An Efficient Method of Computing Static Single Assignment Form

Ron Cytron*
Jeanne Ferrante*
Barry K. Rosen*
Mark N. Wegman*
F. Kenneth Zadeck†

1 Introduction

In optimizing compilers, data structure choices directly influence the power and efficiency of practical program optimization. A poor choice of data structure can inhibit optimization or slow compilation to the point where advanced optimization features become undesirable. Recently, static single assignment form [AWZ88, RWZ88] and the control dependence graph [FOW87] have been proposed to represent data flow and control flow properties of programs. Each of these previously unrelated techniques lends efficiency and power to a useful class of program optimizations. Although both of these structures are attractive, the difficulty of their construction and their potential size have discouraged their use [AJ88]. We present a new algorithm that efficiently computes these data structures for arbitrary control flow graphs. We also give analytical and experimental evidence that they are usually *linear* in the size of the original program. This paper thus presents strong evidence that these structures can be of *practical* use in optimization.

After a program has been transformed into static single assignment (*SSA*) form, it has two useful properties:

1. Each programmer-specified use of a variable is reached by exactly one assignment to that variable.

2. The program contains ϕ-functions, as described in Section 2, that distinguish values of variables transmitted on distinct incoming control flow edges.

A precursor [SS70] of SSA form obtains Property 1 by inserting assignments from variables to themselves at appropriate places in the program. By inserting explicit ϕ-functions instead, SSA form leads to simpler formulations of works like [CLZ86, WZ85] that are based on the precursor.

Property 1 has been exploited by a constant propagation algorithm that deletes branches to code proven unexecutable at compile-time [WZ88]. Without SSA form, data flow information might have to be recomputed each time branches are deleted, rendering the algorithm excessively costly. Static single assignment form nicely summarizes those conditions relevant to code motion [CLZ86, RWZ88]. Additionally, the representation of simple data flow information (def-use chains) is more compact through SSA form. If a variable has D definitions and U uses, then there can be $D * U$ def-use chains. When similar information is encoded in SSA form, there can be at most E def-use chains, where E is the number of edges in the control flow graph [RL86].

Exploitation of Property 2 has led to a *global* value-numbering algorithm that can track redundant computations across control flow paths [RWZ88] and an algorithm for detecting program equivalence [AWZ88]. There is also an algorithm for increasing parallelism in imperative programs through a renaming transformation [CF87b] that is rather like SSA form.

Control dependences [FOW87, CF87a] identify those conditions affecting statement execution. Informally, a statement is control dependent on a branch if one edge from the branch definitely causes that statement to execute while another edge can cause the statement to be skipped. Such information is vital for detection of parallelism [ABC+88], program optimization, and additionally program analysis [HPR88].

Section 2 explains SSA form. Section 3 introduces a new structure called *dominance frontiers*. Then we show how to compute SSA form (Section 4) and the control dependence graph (Section 5) efficiently using dominance frontiers. Section 6 shows that our algorithms behave linearly with respect to program size for programs restricted to certain control structures. We also give evidence of general linear behavior by reporting on experiments with FORTRAN programs.

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

†Computer Science Dept., Brown University, Providence, RI 02912.

```
I ← 1                          I1 ← 1                                  (1)
J ← 1                          J1 ← 1                                  (1)
K ← 1                          K1 ← 1                                  (1)
L ← 1                          L1 ← 1                                  (1)
repeat                         repeat                                  (2)
                                  J2 ← φ(J4, J1)                       (2)
                                  K2 ← φ(K5, K1)                       (2)
                                  L2 ← φ(L9, L1)                       (2)
   if (P)                         if (P)                               (2)
      then do                        then do                           (3)
         J ← I                          J3 ← I1                        (3)
         if (Q)                         if (Q)                         (3)
            then L ← 2                     then L3 ← 2                 (4)
            else L ← 3                     else L4 ← 3                 (5)
                                        L5 ← φ(L3, L4)                 (6)
         K ← K + 1                      K3 ← K2 + 1                    (6)
      end                            end                               (6)
      else K ← K + 2                 else K4 ← K2 + 2                  (7)
                                     J4 ← φ(J3, J2)                    (8)
                                     K5 ← φ(K3, K4)                    (8)
                                     L6 ← φ(L2, L5)                    (8)
   repeat                            repeat                           (9)
                                        L7 ← φ(L9, L6)                (9)
      if (R)                            if (R)                       (9)
         then L ← L + 4                    then L8 ← L7 + 4           (10)
                                        L9 ← φ(L8, L7)                (11)
   until (S)                         until (S)                        (11)
   I ← I + 6                         I4 ← I2 + 6                      (12)
until (T)                      until (T)                               (12)
```

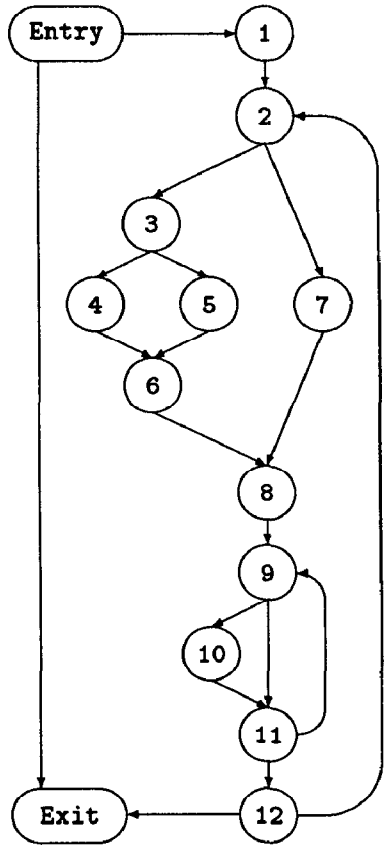


Figure 1. A Simple Program, Its SSA Form and Its Control Flow Graph

2 Static Single Assignment Form

The algorithms presented in this paper work for programs that contain arbitrary control structures. The statements in such programs are restricted to conditional expressions and assignment statements. Only simple, unaliased variables are considered; no arrays or pointer values are considered. Aliasing can be accommodated by techniques in [WZ88].

There are two separate steps required to translate a program into static single assignment (*SSA*) form. In the first step, special assignment statements called *ϕ-functions* are inserted at certain points in the program. In the second step, each variable V is given several new names V_i for various integers i. Each mention of V in the program is replaced by a mention of one of the new names V_i. The SSA form of a simple program is given in Figure 1.

Before explaining SSA form in detail, we review the modeling of program control flow by a directed graph. The statements of a program are organized into (not necessarily maximal) basic blocks, where program flow enters a basic block at its first statement and leaves the basic block at its last statement. Basic blocks are indicated by the column of numbers in parentheses in Figure 1. A *control flow graph* or *CFG* is a directed graph. The nodes of *CFG* are the basic blocks of a program and two additional nodes: `Entry` and `Exit`. There is an edge from `Entry` to any basic block where the program can be entered. There is an edge to `Exit` from any basic block that can exit the program. The other edges of *CFG* represent transfers of control (jumps) between the basic blocks. We assume that each node is on a path from `Entry` and on a path to `Exit`.

For each node X, a *successor* of X is any node Y with an edge $X \rightarrow Y$ in *CFG* and $Succ(X)$ is the set of all successors of X (similarly for predecessors). The control flow graph for our example program is given on the right of Figure 1. For technical reasons related to the representation of control dependences, there is also an edge from `Entry` to `Exit`. Finally, each variable is considered to have an assignment in `Entry` to represent whatever value the variable may have when the program is entered. This assignment is treated just like the ones that appear explicitly in the code.

A ϕ-function has the form $U \leftarrow \phi(V, W, ...)$, where $U, V, W, ...$ are variables and the number of operands $V, W, ...$ is the number of control flow predecessors of the point where the ϕ-function occurs. The control flow predecessors of each point in the program are listed in some arbitrary fixed order, and the j-th operand of ϕ is associated with the j-th predecessor. If control reaches the ϕ-function from its j-th predecessor, then U is assigned the value of the j-th operand. Each execution of a ϕ-function uses only one of the operands, but which one depends on the flow of

control just before the ϕ-function.

For any variable V, one can insert a trivial ϕ-function $V \leftarrow \phi(V, V, ...)$ at the entrance to any CFG node in the program without changing the semantics. Why should one want to perform such insertions? By placing the insertions carefully and then renaming the mentions of V, one can put the program into SSA form. Specifically, we assume that any number of new variables V_i (for $i = 0, 1, 2, ...$) can be generated to serve as new names for V. The transformed program is defined to be *in SSA form* if, for every original variable V, ϕ-functions for V have been inserted and each mention of V has been changed to a mention of a new name V_i such that the following conditions hold:

1. If a CFG node Z is the first node common to two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ that start at nodes X and Y containing assignments to V, then a ϕ-function for V has been inserted at entrance to Z.

2. Each new name V_i for V is the target of exactly one assignment statement in the program text.

3. Along any control flow path, consider any use of a new name V_i for V (in the transformed program) and the corresponding use of V (in the original program). Then V and V_i have the same value.

A program is in *minimal* SSA form if it is in SSA form and if the number of ϕ-functions inserted is as small as possible.[1]

The optimizations that depend on SSA form are still valid if there are some extraneous ϕ-functions, beyond those that would appear in minimal SSA form. Extraneous ϕ-functions sometimes inhibit optimization by concealing useful facts; extraneous ϕ-functions always add unnecessary overhead to the optimization process itself. Thus it is important to place ϕ-functions only where they are required.

For any variable V, the CFG nodes where we should insert ϕ-functions in the original program can be defined recursively by Condition 1 in the definition of SSA form. A node Z *needs a ϕ-function for* V if Z is the first node that two nonnull control flow paths have in common, when those two paths originate at two different nodes containing assignments to V or needing ϕ-functions for V. Nonrecursively, we may observe that a node Z needs a ϕ-function for V because Z is the first node common to two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ that start at nodes X and Y containing assignments to V. If Z did not already contain an assignment to V, then the ϕ-function inserted at Z adds Z to the set of nodes that contain assignments to V. With more nodes to consider as origins of paths, we may observe more nodes appearing as the first node common to two nonnull paths originating at nodes with assignments to V. The set of nodes observed to need ϕ-functions will gradually increase until it stabilizes. When ϕ-functions are placed this way, minimal SSA form can be obtained by an easy adaptation of well-known def-use chaining. The algorithm presented in this paper obtains the same end results as this brute-force approach, but it places the ϕ-functions and performs the renaming in much less time than brute force would require.

Minimal SSA form is a refinement of Shapiro and Saint's [SS70] notion of a pseudo-assignment. The *pseudo-assignment* nodes for V are exactly the nodes that need ϕ-functions for V. For a CFG with E edges that describes a program with V variables, one algorithm [RT82] requires $O(E\alpha(E))$ bit vector operations (where each vector is of length V) to find all the pseudo-assignments. A simpler algorithm [RWZ88] for reducible programs computes SSA form in time $O(E \times V)$. Both of these algorithms are effectively quadratic, and the [RWZ88] algorithm sometimes uses extraneous ϕ-functions. The method proposed here is $O(E + T + DF)$, where T is the total number of ordinary assignments and ϕ-functions and DF is the total size of all dominance frontiers (we describe this structure later). While the numbers T and DF can be quadratic in the size of the program, we give evidence that they are rarely so. The insight allowing us to obtain a bound that does not grow multiplicatively with the number of variables is that we can decide where to insert ϕ-functions from the dominance frontiers. The size of the dominance frontiers depends only on the control flow of the program.

3 Dominance Frontiers

In this section we introduce the *dominance frontier* mapping and give an algorithm for its computation. We then relate proper location of ϕ-functions to dominance frontiers.

Before proceeding, we review the dominance relation [Tar74] between nodes in the control flow graph. Let X and Y be nodes in CFG. If X appears on every path from `Entry` to Y, then X *dominates* Y. Domination is both reflexive and transitive. If X dominates Y and $X \neq Y$, then X *strictly* dominates Y. In formulas, we write $X \gg Y$ for strict domination and $X \gtrdot Y$ for domination. To say that X does *not* strictly dominate Y, we write $X \not\gg Y$. The *immediate dominator* of Y (denoted $idom(Y)$) is the closest strict dominator of Y on any path from `Entry` to Y. In a *dominator tree*, the children of a node X are all immediately dominated by X. Let E be the number of edges in CFG. The dominator tree of CFG can be constructed in $O(E\alpha(E))$ time [LT79] or (by a more difficult algorithm) in $O(E)$ time [Har85].

The dominator tree of CFG has exactly the same set of nodes as CFG but has a very different set of edges. The words *predecessor, successor, path* always refer to CFG here. The words *parent, child, ancestor, descendant* always refer to the dominator tree.

3.1 Definition and Algorithm

The *dominance frontier* $DF(X)$ of a CFG node X is the set of all CFG nodes Y such that X dominates a predecessor of Y but does not strictly dominate Y:

$$DF(X) = \{ Y \mid (\exists P \in Pred(Y))(X \gtrdot P \text{ and } X \not\gg Y) \}.$$

Computing $DF(X)$ directly from the definition would require searching much of the dominator tree. The total time

[1] As is usual in code optimization, we avoid undecidability by considering all paths (rather than those that can actually be taken) and by ignoring the actual semantics of operators (other than ϕ). Formally, sameness of values is what has been called transparent equivalence [RWZ88, §8.1].

to compute $DF(X)$ for all nodes X would be quadratic, even when the sets themselves are small. To compute the dominance frontier mapping in time linear in the size of the mapping, we define two intermediate sets DF_{local} and DF_{up} for each node such that the following equation holds:

$$DF(X) = DF_{local}(X) \cup \bigcup_{Z \in Children(X)} DF_{up}(Z). \quad (1)$$

Given any node X, some of the successors of X may contribute to $DF(X)$. This local contribution $DF_{local}(X)$ is defined by

$$DF_{local}(X) = \{ Y \in Succ(X) \mid X \not\gg Y \}.$$

Given any node Z that is not the root **Entry** of the dominator tree, some of the nodes in $DF(Z)$ may contribute to $DF(X)$ for $X = idom(Z)$. The contribution $DF_{up}(Z)$ that Z passes up to $idom(Z)$ is defined by

$$DF_{up}(Z) = \{ Y \in DF(Z) \mid idom(Z) \not\gg Y \}.$$

Lemma 1 *The dominance frontier equation* (1) *is correct.*

Proof. Because dominance is reflexive, $DF_{local}(X) \subseteq DF(X)$. Because dominance is transitive, each child Z of X has $DF_{up}(Z) \subseteq DF(X)$. We must still show that everything in $DF(X)$ has been accounted for. Suppose $Y \in DF(X)$, and let $U \rightarrow Y$ be an edge such that X dominates U but does not strictly dominate Y. If $U = X$, then $Y \in DF_{local}(X)$ and we are done. If $U \neq X$, on the other hand, then there is a child Z of X that dominates U but cannot strictly dominate Y because X does not strictly dominate Y. This implies $Y \in DF_{up}(Z)$. □

The intermediate sets can be computed with simple equality tests as follows.

Lemma 2 *For any node X,*

$$DF_{local}(X) = \{ Y \in Succ(X) \mid idom(Y) \neq X \}.$$

Proof. We assume $Y \in Succ(X)$ and show that

$$(X \gg Y) \iff (idom(Y) = X).$$

The "if" part is true because strict dominance is the transitive closure of immediate dominance. For the "only if" part, suppose X strictly dominates Y, and hence that some child V of X dominates Y. Then V appears on any path from **Entry** to Y that goes to X and then follows the edge $X \rightarrow Y$, so either V dominates X or $V = Y$. But V cannot dominate X, so $V = Y$ and $idom(Y) = idom(V) = X$. □

Lemma 3 *For any node X and any child Z of X in the dominator tree,*

$$DF_{up}(Z) = \{ Y \in DF(Z) \mid idom(Y) \neq X \}.$$

Proof. We assume $Y \in DF(Z)$ and show that

$$(X \gg Y) \iff (idom(Y) = X).$$

The "if" part is true because strict dominance is the transitive closure of immediate dominance. For the "only if" part, suppose X strictly dominates Y, and hence that some child V of X dominates Y. Choose a predecessor U of Y such that Z dominates U. Then V appears on any path from **Entry** to Y that goes to U and then follows the edge $U \rightarrow Y$, so either V dominates U or $V = Y$. If $V = Y$, then $idom(Y) = idom(V) = X$ and we are done. We suppose $V \neq Y$ (and hence that V dominates U) and derive a contradiction. Only one child of X can dominate U, so $V = Z$ and Z dominates Y. This contradicts the hypothesis that $Y \in DF(Z)$. □

These results imply the correctness of the algorithm in Figure 2 for computing dominance frontiers. The `/*local*/` line effectively computes $DF_{local}(X)$ on the fly and uses it in (1) without needing to devote storage to it. The `/*up*/` line is similar for $DF_{up}(Z)$. We traverse the dominator tree bottom-up, visiting each node X only after having visited each of its children. To illustrate the working of this algorithm, we tabulate the results in Figure 3, where CFG comes from the program in Figure 1.

```
for each X in a bottom-up traversal
        of the dominator tree do
   DF(X) ← ∅
   for each Y ∈ Succ(X) do
      if idom(Y) ≠ X
         then DF(X) ← DF(X) ∪ {Y}        /*local*/
   end
   for each Z ∈ Children(X) do
      for each Y ∈ DF(Z) do
         if idom(Y) ≠ X
            then DF(X) ← DF(X) ∪ {Y}     /*up*/
      end
   end
end
```

Figure 2. Calculation of DF

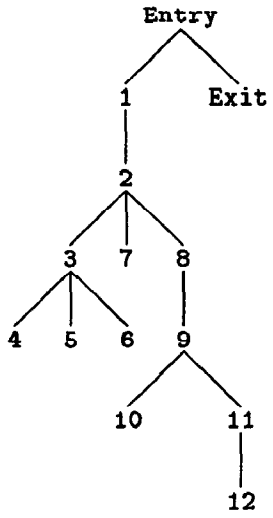
Theorem 1 *The algorithm in Figure 2 is correct.*

Proof. Direct from the preceding lemmas. □

Consider a CFG with N nodes and E edges. The dominance frontier algorithm eventually examines all edges of CFG in computing DF_{local}. Computing DF_{up} will (at worst) require propagating N nodes through the dominator tree. The time required to compute DF_{up} and therefore DF is proportional to the size of DF_{up}. Since the dominator tree has $N - 1$ edges, the computation takes time $O(N^2)$. Thus, the overall algorithm has worst-case complexity $O(E + N^2)$. However, Section 6 shows that the size of the mapping DF is usually linear in practice. We have implemented this algorithm and have observed that it is faster than the standard data flow computations in the PTRAN compiler [ABC+88].

3.2 Using Dominance Frontiers to Find Where ϕ-Functions Are Needed

We start by restating more formally the nonrecursive characterization of where the ϕ-functions should be located. Given a set S of CFG nodes, the set $J(S)$ of *join* nodes



Node	*succ*	*idom*	DF_{local}	DF_{up}	*DF*
Entry	Exit, 1				
1	2	Entry			Exit
2	3, 7	1		Exit	Exit, 2
3	4, 5	2			8
4	6	3	6		6
5	6	3	6		6
6	8	3	8	8	8
7	8	2	8		8
8	9	2		Exit, 2	Exit, 2
9	10, 11	8		Exit, 2	Exit, 2, 9
10	11	9	11		11
11	9, 12	9	9	Exit, 2, 9	Exit, 2, 9
12	Exit, 2	11	Exit, 2	Exit, 2	Exit, 2
Exit		Entry			

Figure 3. Example Dominator Tree and Dominance Frontier Computation

is defined to be the set of all nodes Z such that there are two nonnull CFG paths that start at two distinct nodes in $\mathcal{S}$ and have Z as the first node in common. The *iterated* join $J^+(\mathcal{S})$ is the limit of the increasing sequence of sets of nodes

$$\begin{aligned} J_1 &= J(\mathcal{S}); \\ J_{i+1} &= J(\mathcal{S} \cup J_i). \end{aligned}$$

In particular, if $\mathcal{S}$ happens to be the set of assignment nodes for a variable V, then $J^+(\mathcal{S})$ is the set of ϕ-function nodes for V.

The join and iterated join operations map sets of nodes to sets of nodes. We extend the dominance frontier mapping from nodes to sets of nodes in the natural way:

$$DF(\mathcal{S}) = \bigcup_{X \in \mathcal{S}} DF(X).$$

As with join, the *iterated* dominance frontier $DF^+(\mathcal{S})$ is the limit of the increasing sequence of sets of nodes

$$\begin{aligned} DF_1 &= DF(\mathcal{S}); \\ DF_{i+1} &= DF(\mathcal{S} \cup DF_i). \end{aligned}$$

The actual computation of $DF^+(\mathcal{S})$ is performed by an efficient worklist algorithm; the formulation here is convenient for relating iterated dominance frontiers to iterated joins. If the set $\mathcal{S}$ happens to be the set of assignment nodes for a variable V, then we will show that

$$J^+(\mathcal{S}) = DF^+(\mathcal{S})$$

and hence that the location of the ϕ-functions for V can be computed by the worklist algorithm for computing $DF^+(\mathcal{S})$ that is given in Section 4.

The following lemmas do most of the work by relating dominance frontiers to joins.

Lemma 4 *For any nonnull path $p : X \xrightarrow{+} Z$ in CFG, there is a node $X' \in \{X\} \cup DF^+(\{X\})$ on p that dominates Z.*

Proof. Let X' be the last node in $\{X\} \cup DF^+(\{X\})$ on p. We suppose X' does not dominate Z and derive a contradiction. Because dominance is reflexive, $X' \neq Z$ and there is a first node Y after X' on p such that X' does not dominate Y. The predecessor of Y on p is dominated by X', so

$$Y \in DF(X') \subseteq DF(\{X\} \cup DF^+(\{X\})) = DF^+(\{X\}),$$

which contradicts the choice of X'. □

Lemma 5 *Let $X \neq Y$ be two nodes in CFG and suppose that nonnull paths $p : X \xrightarrow{+} Z$ and $q : Y \xrightarrow{+} Z$ in CFG have Z as the first node in common. Then $Z \in DF^+(\{X\}) \cup DF^+(\{Y\})$.*

Proof. We start by proving the lemma under the added hypotheses that $Z \neq X$ and $Z \neq Y$. Let X' be from Lemma 4 for the path p. Let Y' be from Lemma 4 for the path q. Let p' and q' be the corresponding final segments of p and q. Any path from **Entry** to Y' and then to Z along q' must include X', but Z is the only node in q' that lies on p. Therefore

$$X' \gtrdot Y' \quad \textbf{or} \quad X' = Z.$$

Similarly,

$$Y' \gtrdot X' \quad \textbf{or} \quad Y' = Z.$$

Because X' and Y' cannot dominate each other, one of them must be Z. We may assume $X' = Z$ and hence $Z \in \{X\} \cup DF^+(\{X\})$. But $Z \neq X$, so $Z \in DF^+(\{X\}) \subseteq DF^+(\{X\}) \cup DF^+(\{Y\})$.

Now suppose that Z is one of the origin nodes X, Y. We may assume $Z = X$, which implies $Z \neq Y$. If the first edge along p is $Z \to Z$, then $Z \in DF(\{Z\}) \subseteq DF^+(\{X\})$ and we are done. Suppose instead that the first edge along p is $Z \to A \neq Z$. We may apply the previous paragraph to the paths a and q, where a is the rest of p after $Z \to A$. We get A' and Y' like X' and Y' from the previous paragraph, such that

$$A' \gtrdot Y' \quad \textbf{or} \quad A' = Z; \tag{2}$$

$$Y' \gtrdot A' \quad \textbf{or} \quad Y' = Z. \tag{3}$$

If $Y' = Z$, then we use $Z \neq Y$ to continue as before and derive $Z \in DF^+(\{Y\})$. Suppose instead that $Y' \neq Z$,

so (3) implies $Y' \gg A'$ and then (2) implies $A' = Z$. But $A' \in \{A\} \cup DF^+(\{A\})$, so there is a sequence $A = A_0, ..., A_L = Z$ of nodes on a, such that each i has $A_{i+1} \in DF(A_i)$ with A_i dominating any node after A_i but before A_{i+1} on a. By induction on i, we can also show that

$$A_i \in DF^+(\{Z\}) \quad \textbf{or} \quad Z \gg A_i \tag{4}$$

In particular, for $i = L - 1$, we find that A_i dominates the predecessor of Z on a and satisfies (4). In both cases, $Z \in DF^+(\{Z\}) = DF^+(\{X\})$. □

Lemma 6 *For any set $\mathcal{S}$ of CFG nodes, $J(\mathcal{S}) \subseteq DF^+(\mathcal{S})$.*

Proof. We apply Lemma 5. □

Lemma 7 *For any set $\mathcal{S}$ of CFG nodes such that* **Entry** $\in \mathcal{S}$, $DF(\mathcal{S}) \subseteq J(\mathcal{S})$.

Proof. Consider any $X \in \mathcal{S}$ and any $Y \in DF(X)$. There is a path from X to Y where all nodes before Y are dominated by X. There is also a path from **Entry** to Y where all of the nodes are *not* dominated by X. The first node common to both paths is therefore Y. □

Theorem 2 *The set of nodes that need ϕ-functions for any variable V is the iterated dominance frontier $DF^+(\mathcal{S})$, where $\mathcal{S}$ is the set of assignments for V.*

Proof. The set of nodes that need ϕ-functions for V is $J^+(\mathcal{S})$. By Lemma 6 and induction on i in the definition of J^+, we can show that

$$J^+(\mathcal{S}) \subseteq DF^+(\mathcal{S}).$$

The induction step is as follows:

$$\begin{array}{rcccc} J_{i+1} & = & J(\mathcal{S} \cup J_i) & \subseteq & J(\mathcal{S} \cup DF^+(\mathcal{S})) \\ & \subseteq & DF^+(\mathcal{S} \cup DF^+(\mathcal{S})) & = & DF^+(\mathcal{S}). \end{array}$$

The node **Entry** is in $\mathcal{S}$, so Lemma 7 and another induction yield

$$DF^+(\mathcal{S}) \subseteq J^+(\mathcal{S}). \quad \square$$

4 Construction of Minimal SSA Form

The algorithm in Figure 4 inserts trivial ϕ-functions. The outer loop of this algorithm is performed once for each variable in the program. Several data structures are used:

- W is the worklist of CFG nodes that are being processed. In each iteration of this algorithm, W is initialized to the set $\mathcal{A}(V)$ of nodes that contain assignments to V. Each iteration terminates when the worklist becomes empty.
- $Work(*)$ is an array of flags, one flag for each node, where $Work(X)$ is 1 if X has ever been added to W. Each node may be added to W only once during each iteration.
- $DomFronPlus(*)$ is an array of flags, one for each node, where $DomFronPlus(X)$ is 1 if a ϕ-function for V has already been inserted at X. At the end of each iteration, the nodes X with $DomFronPlus(X) = 1$ are exactly the nodes in the iterated dominance frontier of $\mathcal{A}(V)$.

```
for each variable V do
    DomFronPlus(*) ← 0
    Work(*) ← 0
    W ← ∅
    for each X ∈ A(V) do
        Work(X) ← 1
        W ← W ∪ {X}
    end
    while W ≠ ∅ do
        take X from W
        for each Y ∈ DF(X) do
            if DomFronPlus(Y) = 0
                then do
                    add φ-function for V to Y
                    DomFronPlus(Y) ← 1
                    if Work(Y) = 0
                        then do
                            Work(Y) ← 1
                            W ← W ∪ {Y}
                        end
                end
        end
    end
end
```

Figure 4. Placement of ϕ-functions

The time required to process a single variable in Figure 4 is proportional to the total number of ordinary assignments and ϕ-functions plus the total number of relevant dominance frontier relationships.

The algorithm in Figure 5 renames all mentions of variables while visiting the nodes of the dominator tree in a depth-first search. New names denoted V_i, where i is an integer, are generated for each variable V. The search starts at **Entry**, where the entrance value of V is represented by an assignment with an empty right-hand side. After renaming each V to V_0 here, the search moves on to other nodes. The visit to a node processes the statements associated with the node in sequential order, starting with any ϕ-functions that may have been inserted. The processing of a statement requires work for only those variables actually mentioned in the statement. In contrast with Figure 4, we only need a loop over all variables when we initialize two arrays among the following data structures:

- $S(*)$ is an array of stacks, one stack for each variable V. The stacks can hold integers. The integer i at the top of $S(V)$ is used to construct the name V_i that should replace a use of V.
- $C(*)$ is an array of integers, one for each variable V. The counter value $C(V)$ tells how many assignments to V have been processed.
- $WhichPred(Y, X)$ is an integer telling which predecessor of Y in CFG is X. The j-th operand of a ϕ-function in Y corresponds to the j-th predecessor of Y from the listing of the inedges of Y.

- Each assignment statement A has the form

$$LHS(A) \leftarrow RHS(A)$$

where the right-hand side $RHS(A)$ is an expression and the left-hand side $LHS(A)$ is the target variable V. After renaming has replaced V by V_i as the target, the old target V is still remembered as $oldLHS(A)$.

```
C(*) ← 0
S(*) ← EmptyStack
call SEARCH(Entry)

SEARCH(X) :
   for each assignment A in X do
      if A is an ordinary assignment
         then do
            for each variable V used in RHS(A)
               replace use of V by use of V_i
                     where i = Top(S(V))
         end
      let V be LHS(A) in
         i ← C(V)
         replace V by V_i as LHS(A)
         push i onto S(V)
         C(V) ← i + 1
      end
   end
   for each Y ∈ Succ(X) do
      j ← WhichPred(Y, X)
      for each φ-function F in Y do
         replace the j-th operand V of F by
            V_i where i = Top(S(V))
      end
   end
   for each Y ∈ Children(X) do
      call SEARCH(Y)
   end
   for each assignment A in X do
      pop S(oldLHS(A))
   end
end SEARCH
```

Figure 5. Construction of SSA Form

The next lemma shows that it makes sense to speak of "the" assignment to a new name for a variable in the transformed program.

Lemma 8 *Each new name V_i mentioned in the transformed program is the target of exactly one assignment.*

Proof. Because the counter $C(V)$ is incremented after processing each assignment to V, there can be at most one assignment to V_i. Because any use of V_i has $i = Top(S(V))$ at the time V is replaced by V_i, there is at least one assignment to V_i. □

With each variable V and CFG node X, we can associate the name $TopAfter(V, X)$ of V determined by the top of the stack $S(V)$ at the end of the first loop in `SEARCH(X)`. Specifically,

$$TopAfter(V, X) = V_i \quad \text{where} \quad i = Top(S(V)).$$

If a child Y of X does *not* have a ϕ-function for V, then the first use (if any) of V in Y is replaced by a use of $TopAfter(V, X)$. Thus Y inherits a name for V from its parent $X = idom(Y)$. On the other hand, any predecessor P of Y in CFG determines a name $TopAfter(V, P)$ with more obvious relevance to the question of whether the transformed program is equivalent to the original. Fortunately, there is no conflict between candidate names.

Lemma 9 *For any variable V and any CFG edge $P \rightarrow Y$ such that Y does* **not** *have a ϕ-function for V,*

$$TopAfter(V, P) = TopAfter(V, idom(Y)). \qquad (5)$$

Proof. We may assume $P \neq idom(Y)$. Because Y does not have a ϕ-function for V, if a node X has $Y \in DF(X)$, then X does not assign to a name of V. We use this fact twice below.

By Lemma 2, $Y \in DF_{local}(P) \subseteq DF(P)$ and P does not assign to a name of V. Let U be the first node in the sequence $idom(P), idom(idom(P)), \ldots$ that assigns to a name of V. Then

$$TopAfter(V, P) = TopAfter(V, U). \qquad (6)$$

Because U assigns to a name of V, $Y \notin DF(U)$. But U dominates a predecessor of Y, so U strictly dominates Y. For any X with $U \gg X \gtrdot idom(Y)$, we get $X \gtrdot P$ because $X \gg Y$. By the choice of U, $U \gg X \gtrdot P$ implies that X does not assign to a name of V. Therefore

$$TopAfter(V, U) = TopAfter(V, idom(Y))$$

and (5) follows from (6). □

Lemma 10 *Consider any control flow path in the transformed program and the same path in the original program. For any variable V and any edge $X \rightarrow Y$ encountered along the path, the value of V that flows from X to Y in the original program is equal to the value of $TopAfter(V, X)$ that flows from X to Y in the transformed program.*

Proof. We use induction along the path, starting with $V = V_0$ after each entering value assignment in `Entry`. To continue the induction, consider any consecutive edges $X \rightarrow Y \rightarrow Z$ along the path, and let $j = WhichPred(Y, X)$. We assume $V = TopAfter(V, X)$ along the first edge and show that $V = TopAfter(V, Y)$ along the second edge.

If V has a ϕ-function in Y, then the j-th operand of ϕ is $TopAfter(V, X)$ and the target of the ϕ-function in the transformed program does receive the value of V. If V does not have a ϕ-function in Y, then Lemma 9 is applicable. In both cases, the correct value of V is already available at the start of the basic block of ordinary code that defines Y. Within the basic block, the flow of control in the transformed program is the same as the sequence of processing statements in the first loop in `SEARCH`. For each variable V, we do have $V = TopAfter(V, Y)$ along the second edge. □

Theorem 3 *Any program can be put into minimal SSA form by applying the algorithm in Figure 4 and then the algorithm in Figure 5.*

Proof. Figure 4 places the ϕ-functions for V at the nodes in the iterated dominance frontier $DF^+(\mathcal{S})$, where $\mathcal{S}$ is the set of assignments to V in the original program. By Theorem 2, $DF^+(\mathcal{S})$ is the set of nodes that need ϕ-functions for V, so we have obtained Condition 1 in the definition of SSA form with the fewest possible ϕ-functions.

We must still show that renaming is done correctly by Figure 5. Condition 2 in the definition of SSA form follows from Lemma 8. Condition 3 in the definition of SSA form follows from Lemma 10. □

5 Construction of Control Dependences

In this section we show that control dependences [FOW87] are essentially the dominance frontiers in the *reverse* graph of the control flow graph. Let X and Y be nodes in CFG. If X appears on every path from Y to **Exit**, then X *postdominates* Y.[2] Like the dominator relation, the postdominator relation is reflexive and transitive. If X postdominates Y but $X \neq Y$, then X *strictly* postdominates Y. The *immediate postdominator* of Y is the closest strict postdominator of Y on any path from Y to **Exit**. In a *postdominator tree*, the children of a node X are all immediately postdominated by X.

A CFG node Y is *control dependent* on a CFG node X if both of the following hold:

1. There is a nonnull path $p : X \xrightarrow{+} Y$ such that Y postdominates every node after X on p.
2. The node Y does not strictly postdominate the node X.

The definition of control dependence we use here can be shown to be equivalent to the original definition [FOW87] using elementary first-order logic.

Lemma 11 *Let X and Y be CFG nodes. Then Y postdominates a successor of X if and only if there is a nonnull path $p : X \xrightarrow{+} Y$ such that Y postdominates every node after X on p.*

Proof. Suppose that Y postdominates a successor U of X. Choose any path q from U to **Exit**. Then Y appears on q. Let r be the initial segment of q that reaches the first appearance of Y on q. For any node V on r we can get from U to **Exit** by following r to V and then taking any path from V to **Exit**. Because Y postdominates U but does not appear before the end of r, Y must postdominate V as well. Let p be the path that starts with the edge $X \rightarrow U$ and then proceeds along r. Then $p : X \xrightarrow{+} Y$ and Y postdominates every node after X on p.

Conversely, given a path p with these properties, let U be the first node after X on p. Then U is a successor of X and Y postdominates U. □

The *reverse control flow graph* $RCFG$ has the same nodes as the given control flow graph CFG, but has an edge $Y \rightarrow X$ for each edge $X \rightarrow Y$ in CFG. The roles of **Entry** and **Exit** are also reversed. The postdominator relation on CFG is the dominator relation on $RCFG$.

Corollary 1 *Let X and Y be nodes in CFG. Then Y is control dependent on X in CFG if and only if $X \in DF(Y)$ in $RCFG$.*

Proof. Using Lemma 11 to simplify the first condition in the definition of control dependence, we find that Y is control dependent on X if and only if Y postdominates a successor of X but does not strictly postdominate X. In $RCFG$, this says that Y dominates a predecessor of X but does not strictly dominate X, i.e., $X \in DF(Y)$. □

Figure 6 applies this result to compute control dependences. By applying the algorithm in Figure 6 to the control flow graph in Figure 1, we obtain the control dependences in Figure 7. We remark that the edge from **Entry** to **Exit** was added to CFG so that the control dependence relation, viewed as a graph, would be rooted at **Entry**.

```
build RCFG
build dominator tree for RCFG
apply the algorithm in Figure 2 to find the
      dominance frontier mapping RDF for RCFG

for each node X do CD(X) ← ∅ end
for each node Y do
   for each X ∈ RDF(Y) do
      CD(X) ← CD(X) ∪ {Y}
   end
end
```

Figure 6. Algorithm for Computing the Set $CD(X)$ of Nodes Control Dependent on X

Node	CD(Node)
Entry	1,2,8,9,11,12
1	
2	3,6,7
3	4,5
4	
5	
6	
7	
8	
9	10
10	
11	9,11
12	2,8,9,11

Figure 7. Control Dependences of Program in Figure 1

We now compare previous work on computing control dependences with our new algorithm. Control dependences

[2] The postdominance relation in [FOW87] is irreflexive, while the definition we use here is reflexive. The two relations are identical on pairs of distinct elements. We choose the reflexive definition here to make postdominance the dual relation of the dominance relation.

are computed in [FOW87], but that algorithm as stated uses quadratic space for intermediate computations and multiple passes over the dominator tree. An earlier version [CF87b] of our algorithm was formulated for control dependeces along, with no reference to other uses of dominance frontiers. Control dependences are also computed in [HPR88], but only for programs restricted to the classical structured programming constructs.

6 Analysis and Measurements

The number of nodes that contain ϕ-functions for a variable V is a function of the program control flow structure and the assignments for V. Program structure alone determines dominance frontiers and the number of control dependence arcs. It is possible that dominance frontiers may be larger than necessary for computing ϕ-function locations for some programs, since the actual assignments are not taken into account. In this section, we prove that the size of the dominance frontiers is linear in the size of the program when control flow branching is restricted to **if-then-else** constructs and **while-do** loops.[3] Such programs can be described by the grammar given in Figure 8. We also give experimental results that suggest that the behavior is linear for actual programs.

```
<program>   ::= <statement>                          (1)
<statement> ::= <statement><statement>               (2)
<statement> ::= if <predicate>                       (3)
                   then <statement>
                   else <statement>
<statement> ::= while <predicate>                    (4)
                   do <statement>
<statement> ::= <variable> ← <expression>            (5)
```

Figure 8. Grammar for Control Structures

Theorem 4 *For programs comprised of straight-line code,* **if-then-else**, *and* **while-do** *constructs, the dominance frontier of any CFG node contains at most two nodes.*

Proof. Consider a top-down parse of a program using the grammar shown in Figure 8. Initially, we have a single **<program>** node in the parse tree and a control flow graph CFG with two nodes and one edge: **Entry** $\rightarrow$ **Exit**. The initial dominance frontiers are $DF(\texttt{Entry}) = \emptyset = DF(\texttt{Exit})$. For each production, we consider the associated changes to CFG and to the dominance frontiers of nodes. We show that each CFG node S corresponding to an unexpanded **<statement>** symbol has at most one node in its dominance frontier. When a production expands a nonterminal parse tree node, a new subgraph is inserted into CFG in place of S. In this new subgraph, a CFG node T that corresponds to a terminal symbol has at most two nodes in its dominance frontier.

(1) This production adds a CFG node S and edges **Entry** $\rightarrow S \rightarrow$ **Exit**, yielding $DF(S) = \{\texttt{Exit}\}$.

(2) When this production is applied, a CFG node S is replaced by two nodes S_1 and S_2. Edges previously entering and leaving S now enter S_1 and leave S_2. A single edge is inserted from S_1 to S_2. Although the control flow graph has changed, consider how this production affects the dominator tree. Nodes S_1 and S_2 dominate all nodes that were dominated by S. Additionally, S_1 dominates S_2. Thus, we have $DF(S_1) = DF(S) = DF(S_2)$.

(3) When this production is applied, a CFG node S is replaced by nodes T_{if}, S_{then}, S_{else}, and T_{endif}. Edges previously entering and leaving S now enter T_{if} and leave T_{endif}. Edges are inserted from T_{if} to both S_{then} and S_{else}; edges are also inserted from S_{then} and S_{else} to T_{endif}. In the dominator tree, T_{if} and T_{endif} both dominate all nodes that were dominated by S. Additionally, T_{if} dominates S_{then} and S_{else}. By the argument made for production (2), we have $DF(T_{if}) = DF(S) = DF(T_{endif})$. Now consider nodes S_{then} and S_{else}. From the definition of dominance frontier, we obtain $DF(S_{then}) = DF(S_{else}) = \{T_{endif}\}$.

(4) When this production is applied, a CFG node S is replaced by nodes T_{while} and S_{do}. All edges previously associated with node S are now associated with node T_{while}. Edges are inserted from T_{while} to S_{do} and from S_{do} to T_{while}. Node T_{while} dominates all nodes that were dominated by node S. Additionally, T_{while} dominates S_{do}. Thus, we have $DF(T_{while}) = DF(S) \cup \{T_{while}\}$ and $DF(S_{do}) = \{T_{while}\}$.

(5) After application of this production, the new control flow graph is isomorphic to the old graph.

Thus, each production causes a CFG node to be replaced by a new subgraph. In the new subgraph, only a T_{while} node may have more than one node in its dominance frontier. Such a node will never be expanded to a subgraph, so each node in the final CFG has at most two nodes in its dominance frontier. □

Corollary 2 *For programs comprised of straight-line code,* **if-then-else**, *and* **while-do** *constructs, every node is control dependent on at most two nodes.*

Proof. Consider a program P composed of the allowed constructs, and its associated control flow graph CFG. The reverse control flow graph $RCFG$ is itself a structured control flow graph for some program P'. For all Y in $RCFG$, $DF(Y)$ contains at most two nodes by Theorem 4. By Corollary 1, Y is then control dependent on at most two nodes. □

Unfortunately, these linearity results do not hold for all program structures. In particular, consider the nest of **repeat-until** loops illustrated in Figure 1. For each loop, the dominance frontier of a node in that loop includes each of the entrances to surrounding loops. For n nested loops, this leads to a dominance frontier mapping whose total size is $\Theta(n^2)$, yet each variable needs at most $O(n)$ ϕ-functions. Most of the dominance frontier mapping is not actually used in placing ϕ-functions, so there is a concern that the

[3] We assume expressions and predicates perform no internal branching.

computation of dominance frontiers might take excessive time with respect to the resulting number of actual ϕ-functions. We therefore wish to measure the number of dominance frontier nodes as a function of program size over a diverse set of programs.

We implemented our dominance frontier algorithm and executed it against the FORTRAN routines in EISPACK [SBD+76]. We chose such routines because they contain irreducible intervals and other unstructured constructs. We implemented our algorithm in the PTRAN system, which already offered the required data flow and control flow analysis [ABC+88]. For the 61 programs we tested, the ratio of dominance frontier arcs to program size varied from 1.3 to 2.4. As the plot in Figure 9 shows, the size of the dominance frontier mapping appears to vary linearly with program size.

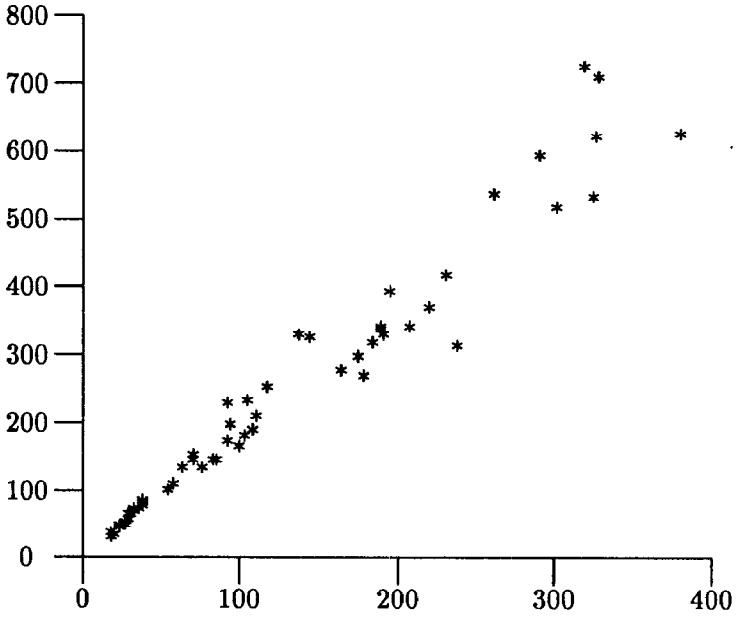


Figure 9. Size of Dominance Frontier Mapping vs. Number of Program Statements

Our next concern is that the number of ϕ-functions might be nonlinear in the size of the original program. For the programs we tested, the plot in Figure 10 confirms linear behavior for ϕ-functions with respect to program size.

The remaining concern is that the control dependence graph might be nonlinear in the size of the original program. For the programs we tested, the plot in Figure 11 confirms linear behavior for control dependence arcs with respect to program size.

Comparing Figures 9 and 11, we find that the same set of programs produced more dominance frontier relationships than control dependence relationships. Informally, dominance frontier relationships are due to joins in the control flow graph, but control dependence relationships result from forks in the control flow graph. The prevalent style of coding in our test suite resulted in control flow graphs with some nodes of high indegree and while most nodes have a low outdegree.

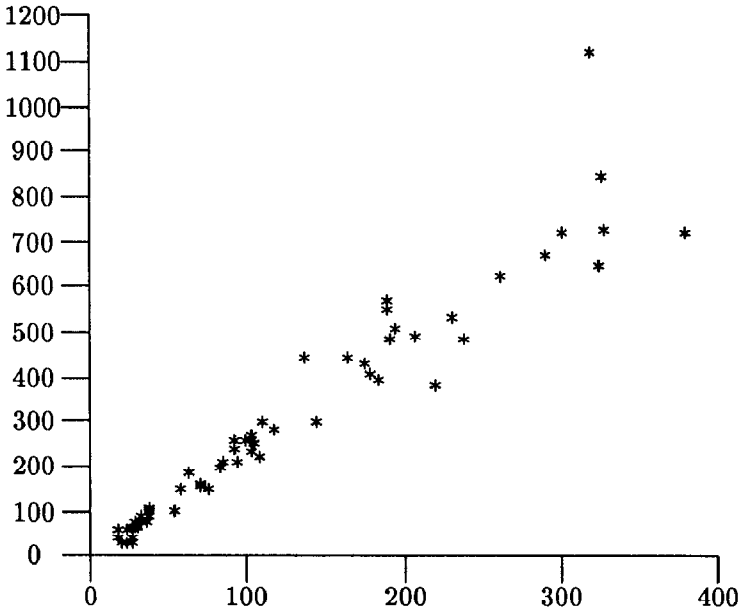


Figure 10. Number of ϕ-functions vs. Number of Program Statements

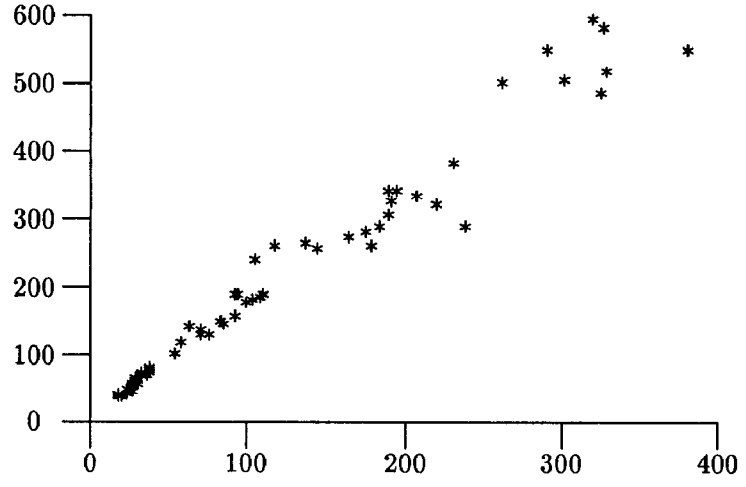


Figure 11. Size of Control Dependence Graph vs. Number of Program Statements

7 Conclusion

Recent previous work has shown that SSA form and control dependences can support powerful code optimizations algorithms that are highly efficient in terms of time and space bounds based on the size of the program *after* translation to the forms. We have shown that this translation can be performed efficiently, that it leads to only a moderate increase in program size, and that applying the early steps in the SSA translation to the reverse graph is an efficient way to compute control dependences. This is strong evidence that SSA form and control dependences form a *practical* basis for optimization.

Acknowledgements

We would like to thank Fran Allen, Julian Padget, and Tom Reps for helpful comments.

References

[ABC+88] F. E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, October 1988.

[AJ88] J. R. Allen and S. Johnson. Compiling c for vectorization, parallelization and inline expansion. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, 23(7):241–249, June 1988.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, January 1988.

[CF87a] R. Cytron and J. Ferrante. An improved control dependence algorithm. Technical Report RC 13291, IBM, 1987.

[CF87b] R. Cytron and J. Ferrante. What's in a name? *Proc. 1987 International Conf. on Parallel Processing*, pages 19–27, August 1987.

[CLZ86] R. Cytron, A. Lowry, and F. K. Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. Thirteenth ACM Symp. on Principles of Programming Languages*, pages 70–85, January 1986.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Har85] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. *Proc. Seventeenth ACM Symp. on Theory of Computing*, pages 185–194, May 1985.

[HPR88] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 133–145, January 1988.

[LT79] T. Lengaurer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. on Programming Languages and Systems*, 1(1):121–141, July 1979.

[RL86] J. H. Reif and H. R. Lewis. Efficent symbolic analysis of programs. *J. Computer and System Sciences*, 32(3):280–313, June 1986.

[RT82] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost linear time. *SIAM J. Computing*, 11(1):81–93, February 1982.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, January 1988.

[SBD+76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – Eispack Guide*. Springer-Verlag, 1976.

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

[Tar74] R. E. Tarjan. Finding dominators in directed graphs. *SIAM J. Computing*, 3(1):62–89, 1974.

[WZ85] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *Conf. Rec. Twelfth ACM Symp. on Principles of Programming Languages*, pages 291–299, January 1985.

[WZ88] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. Technical Report CS-88-02, Dept. of Computer Science, Brown U., February 1988.