The Static Single Assignment Form and its Computation

Gianfranco Bilardi
DEI, Università di Padova, 35131 Padova, Italy
EECS, University of Illinois, Chicago, IL 60607

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

November 26, 1998

Abstract

The Static Single Assignment (SSA) form of programs is a commonly used intermediate representation of programs in optimizing compilers. The key step in converting a program to SSA form is called ϕ-placement. Cytron et al used *dominance frontiers* to design a ϕ-placement algorithm whose running time grows quadratically with the size of the control flow graph of the program. Recent efforts by the compiler community have led to other algorithms whose asymptotic running time grows linearly with program size, although only one of these actually outperforms the quadratic time algorithm in practice. However, the relationship between different algorithms and their relative performance has remained unclear. This paper gives a systematic presentation of known properties of the SSA form, and gives a simple framework in which all existing SSA algorithms can be placed and compared. The utility of the framework is demonstrated by the fact that considerations of this framework led us to invent two new ϕ-placement algorithms described here which require asymptotically linear running time. Finally, we compare the performance of these algorithms on model problems and on the SPEC benchmarks.

[1]Gianfranco Bilardi (bilardi@art.dei.unipd.it) was supported in part by the Italian Ministry of University and Research and by the Italian National Research Council.

[2]Keshav Pingali (pingali@cs.cornell.edu) was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

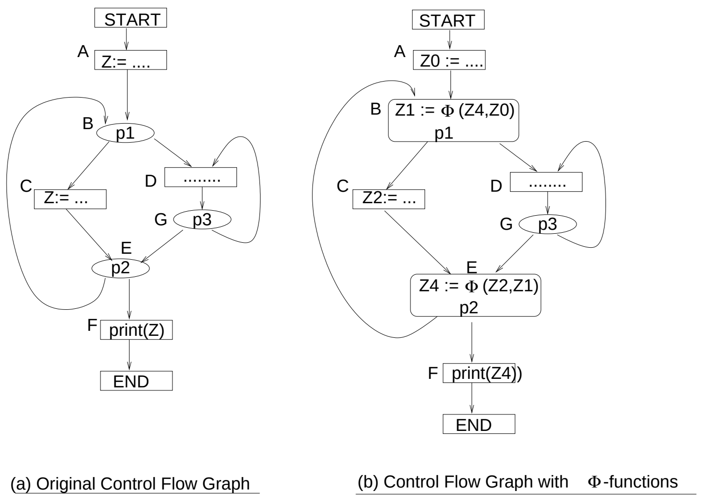


Figure 1: A Program and its SSA Form

1 Introduction

It has been observed that many compiler algorithms for optimizing programs become simpler and faster if programs are first transformed to *Static Single Assignment* (SSA) form [SS70, CFR+91]. In SSA form, each use[3] of a variable in the program is reached by a single definition of that variable. In general, the conversion of a program to SSA form must be accomplished by introducing *pseudo-assignments* at confluence points 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 "merges" multiple definitions that reach a confluence 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 each 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, and 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.

A program can be converted to SSA form very simply by placing ϕ-functions for all variables at every confluence point in the CFG, and performing the necessary renaming. In general, this approach introduces more ϕ-functions than necessary. For example, in Figure 1, this would introduce a ϕ-function at node D unnecessarily. In this paper, we study the problem of transforming an arbitrary program into an equivalent SSA form by inserting ϕ-functions only when they are needed. More specifically, given the CFG $G = (V, E)$ of a program and a set $S \subseteq V$ of nodes containing definitions of a given variable Z, we are interested in computing the set of nodes $\phi(S)$ where pseudo-assignments have to be placed to obtain an SSA form with respect to variable Z. By convention, the set S is assumed to contain `START`. The set $\phi(S)$ can be defined precisely as follows [CFR+91]:

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

Definition 1 *Given a CFG G=(V,E) and a set $S \subseteq V$ of its nodes such that* $\texttt{START} \in S$, *$\phi(S)$ is the set of all nodes v such that there is a pair of paths $u \xrightarrow{+} v$ and $w \xrightarrow{+} v$, with $u, w \in S$, intersecting only at v.*

If several variables have to processed, it may be efficient to preprocess the CFG and obtain a data structure that facilitates the construction of $\phi(S)$ for any given S. Therefore, the performance of an SSA conversion algorithm is appropriately measured by the *preprocessing time* T_p and *preprocessing space* S_p used to respectively build and store the data structure corresponding to G, and by the *query time* T_q used to obtain $\phi(S)$ from S, given the data structure.

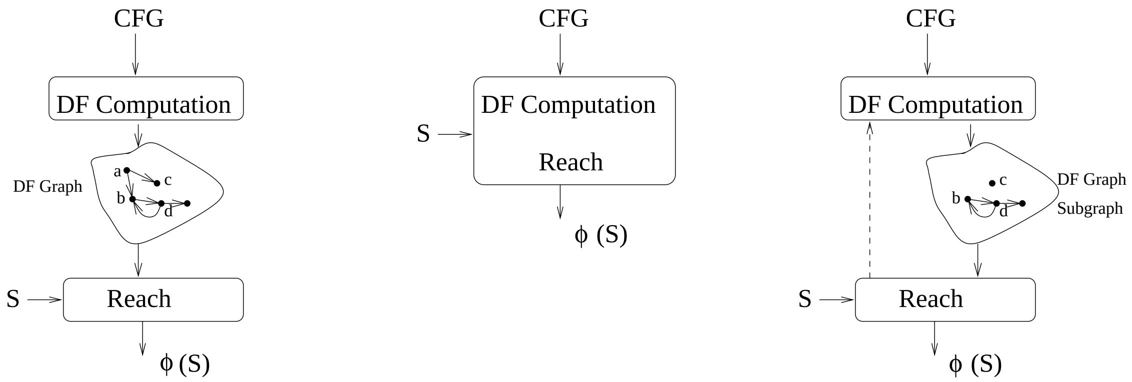
A number of algorithms for ϕ-placement have been proposed in the literature. An outline of an algorithm was given by Shapiro and Saint [SS70]. The first practical algorithm was described by Cytron et al. who introduced the key notion of *dominance frontier* for computing ϕ sets [CFR+91]. Since the size of dominance frontier sets can grow quadratically with program size, numerous attempts were made to improve this asymptotic bound. Cytron et al described an *on-the-fly* algorithm that could compute ϕ sets in $O(|E|\alpha(|E|))$ time, but this algorithm required path compression and other complications, and could not compete with the quadratic time algorithm in practice. Johnson and Pingali showed how the dependence flow graph [PBJ+91] can be used to compute the SSA form in $O(|E|)$ time per variable, but this approach was not competitive with the quadratic time algorithm either [JP93]. Sreedhar and Gao described another approach which traversed the dominator tree of the program to compute ϕ sets on demand [SG95]. This algorithm requires $O(|E|)$ preprocessing time, preprocessing space and query time, and it is easy to implement. Although they reported that this algorithm outperforms the quadratic algorithm in practice, more recent measurements taken by them show that the algorithm actually runs roughly 5 times slower than the quadratic algorithm on practical problems from the SPEC benchmarks [SG96]. The first algorithm with $O(|E|)$ preprocessing time, preprocessing space and query time that actually outperforms the quadratic algorithm was described by us in an earlier paper on optimal control dependence computation [PB95]. This algorithm uses a data structure called the *augmented dominator tree* $\mathcal{ADT}$ which requires a parameter β that controls a particular space-time trade-off. It subsumes the quadratic algorithm of Cytron et al. and the Sreedhar and Gao algorithm since these are special cases of the $\mathcal{ADT}$ algorithm, obtained for particular values of the parameter β.

The need for ϕ-set computation arises not only in SSA computations but in other contexts such as the computation of the *weak control dependence* relation [PC90] that extends the standard notion of control dependence to include non terminating program executions. In this context, the standard notion of postdominance must be replaced with the notion of loop postdominance [BP96]. The transitive reduction of loop postdominance is a forest which can be obtained from the postdominator tree by disconnecting a set K of nodes from their parents. It turns out that the determination of this set is equivalent to the computation of the set $\phi(C)$ in the reverse CFG, where C is a set of nodes called *crowns*(see [BP96] for details).

This paper gives a systematic presentation of known properties of the SSA form, and gives a simple framework in which all existing SSA algorithms can be placed and compared. As an indication of the utility of this framework, we mention that considerations of the framework led us to invent two new algorithms described here which require $O(|E|)$ preprocessing space, preprocessing time and query time.

In Section 2, we introduce the *join* relation between nodes of the CFG. Informally, $(w, v) \in J$, which we also write as $v \in J(w)$, if $v \in \phi(\{\texttt{START}, w\})$; that is, v is a ϕ-node for a variable assigned only at w and $\texttt{START}$. Three key properties make J the cornerstone of SSA computation:

1. If $\{\texttt{START}\} \subseteq S \subseteq V$, then $\phi(S) = \cup_{w \in S} J(w)$.



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

Figure 2: Three Strategies for Computing ϕ-placement

2. $v \in J(w)$ if and only if there is a CFG path from w to v that does not contain any dominator of v (i.e., any node whose removal would disconnect v from `START`.)
3. J is a transitive relation.

Property 1 reduces the computation of ϕ to that of J. Property 2 gives a handle for efficient computation of J by linking the join relation to the extensively studied *dominance* relation. Finally, Property 3 opens the door to efficient preprocessing techniques based on (partial) transitive reductions of relation J. In fact, let us consider a partial transitive reduction R of J. Then, the set $J(S) = \cup_{x \in S} J(x)$ equals the set $R^+(S) = \cup_{x \in S} R^+(x)$ of nodes reachable from some $x \in S$ in graph $G_R = (V, R)$. An 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 $R = J_r$, the (total) transitive reduction of J. However, the preprocessing time T_p to obtain R from the CFG $G = (V, E)$ is not necessarily minimized by the choice $R = J_r$.

In Section 3, we define the *dominance frontier relation* DF among nodes [CFR+91], and show it is a partial reduction of J (that is, $DF^+ = J$). The problem of ϕ-placement has a simple interpretation in the context of the DF graph: *the set $\phi(S)$ is simply the set of nodes in the DF graph that are reachable along non-trivial paths originating at a node in S.* Therefore, the two key steps in computing $\phi(S)$ are: (i) compute the DF relation and (ii) perform a reachability computation in the DF graph from the nodes in S. These two steps may be organized in different ways to produce different algorithms as shown in Figure 2.

1. *Two-phase algorithms*: The entire DF graph is constructed, and then the nodes reachable from input set S are determined. In general, this is like computing $f(g(x))$, where f and g are functions from data structures to data structures, by computing $g(x)$ first and passing the intermediate data structure to f.

   The main virtue of two-phase algorithms is simplicity. In Section 4, we describe two such algorithms, one of which is due to Cytron et al [CFR+91]. The preprocessing space and time requirements of these algorithms are identical: they require preprocessing time $T_p = O(|V| + |E| + |DF|)$ and preprocessing space $S_p = O(|V| + DF|))$. To compute a set $\phi(S)$, they visit the portion of G_{DF} reachable from S in time $T_q = O(|V| + |DF|)$.

2. *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 very sparse ($|E| = O(|V|)$), structured CFGs) [CFR$^+$91]. A lock-step algorithm performs the reachability computation incrementally as the DF relation is computed, so it is not necessary to store the entire DF relation. The idea of lock-step algorithms is quite general and involves computing $f(g(x))$ by applying the composition of f and g to x. Scalarization after loop jamming [Wolfe95] and frontal algorithms for out-of-core sparse matrix factorizations [GL81] exploit similar ideas.

   In Section 5, we discuss two lock-step algorithms in which the reachability computation is performed in lock-step with the DF computation. Both algorithms exploit the fact that the set S is known and avoid computing the entire DF relation, obtaining linear complexity in all measures: $T_p, S_p, T_q = O(|E|)$.

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

   Both issues are addressed in Section 6, with the introduction of the *augmented dominator tree* ($\mathcal{ADT}$), 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]. 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, we present experimental results that compare these algorithms for a micro-benchmark commonly used in this area, and for the SPEC benchmarks.

In summary, the contributions of this paper are as follows.

1. We give a systematic presentation of all known properties of the SSA form.
2. We place all existing algorithms for ϕ-placement into a single framework.
3. We present two new $O(|E|)$ algorithms that emerged from considerations of this framework.

2 The Join Relation and its Use in Computing SSA Form

In this section, we reduce the problem of ϕ-placement to the problem of computing a binary relation on nodes called the *join* relation. We present an algorithm for computing this relation and show how this provides a simple but inefficient solution to the ϕ-placement problem. We conclude the section by showing that the join relation is transitive but that it is difficult to compute its transitive reduction efficiently. This motivates the introduction of the DF relation, a partial transitive reduction of the join relation, in Section 3.

2.1 The Join Relation

Definition 2 *The* **join** *relation is a binary relation on nodes defined as follows:*
$J = \{(w, v) | w \in V, v \in \phi(\{\texttt{START}, w\})\}$.

For any node w, we let $J(w)$ denote the set $\{v|(w,v) \in J\}$, also called the join set *of w. Similarly, $J^{-1}(v)$ denotes the set $\{w|(w,v) \in J\}$.*

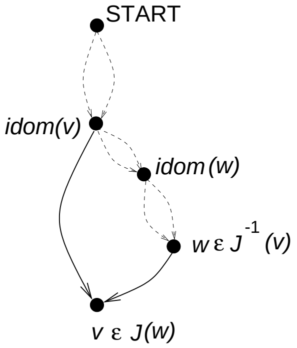


Figure 3: A pictorial representation of Lemma 1

Intuitively, the nodes in $J(w)$ are the nodes where ϕ-functions must be placed if the only assignments to the variable are at `START` and w, while a ϕ-function is needed at v if the variable is assigned in any of the nodes in the set $J^{-1}(v)$. We observe that $J(\texttt{START}) = \{\}$, trivially. The following result shows that, for a set of nodes S containing `START`, $\phi(S)$ is the union of the join sets of the elements of S.

Theorem 1 *Let $G = (V, E)$ and $\{\texttt{START}\} \subseteq S \subseteq V$. Then, $\phi(S) = \cup_{w \in S} J(w)$.*

Proof: It is straightforward to see from the definitions of ϕ and J that $\cup_{w \in S} J(w) \subseteq \phi(S)$.

To show that $\phi(S) \subseteq \cup_{w \in S} J(w)$, consider a node $v \in \phi(S)$. By Definition 1, there are paths $a \overset{+}{\to} v$ and $b \overset{+}{\to} v$, with $a, b \in S$, intersecting only at v. By Definition 12, there is also a path $\texttt{START} \overset{+}{\to} v$. We consider two cases:

1. Path $\texttt{START} \overset{+}{\to} v$ intersects path $a \overset{+}{\to} v$ only at v. Then, $v \in J(a)$, hence $v \in \cup_{w \in S} J(w)$.
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 $v \in J(b)$, hence $v \in \cup_{w \in S} J(w)$.

$\square$

There is a deep connection between join sets and the standard notion of *dominance* (defined in the appendix). The following lemma establishes one such connection that is useful for proving many facts about join sets. A pictorial representation of this lemma is shown in Figure 3.

Lemma 1 *Suppose that $v, w \in V$ and that there exists a path $P = w \overset{+}{\to} v$ that does not contain $idom(v)$. Then, (i) $idom(v)$ strictly dominates all nodes on P, and (ii) no strict dominator of v occurs on P.*

Proof: (i) (by contradiction) Otherwise, there is some node n on P that is not strictly dominated by $idom(v)$. This means 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 $idom(v)$, which is a contradiction.

(ii) By assumption, $idom(v)$ does not occur on P. Consider any node d distinct from $idom(v)$ that strictly dominates v. Since dominance is tree-structured, d must strictly dominate $idom(v)$. Therefore, there is a path $R = \texttt{START} \overset{+}{\to} d$ that does not contain $idom(v)$. If d were to occur on P, the path obtained

```
Procedure Join(CFG);
{
    Assume CFG = (V, E);
    J = {};
    for v ∈ V do
        Let G' = (G − idom(v))^R;
        Traverse G' from v, appending (w, v) to J for each visited w.
    od
    return J;
}
Procedure φ-placement(J, S);
{
    φ = {};
    for each v ∈ S
        for each (v, w) ∈ J append w to φ;
    return φ;
}
```

Figure 4: Reachability Algorithm for SSA

by concatenating R with the suffix of path P starting at d would give a path from `START` to v that does not contain $idom(v)$, which is a contradiction. Therefore, d cannot occur on P. □

The following theorem characterizes join sets in terms of the dominance relation.

Theorem 2 *For any $w \in V$, $J(w)$ is the set of nodes v such that there exists a path $w \overset{+}{\rightarrow} v$ that does not contain $idom(v)$.*

Proof: See Appendix. □

For example, consider the path $P = e \rightarrow b \rightarrow d \rightarrow f$ in Figure 5. This path does not contain $idom(f) = a$. As required by the theorem, there are paths $P_1 = \texttt{START} \rightarrow a \rightarrow b \rightarrow d \rightarrow f$ and $P_2 = e \rightarrow f$ with only f in common, i.e., $f \in J(e)$.

Theorem 2 provides a simple (if inefficient) method for computing the J relation. For any node y, the set $J^{-1}(y)$ is simply the set of nodes reachable from y in the graph obtained by deleting $idom(y)$ from the control flow graph and reversing all edges in the remaining graph (we call this graph $(G - idom(y))^R$. This observation is formalized in the algorithm shown in Figure 4 for constructing J from a CFG $G = (V, E)$.

Proposition 1 *The reachability algorithm for SSA has preprocessing time $T_p = O(|V||E|)$, preprocessing space $S_p = O(|V| + |J|) \leq O(|V|^2)$, and query time $T_q = O(\sum_{v \in S} |J(v)|)$.*

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 $|J|$ arcs to represent the J relation. The bound on query time comes from the fact that each $J(v)$ for $v \in S$ is obtained in time proportional to its own size. In this calculation, we have ignored the time to construct the dominator tree since this can be done in $O(|E|)$ time as explained in the appendix. □

The asymptotic complexity of the algorithm is not as good as that of other algorithms discussed in this paper. In Section 5, we discuss a more efficient version of this algorithm called the *mark pulling* algorithm.

2.2 J is Transitive

In general, the J relation of a CFG can be quite large, so it is natural to see if we can exploit the structure of J to avoid computing the entire relation. We now show that the J relation is transitive. Notice that, if R is a transitively reduced relation such that $J = R^+$, the set of nodes reachable from any node by paths with at least one edge is the same in the two graphs $G_R = (V, R)$ and $G_J = (V, J)$. Since $|R|$ can be considerably smaller than $|J|$, using G_R instead of G_J 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 $G_R(S) = (S \cup J(S), R_S)$ containing all the nodes and arcs reachable from S in G_R. Therefore, query time is $T_q = O(|S| + |J(S)| + |R_S|)$.

Theorem 3 *If $(x, y) \in J$ and $(y, z) \in J$, then $(x, z) \in J$.*

Proof: From Theorem 2, there is a path $P = x \stackrel{+}{\to} y$ that does not contain $idom(y)$, and a path $Q = y \stackrel{+}{\to} z$ that does not contain $idom(z)$. We will show that $idom(z)$ cannot occur in P, so concatenating P and Q gives a path from x to z that does not contain $idom(z)$; from Theorem 2, it follows that $(x, z) \in J$.

We note first that $idom(z)$ is distinct from y since it does not occur on path Q. Furthermore, from Lemma 1(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 1(ii) that $idom(z)$ does not occur on P. □

The problem of getting a relation R such that $R^+ = J$ for a given transitive relation J is a well known problem for which a number of algorithms have been developed [CLR92]. Usually, an R of minimum size, called the *transitive reduction* of J is the goal. Unless the relation J is acyclic (*i.e.*, the graph G_J is a dag), R is not necessarily unique. However, if the strongly connected components of J are collapsed into single vertices, the resulting acyclic condensate (call it J_c) has a unique transitive reduction J_r which can be computed in time $O(|V||J_c|)$ [CLR92] or $O(|V|^\gamma)$ by using an $O(n^\gamma)$ matrix multiplication algorithm[4]. The preceding discussion can be summarized by the following result.

Proposition 2 *The reachability algorithm for ϕ-placement (with transitive reduction preprocessing) has preprocessing time $T_p = O(|V|(|E| + \min(|J|, |V|^{\gamma-1}))$, preprocessing space $S_p = O(|V| + |J_r|)$, and query time $T_q = O(|V| + |J_r^+(S)|)$.*

Clearly, preprocessing time is too high for this algorithm to be of much practical interest. One natural question is whether the join relation J has any special structure that could facilitate the transitive reduction computation. Unfortunately, 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 $G = (V, R \cup (\{\texttt{START}\} \times (V - \texttt{START})))$ has exactly R^+ as its own join relation J. In particular, if R is transitive to start with, then $J = R$.

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

3 The Dominance Frontier Relation

We now introduce the key notion of the dominance frontier relation and establish its connection to the join relation.

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

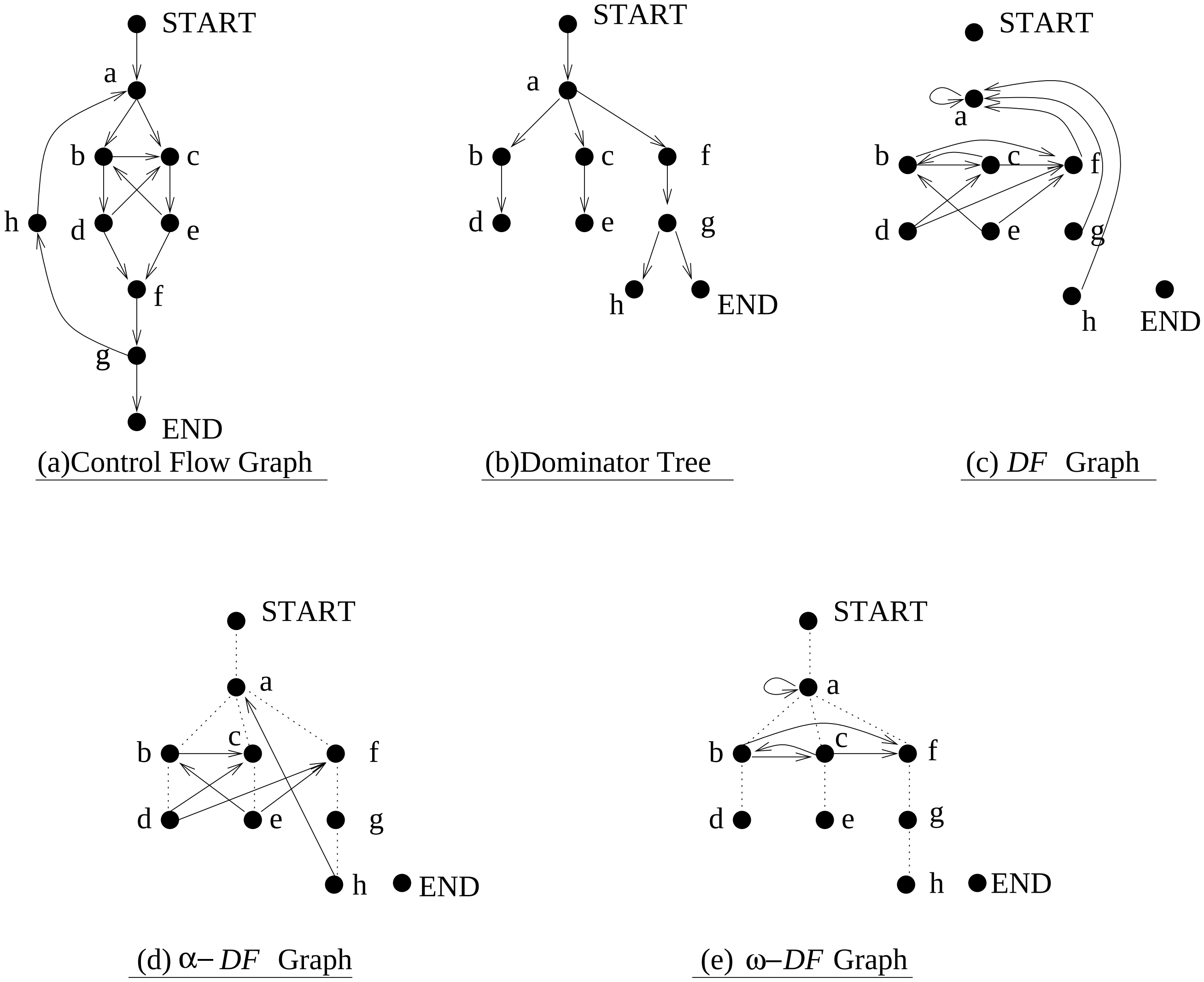


Figure 5: A Control Flow Graph and its DF Graphs

3.1 Definition of Dominance Frontier

The edges of the control flow graph $G = (V, E)$ can be partitioned 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 5(a,b) show a CFG and its dominator tree. In Figure 5(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 3 *Given a CFG $G = (V, E)$, $(u \to v) \in E$ is an up-edge if $u \neq idom(v)$. The subgraph of G containing only the up-edges in G is called the α-DFgraph $A = (V, E_{up})$.*

Figure 5(d) shows the α-DFgraph for the CFG of Figure 5(a). Since an up-edge $(u \to v)$ is a path from u to v that does not contain $idom(v)$, its existence implies $v \in J(u)$ (from Theorem 2); then, from the transitivity of J, $E_{up}^{+} \subseteq J$. In general, the latter relation does not hold with equality (for example, in Figure 5, $a \in J(g)$ but a is not reachable from g in the α-DFgraph). To obtain a relation whose transitive closure is J, we define the *dominance frontier* relation DF [CFR+91].

Definition 4 *An edge $(u \to v)$ is said to be in the* edge dominance frontier *of a node w (written as $EDF(w)$) 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 *of w (written as $DF(w)$).*

In other words, if $v \in DF(w)$, then w dominates a CFG predecessor of v but does not strictly dominate v itself. We will let DF stand for the binary relation corresponding to the dominance frontier, and write $DF(w)$ to denote the set of nodes v for which $(w, v) \in DF$. Since DF is a binary relation on nodes, it can be interpreted in the usual way as a graph. The DF graph for the running example is shown in Figure 5(c).

In general, DF is neither transitively closed nor transitively reduced, as can be seen in Figure 5. The presence of $c \to f$ and $f \to a$ and the absence of $c \to a$ in the DF graph shows that it is not transitively closed. The presence of edges $d \to c$, $c \to f$, and $d \to f$ in the graph of the DF relation shows that this relation is not transitively reduced in general.

3.2 The Transitive Closure of DF is J

We show next that the transitive closure of DF is the J relation.

Theorem 4 *$DF^{+} = J$.*

Proof: (a) $DF^{+} \subseteq J$. Given the transitivity of J (Theorem 3), it suffices to show that $DF \subseteq J$. Let $(w, v) \in DF$; then, by Definition 4, there is an edge $u \to v$ such that w dominates u and does not strictly dominate v. Let $P = w \xrightarrow{*} u$ be the suffix starting at w of a simple path $\texttt{START} \xrightarrow{*} u$. All nodes in P are dominated by w (if some x on P was not dominated by w, a path $\texttt{START} \xrightarrow{*} x \xrightarrow{*} u$ could be constructed reaching u without containing its dominator w). Since w does not strictly dominate v, it does not dominate $idom(v)$, hence $idom(v)$ is not on P. The concatenation of P with the edge $u \to v$ is then a path $w \xrightarrow{+} v$ not containing $idom(v)$, whence, from Theorem 2, $v \in J(w)$, i.e., $(w, v) \in J$.

(b) $J \subseteq DF^{+}$. Let $(w, v) \in J$; then, by Theorem 2, there is a path $P = w \xrightarrow{+} v$ that does not contain $idom(v)$. We then establish that $(w, v) \in DF^{+}$ by an inductive argument on the number n of up-edges on P ($n \geq 1$ since $idom(v) \notin P$).

Base case: $n = 1$. Since $idom(v) \notin P$, P is therefore of the form $w \xrightarrow{*} u \rightarrow v$ where $u \rightarrow v$ is the up-edge, and w dominates all nodes on the path $w \xrightarrow{*} u$. Since P does not contain $idom(v)$, no strict dominator of v can occur on P (Lemma 1(ii)); therefore, w does not strictly dominate v. From Definition 4, it follows that $(w, v) \in DF$; therefore, $(w, v) \in DF^+$.

Inductive hypothesis: If P has fewer than n up-edges, then $(w, v) \in DF^+$.

Suppose P has n up-edges. Let $(p \rightarrow q)$ be the first up-edge on P, where q is distinct from v. We see that $(w, q) \in DF$. Moreover, the path $Q = q \xrightarrow{+} u \rightarrow v$, which is a suffix of path P, does not contain $idom(v)$ and has less than n up-edges; by the inductive hypothesis, $(q, v) \in DF^+$. Therefore, $(w, v) \in DF^+$. □

Putting Theorems 1 and 4 together, we obtain a simple graph-theoretic interpretation of the ϕ relation: *the set $\phi(S)$ for a given set S of nodes is the set of nodes reachable in the DF graph by non-empty paths originating at some node in S.* It follows trivially that $\phi(S \cup \phi(S)) = \phi(S)$. A similar result was proved by Weiss [Weiss92].

3.3 Two Identities for the DF Relation

We now discuss two identities relating to the DF relation. Intuitively, the first expression permits efficient computation of the DF^{-1} relation which describes the set of nodes that have a given node v in their dominance frontiers, while the second expression gives a recursive definition of the DF set of a node w in terms of the DF sets of its children in the dominator tree. These two expressions are key to the rest of the development in the paper.

Definition 5 *Let $T =< V, F >$ be a tree. For $x, y \in V$, let $[x, y]$ denote the set of vertices on the simple path joining 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 5, $[d, a] = \{d, b, a\}$, $[d, a) = \{d, b\}$, and $[d, g] = \{d, b, a, f, g\}$.

Theorem 5 $EDF = \bigcup_{(u \rightarrow v) \in E} \ [u, idom(v)) \quad \times \quad u \rightarrow v.$

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 1 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 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 1, 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. □

Theorem 5 can be used to compute all the nodes w whose dominance frontiers contain a given node v (intuitively, $DF^{-1}(v)$): compute the union of the sets $[u, idom(v))$ for all incoming edges $(u \rightarrow v)$. However, it is not obvious how to use this result to compute efficiently the nodes that are in the dominance frontier of a given node w ($DF(w)$). For this, a recursive characterization of the DF relation that computes the dominance frontier of a node in terms of the dominance frontiers of its children in the dominator tree is useful [CFR+91].

Theorem 6 *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 5. 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 6, 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 rhs expression. Applying, Definition 4, we see that there must be an edge $(u \to v)$ 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 $v \in \alpha(w)$, there is a CFG edge $(w \to v)$ such that w does not strictly dominate v. Applying Definition 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 $(u \to v)$ 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 1. Therefore, w cannot strictly dominate v.
     This means that $v \in DF(w)$ as required.

□

3.4 Strongly Connected Components of the DF and J Graphs

There is an immediate and important consequence of Theorem 5 which is useful in proving many results about the DF and J relations. The *level* of a node in the dominator tree can be defined in the usual way: the root has a level of 0, and the level of any other node is 1 more than the level of its parent. From Theorem 5, 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 J) edges are oriented in a special way with respect to the dominator tree: a DF or J edge overlayed on the dominator tree is always directed "upwards" or "sideways" in this tree, as can be seen in Figure 5. Furthermore, if $(w, v) \in DF$, then $idom(v)$ dominates w (this is a special case of Lemma 1). For future reference, we state these facts explicitly.

Lemma 2 *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.*

Proof: Immediate from Theorem 5. □

This result can be used to prove 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 J relation, this means for any node v, $(x, v) \in J$ iff $(y, v) \in J$, and $(v, x) \in J$ iff $(v, y) \in J$. Therefore, nodes in a scc of the DF graph are $J-equivalent$. The following lemma states that J-equivalent nodes have a special structure with respect to the dominator tree.

Lemma 3 *Given a CFG = (V, E) and its dominator tree D, all nodes in a strongly connected component of the DF relation (equivalently, the J relation) of this graph are siblings in D.*

> **Proof:** Consider any cycle $n_1 \to n_2 \to n_3 .... \to n_1$ in the scc. From Lemma 2, 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 2, it also follows that n_1, n_2, etc. must be siblings in D. □

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.

3.5 Size of DF Relation

How large is DF? Since $DF \subseteq V \times V$, clearly $|DF| \leq |V|^2$. From Theorem 5, 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 15 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 in a preprocessing phase before doing reachability computations to answer queries.

4.1 Edge scan algorithm

The edge scan algorithm is essentially a direct translation of the expression for DF given by Theorem 5. The implementation is shown in Figure 6, and analyzed next.

Proposition 3 *The edge scan algorithm for SSA in Figure 6 has preprocessing time $T_p = O(|V| + |DF|)$, preprocessing space $S_p = O(|V| + |DF|)$, and query time $T_q = O(\sum_{v \in (S \cup J(S))} |DF(v)|)$.*

> **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 $T_p = O(|V| + |E| + |DF|)$. 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 Reach of Figure 6. Query time is proportional to the size of the portion of (V, DF) reachable from S. □

```
Procedure EdgeScanDF(CFG, DominatorTree D);
{
    Assume CFG = (V, E);
    DF = {};
    for each edge e = (u → v) ∈ E do
        if u ≠ idom(v) then
            w = u;
            while w ≠ idom(v) do
                Append edge w → v to DF;
                w = idom(w)
            od
        endif
    od
} Procedure NodeScanDF(CFG,DominatorTree D);
{
    Assume CFG = (V, E);
    Initialize DF(w) = {} for all nodes w;
    for each CFG edge (u → v) do
        if (u ≠ idom(v)) DF(u) = DF(u) ∪ {v}
    od
    for each node w ∈ D in bottom-up order do
        DF(w) = DF(w) ∪ (∪_{c∈children(w)} DF(c) − children(w));
    od
} Procedure φ-placement(G,S);
{
    In G, mark all nodes in set S;
    J(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
            J(S) = J(S) ∪ {v};
            if v is not marked then
                Mark v;
                Enter v into worklist M;
            endif
        od
    od
    Output J(S);
}
```

Figure 6: Edge Scan and Node Scan Algorithms

4.2 Node scan algorithm

The node scan algorithm 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 6. Pseudocode for the node scan algorithm is given in Figure 6. The DF sets can be represented in many ways; one simple approach is to represent them as linked lists of nodes sorted by node number, in which case, union and difference operations can be done in time proportional to the size of the operands of these operations in the usual way.

Proposition 4 *The node scan algorithm for SSA in Figure 6 has preprocessing time $T_p = O(|V| + |DF|)$, preprocessing space $S_p = O(|V| + |DF|)$, and query time $T_q = O(\sum_{v \in (S \cup J(S))} |DF(v)|)$.*

> **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. $\square$

4.3 Discussion

The node scan algorithm in Figure 6 is similar to the one given by Cytron et al [CFR+91]. As we can see from Propositions 3 and 4, 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, for programs with shallow dominator trees, the size of the DF graph may be comparable with the size of the CFG, so two-phase algorithms may be quite efficient. This is usually the case in most real programs, as we discuss at length in Section 7.

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 of the DF graph that is required to answer the $\phi(S)$ query. Intuitively, both are marker propagation algorithms that generate the graph concurrently with marker propagation to determine reachability from an initial set of marked nodes.

Reachability in a graph can be computed efficiently if the graph is acyclic since nodes can be processed in topological order. The initial set of nodes is marked, and at each step of the algorithm, a *mark pulling* algorithm would mark the current node if any of its predecessors in the graph were marked, while a *mark pushing* algorithm would mark all successors in the graph if the current node was marked. Processing nodes in topological order removes the need to examine a node more than once. However, the DF graph is not acyclic in general as can be seen in the running example of Figure 5 (there is a non-trivial cycle between nodes b and c in the DF graph). We show in this section that it is possible to exploit Lemma 3 to compute a topological sort of the acyclic condensate of the DF graph in $O(|E|)$ time. Then, we discuss a mark pulling algorithm and a mark pushing algorithm that process nodes in this order.

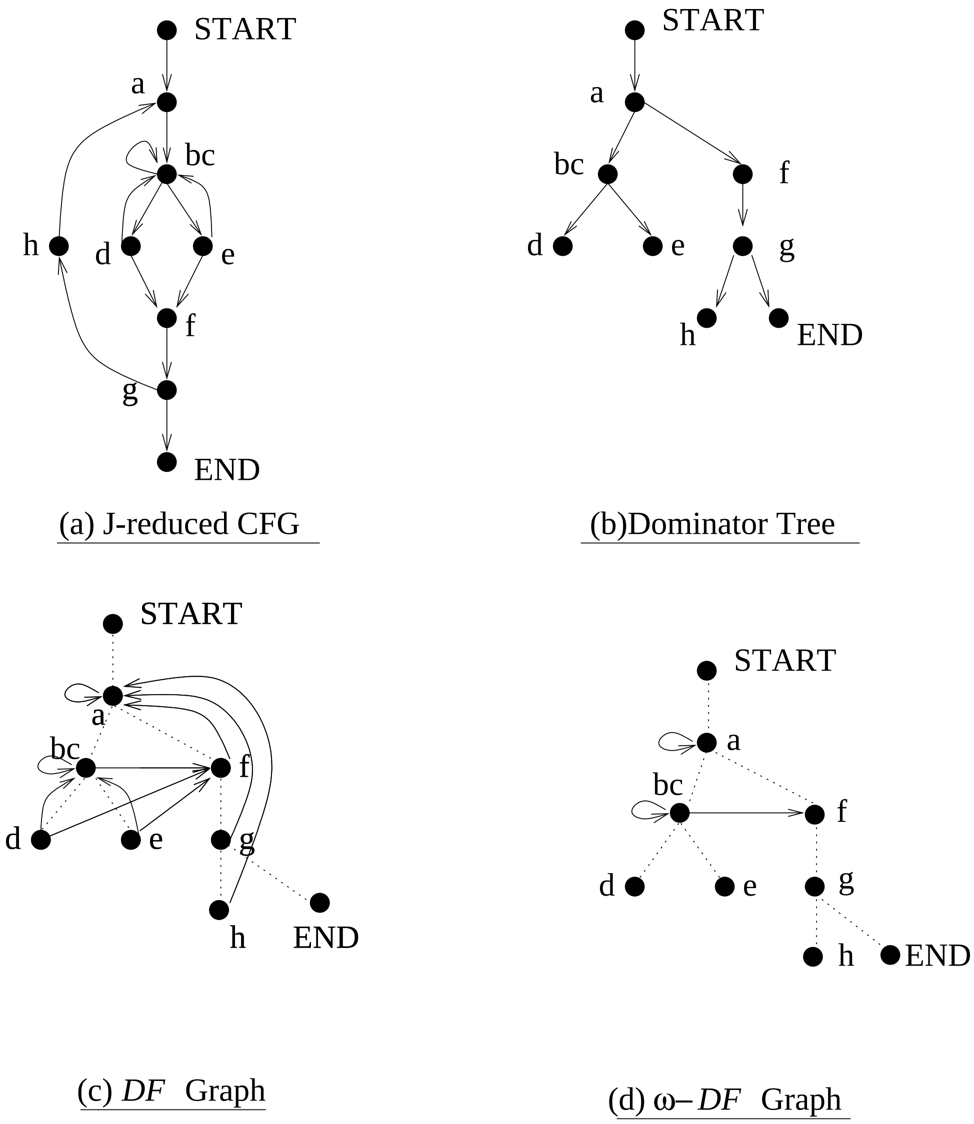


Figure 7: J-reduced CFG Corresponding to CFG of Figure 5

5.1 Topological Sorting of the DF and J Graphs

It is convenient for the presentation to define a derived CFG called the J-reduced CFG that is obtained from a CFG G by collapsing nodes that are part of the same scc in the J graph of CFG G. Figure 7 shows the J-reduced CFG corresponding to the CFG of Figure 5. The only non-trivial scc in the J graph (equivalently, in the DF graph) of the CFG in Figure 5 contains nodes b and c, and these are collapsed into a single node named bc in the J-reduced graph. The dominator tree for the J-reduced graph can be obtained by collapsing these nodes in the dominator tree of the original CFG. *Our algorithms do not build the J-reduced CFG or its dominator tree explicitly, but compute a topological ordering of its DF-graph directly.*

Definition 6 *Given a CFG $G = (V, E)$, the corresponding* J-reduced CFG *is the graph* $\tilde{G} = (\tilde{V}, \tilde{E})$

where $\tilde{V}$ is the set of strongly connected components of J, 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 $v \in b$.

Without loss of generality, the ϕ-placement problem can be solved on the reduced CFG. In fact, if $\tilde{J}$ denotes the join relation in $\tilde{G}$, and $\tilde{v} \in \tilde{V}$ denotes the component to which v belongs, then $J(v) = \cup_{\tilde{x} \in \tilde{J}(\tilde{v})} \tilde{x}$. The key observation permitting the efficient computation of scc's in the DF graph is Lemma 3 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 restrict attention to a subset of the DF graph called the *ω-DF graph* that is defined next.

Definition 7 *The ω-DF relation of a CFG is a subrelation 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 5(e) shows the ω-DF graph for the running example. Figure 8 shows an algorithm for computing this graph.

Lemma 4 *The ω-DF graph for CFG $G = (V, E)$ is constructed in $O(|E|)$ time by the algorithm in Figure 8.*

Proof: From Theorem 5, 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 8. 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 *Given the CFG $G = (V, E)$, its J-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 8.
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 J-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 5, the only non-trivial *scc* in the ω-DF graph is $\{b, c\}$, as is seen in Figure 7. By collapsing this scc, we get the J-reduced CFG and its dominator tree shown in Figure 7.

It remains to compute a topological sort of the DF graph of the J-reduced CFG (without building this DF graph explicitly). Intuitively, this is accomplished by topologically sorting the children of each node according to the ω-DF graph of the J-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.

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

      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;
}
```

Figure 8: Building the ω-DF graph

Definition 8 *Given a 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 J-reduced CFG of the running example is shown in Figure 7. 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 5.

Proposition 6 *An ω-ordering of a reduced CFG $G = (V, E)$ is a topological sorting of the corresponding dominance frontier graph (V, DF).*

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

From Theorem 5, 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. □

Proposition 7 *An ω-ordering of a reduced CFG $G = (V, E)$ can be computed in time $O(|E|)$.*

We state without proof that a control flow graph is reducible (in the classical sense that deletion of edges going from a node to one of its dominators leaves a dag [ASU86]) if and only if it is J-reduced. Therefore, for reducible CFG's, there is no need to look for scc's in the ω-DG graph to compute ω-orderings.

In the rest of this section, we assume that the CFG is J-reduced.

5.2 Mark Pulling Algorithm

The mark pulling algorithm is a variation of the edge scan algorithm of Section 4.1. To implement the mark pulling approach efficiently, we process the nodes in ω-ordering and maintain, for each node w, an *off/on* binary tag, initially turned off and turned on when the first ancestor of w in the dominator tree is marked and processed. Specifically, just before completing processing of a node v, we check if v is marked, and if so, we perform a top-down walk of the dominator subtree rooted at v, turning on all visited nodes. It is easy to see that if we visit a node x already turned on, 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 mark a node v (that is, to add v to $DF^+(S)$), up-edges incoming into v are examined one by one (processing can stop as soon as v is marked). If any up-edge $u \to v$ is encountered such that u is turned on, then v is marked. From Theorem 5, the ancestors of u that precede v in the ω-ordering are precisely the set of DF predecessors of v in the set $[u, idom(v))$. Hence, v is output if and only if one of its DF predecessors is marked. (Note that the predecessor could be v itself, if $v \in S$ and there is a self-loop at v.) Clearly, the overall work to examine and process the up-edges is $O(|E_{up}|) = O(|E|)$.

Pseudocode for the outlined procedure is shown in Figure 9. At the end of the algorithm, all nodes that are in $DF^+(S)$ are marked, while all descendants of these nodes have their off/on bit turned on. The above discussion can be summarized as follows.

Proposition 8 *The mark pulling algorithm for SSA of Figure 9 has preprocessing time* $T_p = O(|V| + |E|)$*, preprocessing space* $S_p = O(|V| + |E|)$*, and query time* $T_q = O(|V| + |E|)$*.*

5.3 Mark Pushing Algorithm

The mark pushing algorithm examines nodes in ω-order, but in contrast to the mark pulling algorithm which examines the DF-predecessors of a node to determine whether to mark that node, the mark pushing algorithm propagates marks from a marked node w to all the DF-successors of w. To avoid doing work proportional to the size of the DF relation, it takes advantage of the fact that once marks have been propagated from a node v to all the nodes in $DF(v)$, these nodes do not have to be considered again in computing the dominance frontier of the parent of v in the dominator tree (or for that matter, any ancestor of v in this tree). If the parent node p remains unmarked at the end, its DF set is irrelevant to computing the output; if, on the other hand, it is marked at some point, nodes in $DF(v) \cap DF(p)$ can be ignored when propagating marks from p since these nodes were marked when v was visited earlier. Exploiting this simple observation gives us a ϕ-placement algorithm that achieves linear bounds for T_p, S_p and T_q.

The code for the mark pushing algorithm is a minor variation of the node scan algorithm in Section 4.2, and is shown in Figure 10. At each node w, it computes the set $PDF(w)$ (pruned dominance frontier) which is a subset of the DF relation. If node w is marked, marks are propagated to nodes in $PDF(w)$ by invoking the $Push$ method. Some of the nodes in $PDF(w)$ may already be marked; for example, a node in α-$DF(w)$ may be in the dominance frontier of a marked node preceding w in the ω-ordering. Therefore, the $Push$ method propagates a mark to a node only if that node is unmarked.

To obtain $O(|V| + |E|)$ query time, the PDF computation at a node must be done carefully. The simplest approach that achieves this complexity is to be lazy in doing the unions and deletions for computing PDF sets. Each PDF set is represented as a singly linked list with a header that

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

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

Figure 9: Mark Pulling Algorithm

has a pointer to the start and the end of the list. The pointer to the end of the list permits constant time concatenations. When a node v is visited, its α-DFlist is simply concatenated with the PDF lists from its children (so the operator $\cup$ in Line 9 of procedure **MarkPush** should be interpreted as list concatenation). The resulting list may have several entries for a given node (for example, if a node is contained in the PDF sets of two or more of v's children); it may also contain nodes that are children of v since no deletions are performed in the lists passed up from the children. In set-theoretic terms, this list represents a multi-set which is a superset of the actual PDF set of node v. If v is not marked, this list is simply passed unexamined to v's parent when the parent node is visited. If v is marked, the nodes in this list are examined, and a node is marked only if (i) it is not already marked, and (ii) it is not a proper descendant of v. It is easy to see that this lazy union and deletion is correct, and that the overall query complexity is $O(|V| + |E|)$.

Proposition 9 *The mark pushing algorithm for ϕ-placement of Figure 10 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: From Theorem 6, it follows that the set $PDF(v)$ computed in line 4 of procedure ϕ-placement is a subset of $DF(v)$; therefore, every node that is marked is in $J(S)$. We must also show that every node in $DF^+(S)$ gets marked by this procedure; that is, that using the PDF set (computed in procedure **MarkPush**) rather than DF to mark nodes does not change the set of marked nodes. This is an induction on the length of the ω-order. The first node in this order must be a leaf node, and for a leaf node, the $PDF(v)$ set is identical to the $DF(v)$ set. Assume inductively that for all nodes n before w in the ω-order, nodes in $DF(n) - PDF(n)$ are marked. Since all the children of w precede it in the ω-order,

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

{
Initialize $DF^+(S)$ set to $\{\}$;
Initialize all nodes as unmarked;
Initialize PDF sets of all nodes to $\{\}$;
Mark all nodes in S;

for each CFG edge $(u \rightarrow v)$ **do**
   if $(u \neq idom(v))$ α-$DF(u) = \alpha$-$DF(u) \cup \{v\}$
od
for each node v in ω-ordering **do**
   $PDF(v) = \alpha$-DF(v) $\cup$ $(\cup_{c \in children(v)} PDF(c) - children(v))$;
   if v is marked **then** Push(v) **endif** ;
od
}

Procedure Push(x);

{
for each node m in $PDF(x)$ **do**
   if (m is not marked) **then**
      Mark m;
      Add m to $DF^+(S)$;
   endif
od
$PDF(x) = \{\}$; $//PDF(x)$ is now irrelevant for parent's computation
od
}

Figure 10: Mark Pushing Algorithm

it is easy to see that all nodes in $DF(w) - PDF(w)$ are marked after w has been visited, satisfying the inductive hypothesis. □

5.4 Discussion

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

The mark pulling algorithm can be viewed as an efficient version of the reachability algorithm of Figure 4. 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 mark pulling algorithm visits all nodes that are reachable from v in the reverse CFG along a *single edge* that does not contain (ie., originate from) $idom(v)$. The mark 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 mark pushing algorithm, the preprocessing step of computing a J-reduced graph can be eliminated and nodes can simply be considered in bottom-up order in the dominator tree, but at the cost of having to revisit a node if it gets marked after it has been visited for computing its PDF set. We do not believe preprocessing can be eliminated for the mark pulling algorithm. Cytron and Ferrante have described a lock-step algorithm similar to the mark pushing algorithm that they call *on-the-fly* computation of join sets [CF93]. Their algorithm is considerably more complicated than ours because there is no preprocessing step to determine scc's of the DF graph. They use path compression to get $O(|E|\alpha(|E|))$ query complexity.

6 Lazy Algorithms

A drawback of lock-step algorithms is that they visit all the nodes in the CFG even if some of these nodes are not in $J(S)$ for the initial set of nodes S. In this section, we discuss algorithms that compute EDF sets *lazily*; that is, they compute a set $EDF(w)$ only if it is determined that w belongs to $J(S)$. Although the asymptotic complexity of these algorithms is not better than that of the lock-step algorithms of Section 5, lazy algorithms can lead to savings since they do not spend effort on parts of the DF graph that are irrelevant for computing $J(S)$. As we show in Section 7, the lazy algorithm described in this section outperforms lock-step algorithms.

We first discuss a lazy algorithm that is optimal for computing EDF sets. This approach is based on ideas that were developed by us to solve the problem of computing the control dependence relation of a CFG optimally [BP96]. Then, we apply these results to the ϕ-placement problem.

6.1 The Augmented Dominator Tree

One way to compute $EDF(w)$ when the EDF sets of w's children are not immediately available is to appeal directly to the definition of EDF in Definition 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 11. In this procedure, each node u of the dominator tree is assumed to have a node list L containing all up-edges whose source is u. A node is assumed to be a *boundary node* iff it is a leaf node of the dominator tree. For the running example of Figure 5, 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.

```
Procedure TopDownEDF(QueryNode);
{
      EDF = {};
      Visit(QueryNode, QueryNode);
      return EDF;
}
Procedure Visit(QueryNode, VisitNode);
{
      for each edge (u → v) ∈ L[VisitNode] do
        if idom(v) is a proper ancestor of QueryNode
          then EDF = EDF ∪ {(u → v)}; endif
        od ;
      if VisitNode is not a boundary node
        then
          for each child C of VisitNode
            do
              Visit(QueryNode,C)
            od ;
      endif ;
}
```

Figure 11: Top-down Query Procedure for EDF

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 *eager* approach and precompute the entire EDF graph, store each $EDF(w)$ in list $L(w)$, let every node be a boundary node, and *still use* **TopDownEDF**. 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 let some arbitrarily selected subset of nodes be boundary nodes; for each boundary node b, we precompute $EDF(b)$ and store this set in list $L(b)$, while for each interior node u, we store only up-edges whose source is u. Formally, we have the following definition.

Definition 9 *If w is an interior node, let $L[w]$ = α-$EDF(w)$; otherwise w is a boundary node and $L[w] = EDF(w)$.*

In Figure 12, boundary nodes are shown as solid dots while interior nodes are shown as hollow dots. Since `START` and `END` play no role in EDF computations, we ignore them in this discussion (for technical reasons, they are classified as boundary nodes as explained later). Figure 12(a) shows one extreme in which the only boundary nodes are the leaves of the tree. For a leaf node w, $EDF(w)$ = α-$DF(w)$, therefore the only EDF edges stored explicitly for this choice of boundary nodes are α-EDF edges. Figure 12(b) shows the other extreme in which all nodes are boundary nodes. Therefore, all EDF edges are stored explicitly. Figure 12(c) shows an intermediate point in which nodes $\{a, d, e, f, h\}$ are boundary nodes.

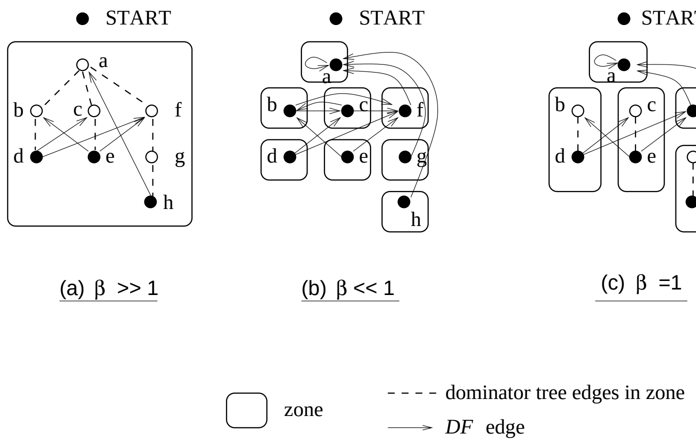


Figure 12: Zone Structure for Different Values of β

Since procedure **TopDownEDF** never traverses an edge from a boundary node to any of its children, we can imagine that such edges are deleted from the dominator tree leaving this tree partitioned into smaller trees called *zones*. Ignoring `START` and `END` in Figure 12(c), there are five zones containing the following sets of nodes: $\{a\}$, $\{b, d\}$, $\{c, e\}$, $\{f\}$, $\{g, h\}$. The portion of a zone below a node q will be called the *subzone* associated with q. The significance of subzones is that a query **TopDownEDF(v)** visits all the nodes in the subzone associated with v. Formally:

Definition 10 *A node w is said to be in the* subzone 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 subzone; that is, a subzone that is not strictly contained in any other subzone.*

It is easy to see that procedure **TopDownEDF** works correctly regardless of the choice of boundary nodes, provided all leaf nodes are boundary nodes to ensure the termination of the recursive procedure **Visit**. If the query node w is a boundary node, then $L(w) = EDF(w)$ by construction, so it suffices simply to enumerate the edges in $L(w)$, which is what procedure **TopDownEDF** does. If the query node w is an interior node, the procedure visits all nodes in the sub-zone below w but it does not look below any boundary node of this sub-zone. It is easy to argue that this is correct. Suppose $(u \to v) \in EDF(w)$. If u is the same zone as w, procedure **TopDownEDF** outputs v when it visits node u; otherwise, $(u \to v) \in EDF(b)$ where b is the first boundary node on the tree path from w to u, and the procedure outputs $(u \to v)$ when it visits b.

So far, we have not specified an order in which nodes should be stored in the L lists at each node. If a list $L(w)$ contains edges $(u_1 \to v_1), (u_2 \to v_2), \ldots$, we note from Lemma 2 that the nodes $idom(v_1), idom(v_2), \ldots$ dominate w and are therefore totally ordered by dominance. Examination of list $L(w)$ during query processing can be made more efficient if 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** can terminate as soon as a node v is encountered where $idom(v)$ does not strictly dominate the query node.

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 [LT79, Har85].) 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:

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

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

$$|Z_q| \leq \beta|EDF(q)| + 1 \; . \tag{2}$$

Intuitively, the number of nodes visited when q is queried is at most one more than some constant proportion of the answer size. The additive term of 1 prevents zone Z_q from becoming empty when $EDF(q)$ is empty. By combining Equations (1) and (2), we obtain

$$Q_q = O((\beta + 1)|EDF(q)|). \tag{3}$$

Thus, query time is basically proportional to the output size, and asymptotically optimal for constant β. Parameter β controls the degree of caching, yielding the full EDF relation ($|V|$ zones of size one) when it is small enough (i.e., $\beta < 1/\max_q |EDF(q)|$) and the α-EDFrelation (a single zone of size $|V|$) when it is large enough (i.e., $\beta \geq \max_q(|D_q| - 1)/|EDF(q)|$, where D_q is the set of nodes dominated by q).

6.1.1 Defining Zones

Can we define zones so as to satisfy Inequality (2) 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 without violating Inequality (2), enforcing 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 11 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 $(\beta|A_v| + 1)$, Inequality (2) fails, so we make v a boundary node. Zones are simply maximal subzones.

To analyze the storage requirements implied by the above definition, let X denote the set of boundary nodes that are not leaves. If $w \in (V - X)$, then only α-DFedges 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:

$$\sum_{w \in V} |L[w]| = \sum_{w \in (V-X)} |L[w]| + \sum_{w \in X} |L[w]| \leq |E_{up}| + \sum_{w \in X} |EDF(w)|. \tag{4}$$

From Definition 11, if $w \in X$, then

$$|EDF(w)| < \sum_{u \in children(w)} |Z_u|/\beta. \tag{5}$$

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

When we sum over $w \in X$ both sides of Inequality (5), we see that the right hand side evaluates at most to $|V|/\beta$, since all subzones Z_u's involved in the resulting double summation are disjoint. Hence, $\sum_{w\in X} |EDF(w)| \leq |V|/\beta$, which, used in Relation (4) yields:

$$\sum_{w\in V} |L[w]| \leq |E_{up}| + |V|/\beta. \tag{6}$$

In conclusion, to store $\mathcal{ADT}$, 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 (6), $O(|E_{up}| + |V|/\beta)$ for the list elements. All together, we have $S = O(|E_{up}| + (1 + 1/\beta)|V|)$.

The design parameter β embodies a tradeoff between query time (increasing with β) and preprocessing space (decreasing with β). The full EDF graph, resulting when $\beta < 1/\max_q |EDF(q)|$, will generally require superlinear space $S = O(|E_{up}| + (1 + \max_q |EDF(q)|)|V|)$. The α-DFgraph, resulting when $\beta \geq \max_q(|D_q| - 1)/|EDF(q)|$, where D_q is the set of nodes dominated by q, will generally require superlinear query time $Q_q = O((\max_x(|D_x| - 1)/|EDF(x)| + 1)|EDF(q)|)$. Small constant values such as $\beta = 1$ yield a reasonable compromise, leading to linear space and query time. Figure 12(c) shows the zone structure of the running example for $\beta = 1$.

6.1.2 $\mathcal{ADT}$ Construction

We now describe an algorithm to construct the search structure $\mathcal{ADT}$ in linear time. The preprocessing algorithm 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 $\beta \geq 0$, which controls the space/query-time tradeoff, as described in the previous subsection.

The preprocessing algorithm consists of a sequence of few simple stages explained below and translated into pseudocode in Figure 13.

1. *For each node x, compute the number $b[x]$ (resp. $t[x]$) of up-edges $(u \to v)$ with $u = x$ (resp., $idom(v) = x$).* 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 constant work done for each of the $|E_{up}|$ routes.

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 the previous step, 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 11 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. A special provision is made for the root of T, whose next boundary is set by convention to $-\infty$, considered as a proper ancestor of any node in the tree.

5. *Construct list $L[x]$ for each node x.* By Definition 9, given an up-edge $(u \to v)$, v appears in list $L[x]$ for $x \in W$, where W 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, let $W = \{w_0 = b, w_1, ..., w_k\}$, where $w_i = NxtBndry[w_{i-1}]$, 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 node x in the corresponding set W. 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 7 *Given a CFG, the corresponding augmented dominator tree can be constructed in time $T = O(|E_{up}| + (1 + 1/\beta)|V|)$ and stored in space $S = O(|E_{up}| + (1 + 1/\beta)|V|)$. A query to the edge dominance frontier of a node q can be answered in time $Q_q = O((\beta + 1)|EDF(q)|)$.*

6.2 SSA Computation Based on $\mathcal{ADT}$

We now consider how to implement the mark pushing paradigm using the $\mathcal{ADT}$ data structure to compute edge dominance frontiers.

First, consider the problem of using the $\mathcal{ADT}$ data structure to compute $\cup_{s \in S} EDF(s)$ where S is a set of nodes known before any queries are made. To avoid visiting tree nodes repeatedly during querying, it is desirable to sort the nodes in S by level number in the tree, and query them in bottom-up order. A node is marked when it is queried, and the query procedure of Figure 11 is modified so that it never visits nodes below a marked node. Therefore, the time for querying all the nodes in S is proportional to the sum of the number of nodes visited and the number of paths in the L_v lists of these nodes. In the worst case, every node in the tree is visited, in which case, using Equation 6, we can see that the total time for querying is

$$Q = O(|V| + |E_{up}| + |V|/\alpha). \tag{7}$$

Since $|E_{up}| \leq |E|$ in our context, this bound can be written as $O(|E| + (1 + 1/\alpha)|V|)$, which, for constant α, is proportional to program size. Sorting the set S initially by level can be done in $O(|V|)$ time and space quite simply by inserting nodes into an array whose length is equal to the level of the tree. Therefore, we can find the set $\cup_{s \in S} EDF(S)$ for a fixed set of nodes S in time proportional to program size.

If S is given online, we cannot sort it before starting queries. However, as long as nodes are presented for querying in order of decreasing level number, the approach discussed above can be

```
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
}
```

Figure 13: Constructing the $\mathcal{ADT}$ Structure

used. To accomplish this in our context, we maintain a priority queue [CLR92] of nodes that must be queried; the key for the priority queue is level number in the tree. Initially, this priority queue contains only the nodes in S. At each step, a node w of highest level is extracted from the priority queue, and a query is made in the $\mathcal{ADT}$ data structure with that node; if there is a node u in the answer to the query that is not in the output set, it is added to the output set and itself inserted into the priority queue. From Lemma 2, it follows that the level number of u is less than or equal to the level number of w. Therefore, querying a node at level l can never cause the insertion of a node at a level strictly greater than l into the priority queue.

The priority queue can be implemented using a heap, with $O(\log(K))$ time per operation [CLR92] where K, the number of keys, is equal to the number of levels of the dominator tree; a more sophisticated data structure, exploiting the integer nature of the keys, achieves $O(\log(\log(K))$ time per operation [VEBKZ77]. One can also exploit the constraint on insertions to produce a simpler data structure. For example, an array M of K lists, one for each possible key in ascending order, can be used. Insertion of an element with key r is performed, in time $O(1)$, by appending it to list $M(r)$. Extraction of an element with minimum key entails scanning the array from the component last examined to the first component whose list is not empty. Hence, the total time for N extractions is $O(N + K)$.

The algorithm is described in Figure 14.

6.3 Discussion

At the extreme in which every node is a boundary node, the $\mathcal{ADT}$ algorithm reduces to a two-phase approach since the DF set of every node is computed and stored before any queries are executed. Obviously, this blows up storage requirements. At the other extreme where only leaf nodes (plus `START` and `END`) are boundary nodes, the $\mathcal{ADT}$ reduces to the fully lazy approach of Sreedhar and Gao [SG95]. This extreme is not competitive in practice because of the expense of having to visit all the nodes in the dominator tree below query nodes, as we show in Section 7. Controlled caching of EDF information in the $\mathcal{ADT}$ improves query time without blowing up preprocessing space and time requirements, and therefore outperforms the two extremes.

We have not made use of ω-ordering in the algorithm described in this section. It is possible to dispense with the priority queue in Figure 14 if we process nodes in ω-order; however, in practice, the expense of computing the ω-order does not seem to be worthwhile.

7 Experimental Results

In this section, we present experimental results comparing the performance of ϕ-placement algorithms for a standard model problem and for the SPEC benchmarks.

For SSA problems, the standard model problem is a repeat-until loop nest such as the one shown in Figure 15. Even though the program is structured, its DF relation grows quadratically with program size, making it a worst-scase scenario for two-phase algorithms. For the experiments reported here, we used a 200-deep loop nest - although such a 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 algorithm of Figure 14 to compute $J(n)$ for different nodes n in the program, and measured the time for doing this as a function of β on a SUN-4. In the 3D plot in Figure 16, the x axis is the value of $log_2(\beta)$, the y-axis is the node number of node n, and the z-axis is the time for computing $J(n)$. The 2D plot in this figure shows slices parallel to the YZ plane of the 3D plot for three different values of β - a very large value, a very small value and 1.

```
Procedure φ-placement (S: set of nodes);
{
    % ADT data structure, is global
    Initialize a Priority Queue PQ;
    J = {}; Set of join 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
      q := ExtractMax(PQ);//q is deepest in tree
      QueryIncr(q);
    od ;
    Delete marks from nodes in T;
    Output J;
}
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
          J = J ∪ {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 ;
}
```

Figure 14: ADT-based Algorithm for ϕ-placement

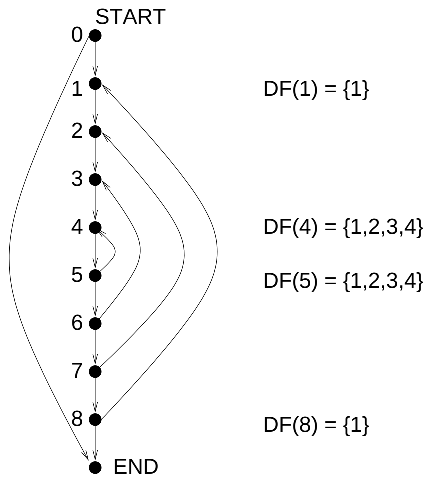


Figure 15: Repeat-until Loop Nest

From these plots, it is clear that for small values of β, the time to compute J grows quadratically as we go from outer loop nodes to inner loop nodes. In contrast, for large values of β, this time is essentially constant. These results can be explained analytically as follows. The time to compute J sets depends on the number of nodes and the number of EDF graph edges that are visited during the computation. The following formula gives the DF sets of nodes in a loop nest with l loops.

$$1 \le n \le l \quad : DF(n) = \{1, 2, ..., n\}$$
$$l < n \le 2l \quad : DF(n) = \{2l - n + 1, 2l - n, ..., 1\}$$

For very small values of β, the DF set of every node is stored at the node. If we assume that $1 \le n \le l$, computing $J(n)$ requires a visit to all nodes in the set {1,2,...,n}. The number of DF edges examined during these visits is $n + (n-1) + (n-2) + ... + 1$ which is $n(n+1)/2$; each of these edge traversals involves a visit to the target node of the DF edge. Therefore, with full caching, computing $J(n)$ requires visiting $n(n+1)/2 + 1$ nodes and traversing $n(n+1)/2$ DF edges. 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 J set computation if full caching is used.

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

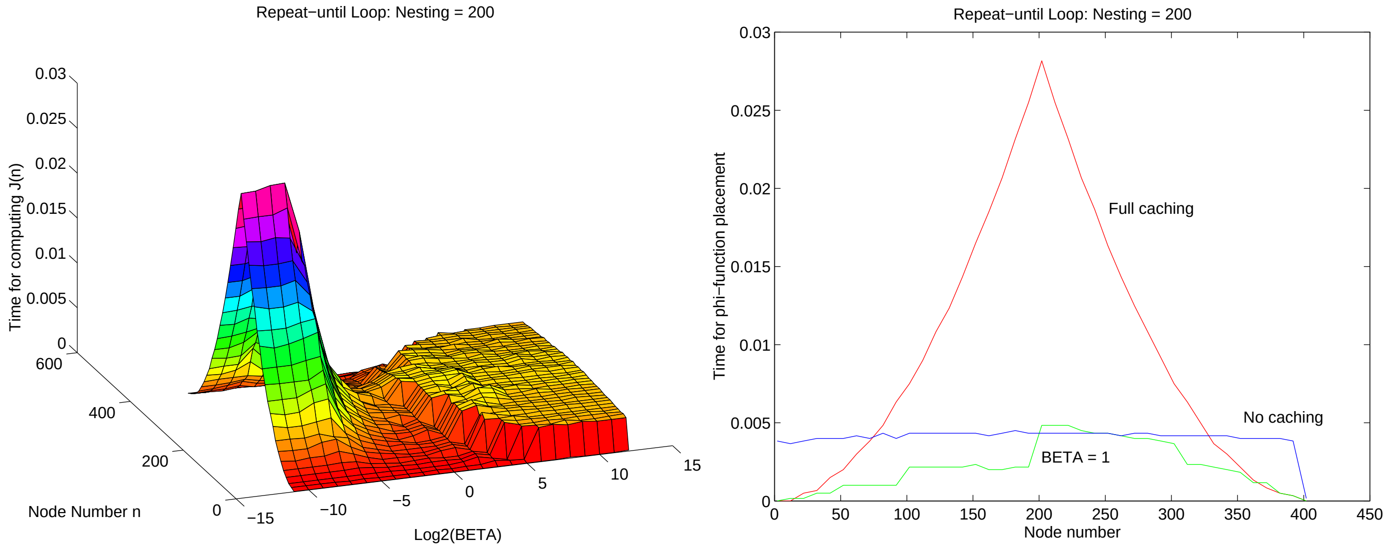


Figure 16: Time for ϕ-function Placement in Model Problem

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 16, 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 16 along 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 17 shows the time required to do ϕ-function placement for all the programs in the SPEC benchmarks. It can be seen that full caching outperforms no caching by a factor of four. In [SG95], Sreedhar and Gao reported that their algorithm, which is the same as our $\mathcal{ADT}$ algorithm with no caching, outperformed full caching by factors of 5 to 10 on the SPEC benchmarks. These measurements were apparently erroneous, and new measurements taken by them are in line with our numbers [SG96]. Using $\beta = 1$ gives the best performance, although the performance advantage over full caching is small in practice.

Our experiments also 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 mark pulling algorithm is a remarkably simple ϕ-placement algorithm that achieves linear space and time bounds for preprocessing and query, but for the SPEC benchmarks, for example, the time it took for ϕ-placement was almost 10 seconds, an order of magnitude slower than the best $\mathcal{ADT}$ algorithm.

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

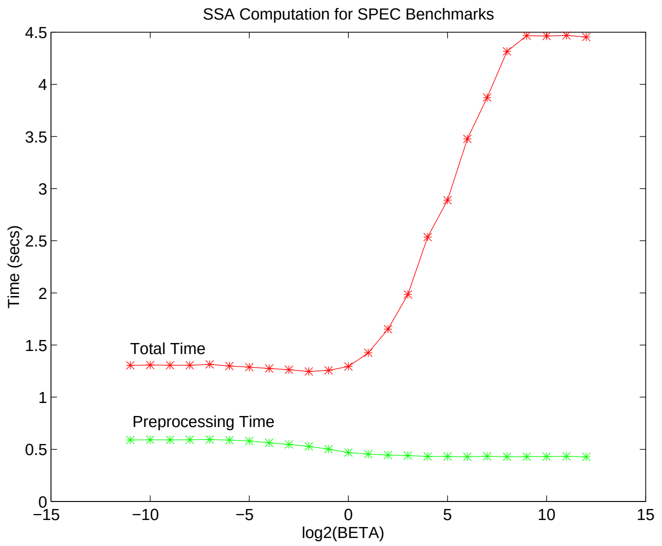


Figure 17: ϕ-function placement for SPEC Benchmarks

8 Conclusions

In this paper, we have given a unified presentation of all known results about the SSA form, and we have proposed a framework in which all existing SSA algorithms can be placed. Considerations of this framework led us to invent two new algorithms which perform ϕ-placement in time and space that grow linearly with the size of the program.

We showed that the problem of ϕ-placement requires computing reachability from sets of nodes in the DF graph of the CFG. Two-phase algorithms compute the DF graph in its entirety before doing reachability computations. The problem with these algorithms is that the size of the DF relation grows quadratically with program size. Lock-step algorithms visit all the nodes in the graph, but compute a subset of the full DF relation required to answer the reachability query. Lazy algorithms compute the DF graph on demand without visiting all the nodes of the CFG. Their efficiency can be improved by precomputing and caching DF sets at carefully selected nodes. At the extreme where this precomputation is done for all nodes, the selectively lazy approach reduces to the two-phase approach. However, by carefully choosing the nodes for which precomputation of DF sets is done, we obtain an algorithm for which preprocessing space and time and query time grow proportionally with the size of the program. This algorithm is also the most efficient one for practical problems.

Although we have presented a number of SSA algorithms that achieve $O(|E|)$ preprocessing time, preprocessing space and query time, none of these algorithms is necessarily optimal for doing ϕ-placement for multiple variables since the query time for a multi-variable ϕ-placement problem would ideally be proportional to the size of the *output* (the size of $\phi(S)$ in other words). Is there such an algorithm for ϕ-placement?

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BP96] 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.

[BKRW98] 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.

[CF93] 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.

[CFR+91] 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.

[CLR92] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1992.

[GL81] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Inc., 1981.

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

[Har85] D. Harel. A linear time algorithm for finding dominators in flowgraphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, Providence, Rhode Island, May 6–8, 1985.

[JP93] 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.

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

[PB95] K. Pingali and G. Bilardi. $\mathcal{APT}$: A data structure for optimal control dependence computation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation.* ACM, New York, 32–46, 1995.

[PB97] 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.

[PBJ+91] 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.

[PC90] 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.

[SG95] 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.

[SG96] Vugranam C. Sreedhar and Guang R. Gao. Personal Communication. September 1996.

[SS70] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

[VEBKZ77] P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Weiss92] 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.

[Wolfe95] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.

A Appendix

Definition 12 *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.*

By convention, START is assumed to have an assignment to all the variables in the program.

The notation $x \xrightarrow{*} y$ denotes a path from x to y, and $x \xrightarrow{+} y$ denotes a path from x to y containing at least one edge.

Definition 13 *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 $idom(v)$. 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 using a more complicated algorithms due to Buchsbaum et al. [BKRW98].

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 14 *The* def-use graph *of a control flow graph* $G = (V, E)$ *is a graph* $D = (V, F)$ *which has the same vertices as* G *and in which edge* (n_1, n_2) *exists if* n_1 *is a definition of a variable* 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) *is an edge in the def-use graph, 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.

Theorem 8 *For any* $w \in V$*,* $v \in J(w)$ *iff there exists a path* $w \xrightarrow{+} v$ *that does not contain* $idom(v)$*.*

Proof: ($\Rightarrow$) If $v \in J(w)$, Definition 2 asserts that there are paths $P1 = \texttt{START} \xrightarrow{+} v$ and $P2 = w \xrightarrow{+} v$ which intersect only at v. Since, by Definition 13, 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., $w \in J(v)$.

Base case: Let the length of P be 1, i.e., P consists only of edge $w \to v$. 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 1(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 $P = w \to y \xrightarrow{+} v$. 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 $w \to y$ to the path $R2$ and consider the following two cases:

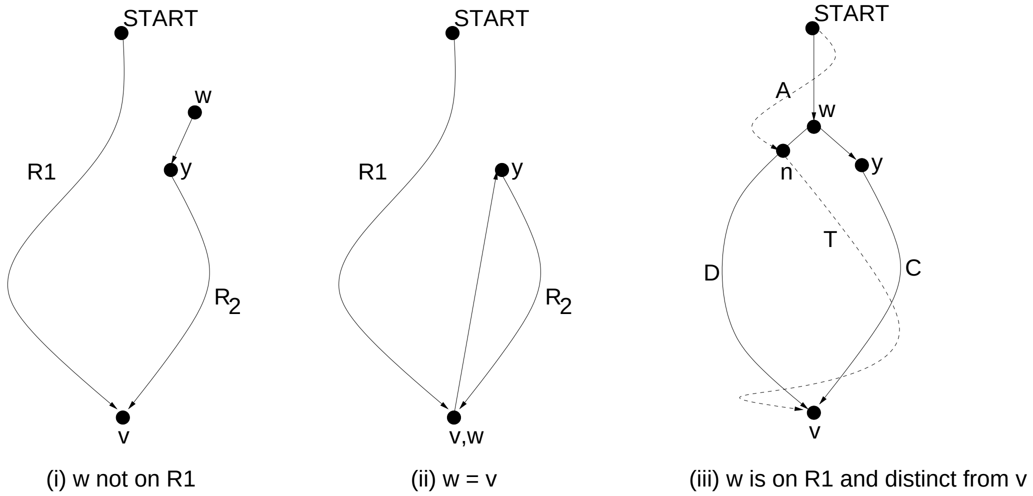


Figure 18: Case Analysis for Theorem 8.

- $w \notin (R1 - \{v\})$. Then, let $P1 = R1$ and $P2 = C$. Figures 18(i) and (ii) illustrate the subcases $w \neq v$ 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 18(iii)). Let also $T = \mathtt{START} \xrightarrow{+} v$ 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 18, let $P1$ be the concatenation of the prefix $\mathtt{START} \xrightarrow{+} n$ 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.

□