x] + 1)
      then // Begin a new zone
         Bndry?[x] := true;
         z[x] := 1;
      else //Put x into same zone as its children
         Bndry?[x] := false;
   endif
od ;
// Chain each node to the first boundary node that is an ancestor.
for each node x in T in top-down order do
   if x is root of dominator tree
      then NxtBndry[x] := - ∞;
      else if Bndry?[idom(x)]
         then NxtBndry[x] := idom(x);
         else NxtBndry[x] := NxtBndry[idom(x)];
         endif
   endif
od
// Build the lists L[x]
for each up-edge (u → v) do
   w := u;
   while idom(v) properly dominates w do
      append v to end of list L[w];
      w := NxtBndry[w];
   od
}
```

Fig. 16. Constructing the $\mathcal{ADT}$ structure

With a little more effort, it can also be shown that each of the above bound is achieved, to within constant factors, by some family of CFGs.

Next, we argue that the values $\beta_1(G)$ and $\beta_2(G)$ for parameter β correspond to extreme behaviors for the $\mathcal{ADT}$. We begin by observing that, by Definition 6.3, if $q \notin Y$, then q is a boundary node of the $\mathcal{ADT}$, for any value of β. Furthermore, $EDF(q) = \alpha\text{-}EDF(q)$.

When $\beta < \beta_1(G)$, the $\mathcal{ADT}$ stores the full EDF relation. In fact, in this case, the right-hand-side of Condition (5) is strictly less than 2 for all $q's$. Hence, each node is a boundary node.

When $\beta \geq \beta_2(G)$, the $\mathcal{ADT}$ stores the α-EDFrelation. In fact, in this case, each $q \in Y$ is an interior node, since the right-hand side of Condition (5) is no smaller than $|D_q|$, thus permitting Z_q to contain all descendants of q.

Finally, in the range $\beta_1(G) \leq \beta < \beta_2(G)$, one can expect intermediate behaviors where the $\mathcal{ADT}$ stores something in between α-EDF and EDF.

To obtain linear space and query time, β must be chosen to be a constant, independent of G. A reasonable choice can be $\beta = 1$, illustrated in Figure 15(c) for the running example. Depending on the values of $\beta_1(G)$ and $\beta_2(G)$, this choice can yield anywhere from no caching to full caching. For many CFG's arising in practice, $\beta_1(G) < 1 < \beta_2(G)$; for such CFG's, $\beta = 1$ corresponds to an intermediate degree of caching.

6.2 Lazy Pushing Algorithm

We now develop a lazy version of the the pushing algorithm. Preprocessing consists in constructing the $\mathcal{ADT}$ data structure. The query to find $J(S) = DF^+(S)$ proceeds along the following lines:

- The successors $DF(w)$ are determined only for nodes $w \in S \cup J(S)$.
- Set $DF(w)$ is obtained by a query $EDF(w)$ to the $\mathcal{ADT}$, modified to avoid reporting of some nodes already found to be in $J(S)$.
- The elements of $J(S)$ are processed according to a bottom-up ordering of the dominator tree.

To develop an implementation of the above guidelines, consider first the simpler problem where a set $I \subseteq V$ is given, with its nodes listed in order of non increasing level, and the set $\cup_{w \in I} EDF(w)$ must be computed. For each element of I in the given order, an EDF query is made to the $\mathcal{ADT}$. To avoid visiting tree nodes repeatedly during different EDF queries, a node is marked when it is queried and the query procedure of Figure 14 is modified so that it never visits nodes below a marked node. The time $T'_q(I)$ to answer this simple form of query is proportional to the size of the set $V_{vis} \subseteq V$ of nodes visited and the total number of up-edges in the $L[v]$ lists of these nodes. Considering Bound 9 on the latter quantity, we obtain

$$T'_q(I) = O(|V_{vis}| + |E_{up}| + |V|/\beta) = O(|E| + (1 + 1/\beta)|V|). \tag{15}$$

For constant β, the above time bound is proportional to program size.

In our context, set $I = I(S) = S \cup DF^+(S)$ is not given directly; rather, it must be incrementally constructed and sorted, from input S. This can be accomplished by keeping those nodes already discovered to be in I but not yet queried for EDF in a priority queue [CLR92], organized by level number in the tree. Initially, the queue contains only the nodes in S. At each step, a node w of highest level is extracted from the priority queue and an $EDF(w)$ query is made in the $\mathcal{ADT}$; if a reported node v is not already in the output set, it is added to it as well as inserted into the queue. From Lemma 3.10, $level(v) \leq level(w)$, hence the level number is non increasing throughout the entire sequence of extractions from the priority queue. The algorithm is described in Figure 17. Its running time can be expressed as

$$T_q(S) = T'_q(I(S)) + T_{PQ}(I(S)). \tag{16}$$

The first term accounts for the $\mathcal{ADT}$ processing and satisfies Equation 15. The second term accounts for priority queue operations. The range for the keys has size K, equal to the number of levels of the dominator tree. If the priority queue is implemented using a heap, the time per operation is $O(\log K)$ [CLR92], whence $T_{PQ}(I(S)) = O(|I(S)| \log K)$. A more sophisticated data structure, exploiting the integer nature of the keys, achieves $O(\log \log K)$ time per operation [VEBKZ77], hence $T_{PQ}(I(S)) = O(|I(S)| \log \log K)$.

A simpler implementation, which exploits the constraint on insertions, consists of an array A of K lists, one for each possible key in decreasing order. An element with key r is inserted, in time $O(1)$, by appending it to list $A[r]$. Extraction of an element with maximum key entails scanning the array from the component where the last extraction has occurred to the first component whose list is not empty. Clearly, $T_{PQ}(I(S)) = O(|I(S)| + K) = O(|V|)$. Using this result together with Equation 15 in Equation 16, the SSA query time can be bounded as

$$T_q(S) = O(|E| + (1 + 1/\beta)|V|). \tag{17}$$

The DF subgraph $G'_{DF} = f'(G, S)$ implicitly built by the lazy pushing algorithm contains, for each $v \in DF^+(S)$, the DF edge $(w \to v)$ where w is the first node of $DF^{-1}(v) \cap (S \cup DF^+(S))$ occurring in the processing ordering. This ordering is sensitive to the specific way the priority queue is implemented and ties between nodes of the same level are broken.

7. EXPERIMENTAL RESULTS

In this section, we evaluate the lazy pushing algorithm of Figure 17 experimentally, focusing on the impact that the choice of parameter β has on performance. These experiments shed light on the two-phase and fully lazy approaches because the lazy algorithm reduces to these approaches for extreme values of β, as explained in Section 6.1.3. Intermediate values of β in the lazy algorithm let us explore tradeoffs between preprocessing time (a decreasing function of β) and query time (an increasing function of β).

The programs used in these experiments include a standard model problem and the SPEC92 benchmarks. The SPEC programs tend to have sparse dominance frontier relations, so we can expect a two-phase approach to benefit from small query time without paying much penalty in preprocessing time and space; in contrast, the fully lazy approach might be expected to suffer from excessive recomputation of dominance frontier information. The standard model problem on the other hand exhibits a dominance frontier relation that grows quadratically with program size, so we can expect a two-phase approach to suffer considerable overhead, while a fully lazy algorithm can get by with little preprocessing effort. The experiments support these intuitive expectations and at the same time show that intermediate values of β (say, $\beta = 1$) are quite effective for all programs.

Next, we describe the experiments in more detail.

A model problem for SSA computation is a nest of l repeat-until loops, whose CFG we denote G_l, illustrated in Figure 18. Even though G_l is structured, its DF

```
Procedure φ-placement (S: set of nodes);
{
      // ADT data structure, is global
      Initialize a Priority Queue PQ;
      DF+(S) = {}; Set of output nodes (global variable)
      Insert nodes in set S into PQ; //key is level in tree
      In tree T, mark all nodes in set S;

      while PQ is not empty do
         w := ExtractMax(PQ); //w is deepest in tree
         QueryIncr(w);
      od ;
      Delete marks from nodes in T;
      Output DF+(S);
}
Procedure QueryIncr(QueryNode);
{
      VisitIncr(QueryNode, QueryNode);
}
Procedure VisitIncr(QueryNode,VisitNode);
{
      for each node v in L[VisitNode]
         in list order do
            if idom(v) is strict ancestor of QueryNode
            then
               DF+(S) = DF+(S) ∪ {v};
               if v is not marked
                  then
                     Mark v;
                     Insert v into PQ;
               endif ;
            else break ; // exit from the loop
         od ;
      if VisitNode is not a boundary node
         then
            for each child C of VisitNode
               do
                  if C is not marked
                     then VisitIncr(QueryNode,C);
               od ;
      endif ;
}
```

Fig. 17. Lazy pushing algorithm, based on $\mathcal{ADT}$

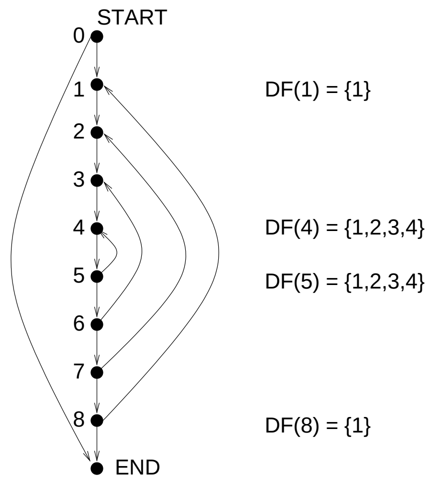


Fig. 18. Repeat-until loop nest G_4

relation grows quadratically with program size, making it a worst-case scenario for two-phase algorithms. The experiments reported here are based on G_{200}. Although a 200-deep loop nest is unlikely to arise in practice, it is large enough to exhibit the differences between the algorithms discussed in this paper. We used the lazy pushing algorithm to compute $DF^+(n)$ for different nodes n in the program, and measured the corresponding running time as a function of β on a SUN-4. In the 3D plot in Figure 19, the x axis is the value of $log_2(\beta)$, the y-axis is the node number n, and the z-axis is the time for computing $DF^+(n)$.

The 2D plot in Figure 18 shows slices parallel to the yz plane of the 3D plot for three different values of β - a very large value (Sreedhar-Gao), a very small value (Cytron et al), and 1.

From these plots, it is clear that for small values of β (full caching/two-phase), the time to compute DF^+ grows quadratically as we go from outer loop nodes to inner loop nodes. In contrast, for large values of β (no caching/fully lazy), this time is essentially constant. These results can be explained analytically as follows.

The time to compute DF^+ sets depends on the number of nodes and the number of DF graph edges that are visited during the computation. It is easy to show that, for $1 \leq n \leq l$, we have $DF(n) = DF(2l - n + 1) = \{1, 2, ..., n\}$.

For very small values of β, the dominance frontier information of every node is stored at that node (full caching). For $1 \leq n \leq l$, computing $DF^+(n)$ requires a visit to all nodes in the set $\{1, 2, ..., n\}$. The number of DF edges examined during these visits is $1 + 2 + \ldots + n = n(n+1)/2$; each of these edge traversals involves a visit to the target node of the DF edge. The reader can verify that a symmetric formula holds for nodes numbered between l and $2l$. These results explain the quadratic growth of the time for DF^+ set computation when full caching is used.

For large values of β, we have no caching of dominance frontier information. Assume that $1 \leq n \leq l$. To compute $DF(n)$, we visit all nodes in the dominator tree subtree below n, and traverse l edges to determine that $DF(n) = \{1, 2, ..., n\}$.

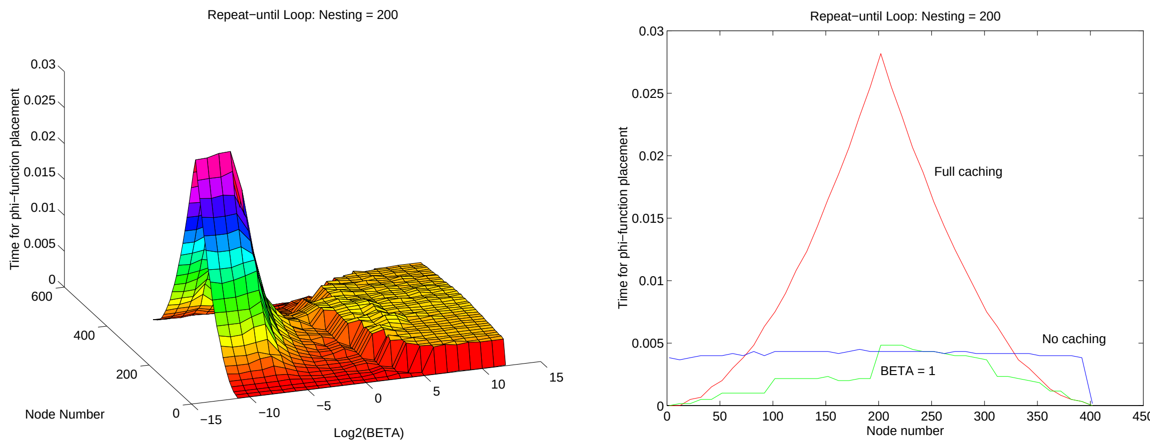


Fig. 19. Time for ϕ-placement in model problem G_{200} by lazy pushing with parameter β

Subsequently, we visit nodes $(n-1)$, $(n-2)$ etc., and at each node, we visit only that node and the node immediately below it (which is already marked); since no DF edges are stored at these nodes, we traverse no DF edges during these visits. Therefore, we visit $(3l+n)$ nodes, and traverse l edges. Since n is small compared to $3l$, we see that the time to compute $DF^+(n)$ is almost independent of n, which is borne out by the experimental results.

Comparing the two extremes, we see that for small values of n, full caching performs better than no caching. Intuitively, this is because we suffer the overhead of visiting all nodes below n to compute $DF(n)$ when there is no caching; with full caching, the DF set is available immediately at the node. However, for large values of n, full caching runs into the problem of repeatedly discovering that certain nodes are in the output set — for example, in computing $DF^+(n)$, we find that node 1 is in the output set when we examine $DF(m)$ for every m between n and 1. It is easy to see that with no caching, this discovery is made exactly once (when node $2l$ is visited during the computation of $DF^+(n)$). The cross-over value of n at which no caching performs better than full caching is difficult to estimate analytically but from Figure 19, we see that a value of $\beta = 1$ outperforms both extremes for almost all problem sizes.

Since deeply nested control structures are rare in real programs, we would expect the time required for ϕ-function placement in practice to look like a slice of Figure 19 parallel to the xz plane for a small value of n. That is, we would expect full caching to outperform no caching, and we would expect the use of $\beta = 1$ to outperform full caching by a small amount. Figure 20 shows the total time required to do ϕ-function placement for all unaliased scalar variables in all of the programs in the SPEC92 benchmarks. It can be seen that full caching (small β) outperforms no caching (large β) by a factor between 3 and 4. In [SG95], Sreedhar and Gao reported that their algorithm, essentially lazy pushing with no caching, outperformed the Cytron et al algorithm by factors of 5 to 10 on these benchmarks. These measurements were

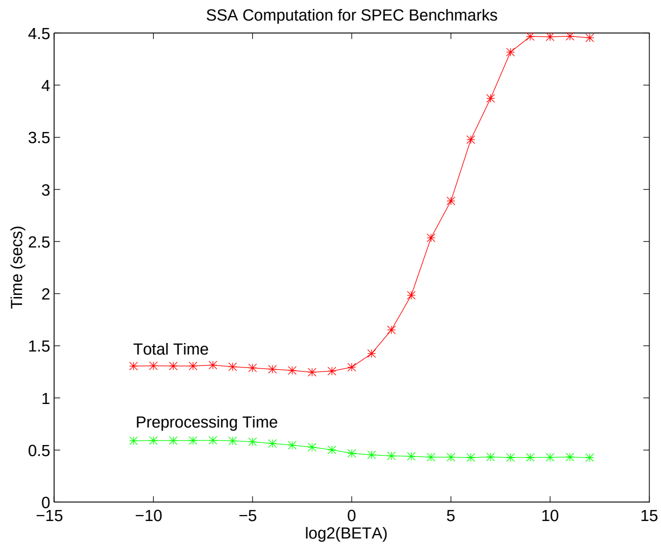


Fig. 20. Time for ϕ-placement in SPEC92 benchmarks by lazy pushing with parameter β

apparently erroneous, and new measurements taken by them are in line with our numbers [SG96]. Using $\beta = 1$ gives the best performance, although the advantage over full caching is small in practice.

Other experiments we performed showed that lock-step algorithms were not competitive with two phase and lazy algorithms because of the overhead of preprocessing which requires finding strongly connected components and performing topological sorting. The pulling algorithm is a remarkably simple ϕ-placement algorithm that achieves linear space and time bounds for preprocessing and query, but for these benchmarks, for example, the time it took for ϕ-placement was almost 10 seconds, an order of magnitude slower than the best lazy pushing algorithm.

Therefore, for practical intra-procedural SSA computation, we recommend the lazy pushing algorithm based on the $\mathcal{ADT}$ with a value of $\beta = 1$ since its implementation is not much more complicated than that of two-phase algorithms.

8. ϕ-PLACEMENT FOR MULTIPLE VARIABLES IN STRUCTURED PROGRAMS

The ϕ-placement algorithms presented in the previous sections are quite efficient, and indeed asymptotically optimal when only one variable is processed for a given program. However, when several variables must be processed, the query time T_q for each variable could be improved by suitable preprocessing of the CFG. Clearly, query time satisfies the lower bound

$$T_q = \Omega(|S| + |J(S)|),$$

where $J(S) = \cup_{x \in S} J(x)$, because $|S|$ and $|J(S)|$ are the input size and the output size of the query, respectively. The quantity $|S|+|J(S)|$ can be considerably smaller than $|E|$.

Achieving optimal, *i.e.*, $O(|S| + |J(S)|)$, query time for arbitrary programs is not a trivial task, even if we are willing to tolerate high preprocessing costs in time and space. For instance, let $R^+ = M$. Then, a search in the graph (V, R) starting at the nodes in S will visit a subgraph $(S \cup J(S), E_S)$ in time $T_q = O(|S| + |J(S)| + |E_S|)$. Since $|E_S|$ can easily be the dominating term in the latter sum, T_q may well be considerably larger than the target lower bound. Nevertheless, optimal query time can be achieved in an important special case described next.

Definition 8.1. We say that the M relation for a CFG $G = (V, E)$ is *forest structured* if its transitive reduction M_r is a forest, with edges directed from child to parent and with additional self-loops at some nodes.

PROPOSITION 8.2. *If M is forest structured then, for any $S \subseteq V$, the set $J(S)$ can be obtained in query time $T_q = O(|S| + |J(S)|)$.*

PROOF. To compute the set $J(S)$ of all nodes that are reachable from S by nontrivial M-paths, for each $w \in S$, we mark and output w if it has a self-loop; then we mark and output the interior nodes on the path in M_r from w to its highest ancestor that is not already marked.

In the visited sub-forest, each edge is traversed only once. The number of visited nodes is no smaller than the number of visited edges. A node v is visited if and only if it is a leaf of the sub-forest ($v \in S$), or an internal node of the sub-forest ($v \in J(S)$). Hence, $T_q = O(|S| + |J(S)|)$, as stated. □

For the interesting class of *structured programs* (defined in Section 8.1), we show (in Section 8.2) that the merge relation is indeed forest structured. Hence, by Proposition 8.2, $J(S)$ can be computed in optimal query time. In Section 8.3, we also show that M_r can be constructed optimally in preprocessing time $O(|V| + |E|)$.

8.1 Structured Programs

We begin with the following inductive definition of structured programs.

Definition 8.3. The CFG $G_0 = (\mathtt{START} = \mathtt{END}, \emptyset)$ is *structured*. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are structured CFGs, with $V_1 \cap V_2 = \emptyset$, then the following CGFs are also *structured*:

—The *series* $G_1G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{\mathtt{END}_1 \rightarrow \mathtt{START}_2\})$, with $\mathtt{START} = \mathtt{START}_1$ and $\mathtt{END} = \mathtt{END}_2$. We say that G_1G_2 is a *series* region.

—The *parallel* or *if-then-else* $G_1 \otimes G_2 = (V_1 \cup V_2 \cup \{\mathtt{START}, \mathtt{END}\}, E_1 \cup E_2 \cup \{\mathtt{START} \rightarrow \mathtt{START}_1, \mathtt{START} \rightarrow \mathtt{START}_2, \mathtt{END}_1 \rightarrow \mathtt{END}, \mathtt{END}_2 \rightarrow \mathtt{END}\})$. We say that $G_1 \otimes G_2$ is a *conditional* region.

—The *repeat-until* $G_1^* = (V_1 \cup \{\mathtt{START}, \mathtt{END}\}, E_1 \cup \{\mathtt{START} \rightarrow \mathtt{START}_1, \mathtt{END}_1 \rightarrow \mathtt{END}, \mathtt{END} \rightarrow \mathtt{START}\})$. We say that G_1^* is a *loop* region.

If $W \subseteq V$ is (the vertex set of) a series, loop, or a conditional region in a structured CFG $G = (V, E)$, we use the notation $\mathtt{START}(W)$ and $\mathtt{END}(W)$ for the entry and the exit points of W, respectively, we let $boundary(W) = \{\mathtt{START}(W), \mathtt{END}(W)\}$, $interior(W) = W - boundary(W)$, and write $W =< \mathtt{START}(W), \mathtt{END}(W) >$.

Abusing notation, we will use $W =< \mathtt{START}(W), \mathtt{END}(W) >$ to denote also the sub-graph of G induced by the vertex set W.

The following lemma lists a number of useful properties of dominance in a structured program. The proofs are simple exercises and hence are omitted.

LEMMA 8.4. *Let $W =< s, e >$ be a series, loop, or conditional region in a structured CFG. Then:*

1. *Node s dominates any $w \in W$.*
2. *Node e does not properly dominate any $w \in W$.*
3. *If w is dominated by s and not properly dominated by e, then $w \in W$.*
4. *A node $w \in W$ dominates e if and only if w does not belong to the interior of any conditional region $C \subseteq W$.*
5. *Any loop or conditional region U is either (i) disjoint from, (ii) equal to, (iii) subset of, or (iv) superset of W.*

8.2 The M Relation is Forest-structured

It is easy to see that, in a structured program, an up-edge is either a back-edge of a loop or an edge to the `END` of a conditional. The nodes whose EDF set contains a given up-edge are characterized next.

LEMMA 8.5. *Let $W =< s, e >$ be a region in a structured CFG $G = (V, E)$.*

1. *If W is a* loop *then $(e \to s) \in EDF(w)$ iff (i) $w \in W$ and (ii) w dominates e.*
2. *If $W =< s_1, e_1 > \otimes < s_1, e_2 >$ is a conditional then, for $i = 1, 2$, $(e_i \to e) \in EDF(w)$ iff $w \in < s_i, e_i >$ and w dominates e_i.*

PROOF. We give the proof only for *1* and omit the proof for *2*, which is similar.

($\Rightarrow$) By the assumption $(e \to s) \in EDF(w)$ and Definition 3.4 we have that (ii) w dominates e and (iii) w does not strictly dominate s. Thus, (ii) is immediately established. To establish (i), we show that (iv) e does not strictly dominate w, that (v) s dominates w, and then invoke part 3 of Lemma 8.4.

Indeed, (iv) follows from (ii) and the asymmetry of dominance.

Observe next that both s and w are dominators of e (from part 1 of Lemma 8.4 and (ii), respectively), hence one of them must dominate the other. In view of (iii), the only possibility remains (v).

($\Leftarrow$) By assumption, (ii) w dominates e. Also by assumption, $w \in W$ so that, by part 3 of Lemma 8.4, (v) s dominates w. By (v) and asymmetry of dominance, we have that (iii) w does not strictly dominate s. By (ii), (iii), and Definition 3.4, it follows that $(e \to s) \in EDF(w)$. $\square$

Lemma 8.5 indicates that $DF(w)$ can be determined by examining the loop and conditional regions C that contain w and checking whether w dominates an appropriate node. By part 4 of Lemma 8.4, this check amounts to determining whether w belongs to the interior of some conditional region $C \subseteq W$. Since the regions containing w are not disjoint, by part 5 of Lemma 8.4, they form a sequence ordered by inclusion. Thus, each region in a suitable prefix of this sequence contributes one node to $DF(w)$. To help formalizing these considerations, we introduce some notation.

Definition 8.6. Given a node w in a structured CFG, let $H_1(w) \subset H_2(w) \subset \ldots \subset H_{d(w)}(w)$ be the sequence of loop regions containing w and of conditional

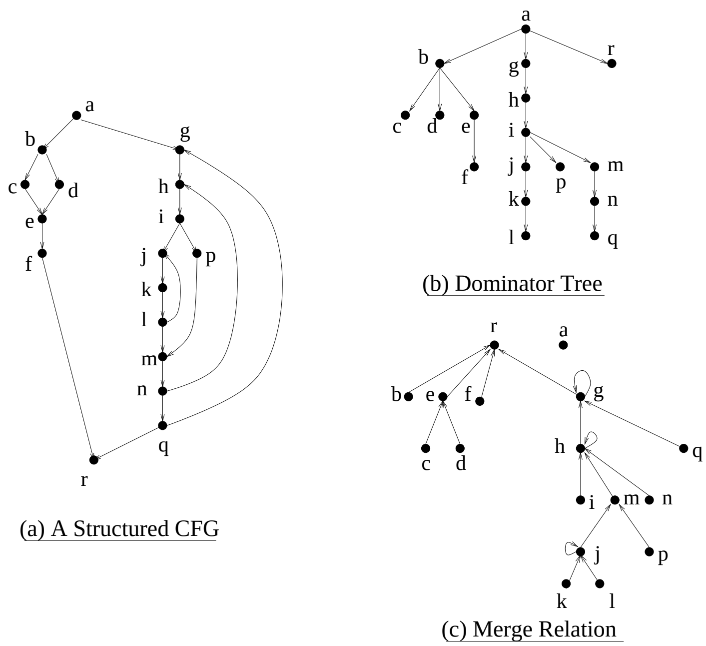


Fig. 21. A structured CFG and its M_r forest

regions containing w as an interior node. We also let $\ell(w)$ be the largest index ℓ for which $H_1(w), \ldots, H_{\ell(w)}(w)$ are all loop regions.

Figure 21(a) illustrates a structured CFG. The sequence of regions for node k is $H_1(k) =< j, l >$, $H_2(k) =< i, m >$, $H_3(k) =< h, n >$, $H_4(k) =< g, q >$, $H_5(k) =< a, r >$, with $d(w) = 5$, and $\ell(w) = 1$, since $H_2(w)$ is the first conditional region in the sequence. With the help of the dominator tree shown in Figure 21(b), one also sees that $DF(k) = \{j, m\} = \{\texttt{START}(H_1(k)), \texttt{END}(H_2(k))\}$. For node c, we have $H_1(c) =< b, e >$, $H_2(c) =< a, r >$, $d(c) = 2$, $\ell(c) = 0$, and $DF(c) = \{r\} = \{\texttt{END}(H_1(c))\}$.

Proposition 8.7. *For $w \in V$, if $\ell(w) < d(w)$, then we have:*

$$DF(w) = \{\texttt{START}(H_1(w)), \ldots, \texttt{START}(H_{\ell(w)}(w)), \texttt{END}(H_{\ell(w)+1}(w))\},$$

else ($\ell(w) = d(w)$, i.e.*, no conditional region contains w in its interior) we have:*

$$DF(w) = \{\texttt{START}(H_1(w)), \ldots, \texttt{START}(H_{\ell(w)}(w))\}.$$

Proof. $\ldots \subseteq DF(w)$. Consider a node $\texttt{START}(H_i(w))$ where $i \leq \ell(w)$. By definition, $w \in H_i(w)$ and there is no conditional region $C \subset H_i(w)$ that contains w as an internal node; by part 4 of Lemma 8.4, w dominates $\texttt{END}(H_i(w))$. By Lemma 8.5, $\texttt{START}(H_i(w)) \in DF(w)$. A similar argument establishes that $\texttt{END}(H_{\ell(w)+1}(w)) \in DF(w)$.

$DF(w) \subseteq \ldots$. Let $(u \to v) \in EDF(w)$. If $(u \to v)$ is the back-edge of a loop region $W =< v, u >$, Lemma 8.5 asserts that w dominates u and is contained in W. Since w dominates u, no conditional region $C \subseteq W$ contains w as an internal node. Therefore, $w \in \{\texttt{START}(H_1(w)), \ldots, \texttt{START}(H_{\ell(w)}(w))\}$. A similar argument if v is the $\texttt{END}$ node of a conditional region. $\square$

We can now establish that the M relation for structured programs is forest structured.

THEOREM 8.8. *The transitive reduction M_r of the M relation for a structured CFG $G = (V, E)$ is a forest, with an edge directed from child w to its parent, denoted $iM(w)$. Specifically, w is a root of the forest whenever $DF(w) - \{w\} = \emptyset$ and $iM(w) = \min(DF(w) - \{w\})$ otherwise. In addition, there is a self-loop at w if and only if w is the start node of a loop region.*

PROOF. *Forest structure.* From Proposition 8.7, the general case is

$$DF(w) = \{\texttt{START}(H_1(w)), \ldots, \texttt{START}(H_{\ell(w)}(w)), \texttt{END}(H_{\ell(w)+1}(w))\}.$$

Let x and y be distinct nodes in $DF(w)$. If $x = \texttt{START}(H_i(w))$ and $y = \texttt{START}(H_j(w))$, with $i < j \leq \ell$, then $H_i(w) \subset H_j(w)$ (see Definition 8.6). Furthermore, there is no conditional region C such that $H_i(w) \subset C \subset H_j(w)$, otherwise we would have $\ell(w) + 1 < j$ against the assumption. From Proposition 8.7, it follows that $y \in DF(x)$.

The required result can be argued similarly if $x = \texttt{START}(H_i(w))$ and $y = \texttt{END}(H_{\ell(w)+1}(w))$.

Self-loop property. If $w \in DF(w)$, there is a prime M-path $w \xrightarrow{*} u \rightarrow w$ on which every node other than w is strictly dominated by w. Therefore, the last edge $u \rightarrow w$ is an up-edge. With reference to Lemma 8.5 and its preamble, the fact that w dominates v rules out case 2 (w is the $\texttt{END}$ of a conditional). Therefore, $u \rightarrow w$ it is the back-edge of a loop, of which w is the $\texttt{START}$ node.

Conversely, suppose that w is the $\texttt{START}$ node of a loop $< w, e >$. Consider the path $P = w \xrightarrow{+} w$ obtained by appending back-edge $e \rightarrow w$ to any path $w \xrightarrow{+} e$ on which every node is contained in the loop. Since w strictly dominates all other nodes on P, P is a prime M-path, whence $w \in DF(w)$. $\square$

8.3 Computing M_r

The characterization developed in the previous section can be the basis of an efficient procedure for computing the M_r forest of a structured program. Such a procedure would be rather straightforward if the program were represented by its abstract syntax tree. However, for consistency with the framework of this paper, we present here a procedure BuildMForest based on the CFG representation and the associated dominator tree. This procedure exploits a property of dominator trees of structured programs stated next, omitting the simple proof.

LEMMA 8.9. *Let D be the dominator tree of a structured CFG where the children of each node in D are ordered left to right in ω-order. If node s has more than one child, then*

1. *s is the $\texttt{START}$ of a conditional region $< s, e > = < s_1, e_1 > \otimes < s_2, e_2 >$;*
2. *the children of s are s_1, s_2, and e, with e being the rightmost one;*
3. *e_1 and e_2 are leaves.*

Procedure BuildMForest(CFG G, DominatorTree D):returns M_r**;**

{
1: Assume CFG $= (V, E)$;
2: **for** $w \in V$ **do**
3: MSelfLoop[w] = FALSE;
4: iM[w] = NIL;
5: **od**
6: Stack = {};
7: **for** each $w \in V$ in ω-order **do**
8: **for** each v s.t. $(w \rightarrow v) \in E_{up}$ in reverse ω-order **do**
9: PushOnStack(v) **od**
10: **if** NonEmptyStack **then**
11: **if** TopOfStack $= w$ **then**
12: MSelfLoop[w] = TRUE;
13: DeleteTopOfStack;
14: **endif**
15: **if** NonEmptyStack **then**
16: iM[w] = TopOfStack;
17: **if** ($idom$(TopOfStack)=$idom(w)$)
18: DeleteTopOfStack;
19: **endif**
20: **od**
21: return M_r = (iM, MSelfLoop);
}

Fig. 22. Computing forest M_r for a structured program

Node	c	d	f	e	b	l	k	j	p	q	n	m	i	h	g	r	a
Stack at Line 10	e	e	r	r	r	j m	j m	j m	m	g r	h g r	h g r	h g r	h g r	g r	..	..

Fig. 23. Algorithm of Figure 22 operating on program of Figure 21

The algorithm in Figure 22 visits nodes in ω-order and maintains a stack. When visiting w, first the nodes in α-$DF(w)$ are pushed on the stack in reverse ω-order. Second, if the top of the stack is w itself, then it is removed from the stack. Third, if the top of the stack is now a sibling of w, it also gets removed. We show that, at Line 10 of the algorithm, the stack contains the nodes of $DF(w)$ in w-order from top to bottom. Therefore, examination of the top of the stack is sufficient to determine whether there is a self-loop at w in the M-graph and to find the parent of w in the forest M_r, if it exists. Figure 23 shows the contents of the stack at Line 10 of Figure 22 when it is processing the nodes of the program of Figure 21 in ω-order.

PROPOSITION 8.10. *Let $G = (V, E)$ be a structured CFG. Then, the parent $iM(w)$ of each node $w \in V$ in forest M_r and the presence of a self-loop at w can be computed in time $O(|E| + |V|)$ by the algorithm of Figure 22.*

PROOF. Let $w_1, w_2, \ldots w_{|V|}$ be the ω-ordered sequence in which nodes are visited by the loop beginning at Line 7. We establish the loop invariant I_n: *at Line 10 of* *the n-th loop iteration, the stack holds the nodes in $DF(w_n)$, in ω-order from top to bottom.* This ensures that self-loops and $iM(w)$ are computed correctly. The proof is by induction on n.

Base case: The stack is initially empty and Lines 8 and 9 will push the nodes of α-$DF(w_1)$, in reverse-ω-order. Since w_1 is a leaf of the dominator tree, by Theorem 3.9, $DF(w_1) = \alpha$-$DF(w_1)$, and I_1 is established.

Inductive step: We assume I_n and prove I_{n+1}. From the properties of post-order walks of trees, three cases are easily seen to exhaust all possible mutual positions of w_n and w_{n+1}.

1. w_{n+1} is the leftmost leaf of the subtree rooted at the first sibling r of w_n to the right of w_n.
   ¿From Lemma 8.9 applied to $parent(w_n)$, there is a region $< parent(w_n), e >=< w_n, e_1 > \otimes < s_2, e_2 >$. From Proposition 8.7, $DF(w_n) \subseteq \{w_n, e\}$. Nodes w_n and e will be popped off the stack by the time control reaches the bottom of the loop at the n-th iteration, leaving an empty stack at Line 7 of the $(n+1)$-st iteration. Then the nodes in α-$DF(w_{n+1})$ will be pushed on the stack in reverse-ω order. Since w_{n+1} is a leaf, $DF(w_{n+1}) = \alpha$-$DF(w_{n+1})$ and I_{n+1} holds.
2. w_n is the rightmost child of w_{n+1}, with w_{n+1} having other children.
   From Lemma 8.9, $< w_{n+1}, w_n >$ is a conditional region. Since every loop and conditional region that contains w_n also contains w_{n+1} and vice versa, it follows from Proposition 8.7 that $DF(w_{n+1}) = DF(w_n)$. Furthermore, the children of w_{n+1} cannot be in $DF(w_{n+1})$, so they cannot be in $DF(w_n)$ either. By assumption, at Line 10 of the n-th iteration, the stack contains $DF(w_n)$. We see that nothing is removed from the stack in Lines 10-19 during the n-th iteration because neither w_n nor the siblings of w_n are in $DF(w_n)$. Also, α-$DF(w_{n+1})$ is empty, as no up-edges emanate from the end of a conditional, so nothing is pushed on the stack at Line 9 of the $(n+1)$-st iteration, which then still contains $DF(w_n) = DF(w_{n+1})$. Thus, I_{n+1} holds.
3. w_n is the only child of w_{n+1}.
   By Theorem 3.9, $DF(w_{n+1}) = \alpha$-$DF(w_{n+1}) \cup (DF(w_n) - \{w_n\})$. At the n-th iteration, the stack contains $DF(w_n)$, from which Lines 10-14 will remove w_n from the stack, if it is there, and Lines 15-19 will not pop anything, since w_n has no siblings. At the $(n+1)$-st iteration, Lines 8-9 will push the nodes in α-$DF(w_{n+1})$ on the stack, which will then contain $DF(w_{n+1})$. It remains to show that the nodes on the stack are in ω-order.
   If α-$DF(w_{n+1})$ is empty, ω-ordering is a corollary of I_n. Otherwise, there are up-edges emanating from w_{n+1}. Since w_{n+1} is not a leaf, part 3 of Lemma 8.9 rules out case 2 of Lemma 8.5. Therefore, w_{n+1}must be the end node of a loop $< s, w_{n+1} >$ and α-$DF(w_{n+1}) = \{s\}$.
   From Lemma 8.4, any other region $W =< s', e >$ that contains w_{n+1} in the interior will properly include $< s, w_{n+1} >$, so that s' strictly dominates s (from Lemma 8.4, part 1.) If W is a loop region, then $s \in DF(w_n)$ occurs before s' in ω-order. If W is a conditional region, then since $e \in DF(w_n)$ is the rightmost child of s', s must occur before e in ω-order. In either case, s will correctly be above s' or e in the stack.

The complexity bound of $O(|E| + |V|)$ for the algorithm follows from the observation that each iteration of the loop in Lines 7-20 pushes the nodes in α-$DF(w)$ (which is charged to $O(|E|)$) and performs a constant amount of additional work (which is charged to $O(|V|)$). $\square$

The class of programs with forest-structured M contains the class of structured programs (by Theorem 8.8) and is contained in the class of reducible programs (by Proposition 3.12). Both containments turn out to be strict. For example, it can be shown that for any CFG whose dominator tree is a chain M_r is a forest even though such a program may not be structured, due to the presence of non-well-nested loops. One can also check that the CFG with edges $(s, a), (s, b,), (s, c), (s, d), (a, b), (b, d), (a, c), (a, d)$ is reducible but its M_r relation is not a forest.

If the M_r relation for a CFG G is a forest, then it can be shown easily that $iM(w) = \min DF(w)$, where the min is taken with respect to an ω-ordering of the nodes. Then, M_r can be constructed efficiently by a simple modification of the node-scan algorithm, where the DF sets are represented as balanced trees, thus enabling dictionary and merging operations in logarithmic time. The entire preprocessing then takes time $T_p = O(|E| \log |V|)$. Once the forest is available, queries can be handled optimally as in Proposition 8.2.

8.4 Applications to Control Dependence

In this section, we briefly and informally discuss how the M_r forest enables the efficient computation of set $DF(w)$ for a given w. This is equivalent to the well-known problem of answering *node control dependence* queries [PB97]. In fact, the node control dependence relation in a CFG G is the same as the dominance frontier relation in the reverse CFG G^R, obtained by reversing the direction of all arcs in G. Moreover, it is easy to see that G is structured if and only if G^R is structured.

By considering the characterization of $DF(w)$ provided by Proposition 8.7, it is not difficult to show that $DF(w)$ contains w if and only if M_r has a self-loop at w and, in addition, it contains all the proper ancestors of w in M_r up to and including the first one that happens to be the end node of a conditional region. Thus, a simple modification of the procedure in the proof of Proposition 8.2 will output $DF(w)$ in time $O(|DF(w)|)$.

One can also extend the method to compute set $EDF(w)$ or, equivalently (edge) *control dependence* sets, often called *cd* sets. The key observation is that each edge in M_r is "generated" by an up-edge in the CFG, which could be added to the data structure for M_r and output when traversing the relevant portion of the forest path starting at w.

Finally, observe that $DF(u) = DF(w)$ if and only if, in M_r, (i) u and w are siblings or are both roots and (ii) u and v have no self-loops. On this basis, one can obtain DF-equivalence classes which, in the reverse CFG, correspond to the so called *cdequiv* classes.

In summary, for control dependence computations on structured programs, an approach based on augmentations of the M_r data structure offers a viable alternative to the more general, but more complex approach using augmented postdominator trees, proposed in [PB97].

9. CONCLUSIONS

This paper is a contribution to the state of the art of ϕ-placement algorithms for converting programs to SSA form. Our presentation is based on a new relation on CFG nodes called the *merge* relation which we use to derive all known properties of the SSA form in a systematic way. Consideration of this framework led us to invent new algorithms for ϕ-placement which exploit these properties to achieve asymptotic running times that match those of the best algorithms in the literature. We presented both known and new algorithms for ϕ-placement in the context of this framework, and evaluated performance on the SPEC benchmarks.

Although these algorithms are fast in practice, they are not optimal when ϕ-placement has to be done for multiple variables. In the multiple variable problem, a more ambitious goal can be pursued. Specifically, after suitable preprocessing of the CFG, one can try to determine ϕ-placement for a variable in time $O(|S| + |J(S)|)$ (that is, proportional to the number of nodes where that variable generates a definition in the SSA form). We showed how this could be done for the special case of structured programs by discovering and exploiting the forest structure of the merge relation. The extension of this result to arbitrary programs remains a challenging open problem.

ACKNOWLEDGMENTS

Many of the results reported in this paper were derived by the authors over the course of several visits to the IBM T.J. Watson Research Center at Yorktown Heights. The authors would like to thank the IBM Corporation for maintaining a world-class research center.

REFERENCES

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

G. Bilardi and K. Pingali. A framework for generalized control dependence. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation.* ACM, New York, 291–300.

A. L. Buchsbaum, H. Kaplan, A. Rogers and J. R. Westbrook. Linear-Time Pointer-Machine Algorithms for Least Common Ancestors, MST Verification, and Dominators, *Symposium on the Theory of Computing*, 1998, page 279-88.

Ron Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 461–476, August 1993. Published as Lecture Notes in Computer Science, number 768.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1992.

Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Inc., 1981.

J. Peterson et al. Haskell: a purely functional language. http://www.haskell.org, April 1997.

Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, Albuquerque, New Mexico, June 23–25, 1993.

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

K. Pingali and G. Bilardi. $\mathcal{APT}$: A data structure for optimal control dependence computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 32–46, 1995.

K. Pingali and G. Bilardi. Optimal control dependence computation and the Roman Chariots problem. In *ACM Transactions on Programming Languages and Systems.* ACM, New York, pages 462-491, May 1997.

Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence Flow Graphs: An algebraic approach to program dependencies. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

Andy Podgurski and Lori Clarke. A formal model of program dependences and its implications for software testing, debugging and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

G. Ramalingam. On loops, dominators, and dominance frontiers In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), pages 233-241.*

H. Reif and R. Tarjan. Symbolic Program Analysis in Almost-linear Time. *Journal of Computing*, 11(1):81-93, February 1981.

Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, San Francisco, California, January 1995.

Vugranam C. Sreedhar and Guang R. Gao. Personal Communication. September 1996.

R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

M. Weiss. The transitive closure of control dependence: the iterated join. *ACM Letters on Programming Languages and Systems*, pages 178-190, Volume 1:2, June 1992.

M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.

A. APPENDIX

Definition A.1. A *control flow graph (CFG)* $G = (V, E)$ is a directed graph in which a node represents a statement and an edge $u \to v$ represents possible flow of control from u to v. Set V contains two distinguished nodes: `START`, with no predecessors and from which every node is reachable; and `END`, with no successors and reachable from every node.

Definition A.2. A *path* from x_0 to x_n in graph G is a sequence of edges of G of the form $x_0 \to x_1, x_1 \to x_2, \ldots, x_{n-1} \to x_n$. Such a path is said to be *simple* if nodes $x_0, x_1, \ldots, x_{n-1}$ are all distinct; if $x_n = x_0$ the path is also said to be a *simple cycle*. The length of a path is the number n of its edges. A path with no edges $(n = 0)$ is said to be *empty*. A path from x to y is denoted as $x \overset{*}{\to} y$ in general and as $x \overset{+}{\to} y$ if it is not empty. Two paths of the form $P_1 = x_0 \to x_1, \ldots, x_{n-1} \to x_n$ and $P_2 = x_n \to x_{n+1}, \ldots, x_{n+m-1} \to x_{n+m}$ (last vertex on P_1 equals first vertex on P_2) are said to be *concatenable* and the path $P = P_1 P_2 = x_0 \to x_1, x_1 \to x_2, \ldots, x_{n+m-1} \to x_{n+m}$ is referred to as their *concatenation*.

Definition A.3. A node w *dominates* a node v, denoted $(w, v) \in D$, if every path from `START` to v contains w. If, in addition, $w \neq v$, then w is said to *strictly dominate* v.

It can be shown that dominance is a transitive relation with a tree-structured transitive reduction called the *dominator tree*, $T = (V, D_r)$. The root of this tree is `START`. The parent of a node v (distinct from `START`) is called the *immediate dominator* of v and is denoted by $idom(v)$. We let $children(w) = \{v : idom(v) = w\}$ denote the set of children of node w in the dominator tree. The dominator tree can be constructed in $O(|E|\alpha(|E|))$ time by an algorithm due to Tarjan and Lengauer [LT79], or in $O(|E|)$ time by a more complicated algorithm due to Buchsbaum *et al.* [BKRW98]. The following lemma is useful in proving properties that rely on dominance.

LEMMA A.4. *Let $G = (V, E)$ be a CFG. If w dominates u, then there is a path from w to u on which every node is dominated by w.*

PROOF. Consider any acyclic path $P = \texttt{START} \overset{*}{\to} u$. Since w dominates u, P must contain w. Let $P_1 = w \overset{+}{\to} u$ be the suffix of path P that originates at node w.

Suppose there is a node n on path P_1 that is not dominated by w. We can write path P_1 as $w \overset{+}{\to} n \overset{+}{\to} u$; let P_2 be the suffix $n \overset{+}{\to} u$ of this path. Node w cannot occur on P_2 because P is acyclic.

Since n is not dominated by w, there is a path $Q = \texttt{START} \overset{+}{\to} n$ that does not contain w. The concatenation of Q with P_2 is a path from `START` to u not containing w, which contradicts the fact that w dominates u. □

A key data structure in optimizing compilers is the *def-use chain* [ASU86]. Briefly, a statement in a program is said to *define* a variable Z if it may write to Z, and it is said to *use* Z if it may read the value of Z before possibly writing to Z. By convention, the `START` node is assumed to be a definition of all variables. The *def-use graph* of a program is defined as follows.

Definition A.5. The *def-use graph* of a control flow graph $G = (V, E)$ for variable Z is a graph $DU = (V, F)$ with the same vertices as G and an edge (n_1, n_2) whenever n_1 is a definition of a Z, n_2 is a use of Z, and there is a path in G from n_1 to n_2 that does not contain a definition of Z other than n_1 or n_2. If $(n_1, n_2) \in F$, then definition n_1 is said to *reach* the use of Z at n_2.

In general, there may be several definitions of a variable that reach a use of that variable. Figure 1(a) shows the CFG of a program in which nodes `START`, A and C are definitions of Z. The use of Z in node F is reached by the definitions in nodes A and C.