Efficiently Computing Static Single Assignment Form and the Control Dependence Graph

RON CYTRON (IBM Research Division)
JEANNE FERRANTE (IBM Research Division)
BARRY K. ROSEN (IBM Research Division)
MARK N. WEGMAN (IBM Research Division)
F. KENNETH ZADECK (Brown University)

March 7, 1991

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 that advanced optimization features become undesirable. Recently, static single assignment form and the control dependence graph 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. We present new algorithms that efficiently compute these data structures for arbitrary control flow graphs. The algorithms use *dominance frontiers*, a new concept that may have other applications. We also give analytical and experimental evidence that all of these data structures 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.

Categories and Subject Descriptors:

D3.3[**Programming Languages**]: Language Constructs – *control structures; data types and structures; procedures, functions and subroutines;*

D3.4[**Programming Languages**]: Processors – *compilers; optimization;*

I1.2[**Algebraic Manipulation**]: Algorithms - *analysis of algorithms*

I2.2[**Computing Methodologies**]: Automatic Programming – *program transformation.*

General Terms: Optimizing Compilers

Additional Key Words and Phrases: Control Flow Graph, Control Dependence, Dominator, Def-Use Chain

A preliminary version of this paper, "An Efficient Method of Computing Static Single Assignment Form," appeared in Conf. Rec. Sixteenth ACM Symp. on Principles of Programming Languages (January 1989).

F. K. Zadeck's work on this paper was partially supported by IBM Corp. and the Office of Naval Research and the Defense Advanced Research Projects Agency under Contract N00014-83-K-0146 and ARPA Order 6320, Amendment 1.

Authors' current addresses are: *Ron Cytron*, Computer Sciences Dept., IBM Research Division, P.O. Box 704, Yorktown Heights, NY 10598; *Jeanne Ferrante*, Computer Sciences Dept., IBM Research Division, P.O. Box 704, Yorktown Heights, NY 10598; *Barry K. Rosen*, Mathematical Sciences Dept., IBM Research Division, P.O. Box 218, Yorktown Heights, NY 10598; *Mark N. Wegman*, Computer Sciences Dept., IBM Research Division, P.O. Box 704, Yorktown Heights, NY 10598; *F. Kenneth Zadeck*, Computer Science Dept., P.O. Box 1910, Brown University, Providence, RI 02912.

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 *(SSA)* form [5, 43] and the control dependence graph [24] 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 [4]. We present new algorithms that efficiently compute these data structures for arbitrary control flow graphs. The algorithms use *dominance frontiers*, a new concept that may have other applications. We also give analytical and experimental evidence that the sum of the sizes of all the dominance frontiers is usually linear in the size of the original program. This paper thus presents strong evidence that SSA form and control dependences can be of practical use in optimization.

Figure 1 illustrates the role of static single assignment form in a compiler. The intermediate code is put into SSA form, optimized in various ways, and then translated back out of SSA form. Optimizations that can benefit from using SSA form include code motion [22] and elimination of partial redundancies [43], as well as the constant propagation discussed later in this section.

$$
\begin{tikzcd}
\begin{array}{c}\texttt{Original}\\ \texttt{Intermediate Code}\end{array} \arrow[d] & & & & \begin{array}{c}\texttt{Optimized}\\ \texttt{Intermediate Code}\end{array} \\
P_0 \arrow[r] & P_1 \arrow[r] & P_2 \arrow[r] & \cdots & P_n \arrow[u]
\end{tikzcd}
$$

Figure 1. Vertical arrows represent translation to/from static single assignment form.
Horizontal arrows represent optimizations.

Variants of SSA form have been used for detecting program equivalence [5, 52] and for increasing parallelism in imperative programs [21]. The representation of simple data flow information (def-use chains) may be more compact through SSA form. If a variable has D definitions and U uses, then there can be $D \times U$ 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 [40]. Moreover, the def-use information encoded in SSA form can be updated easily when optimizations are applied. This is important for a constant propagation algorithm that deletes branches to code proven at compile time to be unexecutable [50]. Specifically, the def-use information is just a list, for each variable, of the places in the program text that use that variable. Every use of `V` is indeed a use of the value provided by the (unique) assignment to `V`.

To see the intuition behind SSA form, it is helpful to begin with straight-line code. Each assignment to a variable is given a unique name (shown as a subscript in Figure 2) and all of the uses reached by that assignment are renamed to match the assignment's new name.

$\mathtt{V} \leftarrow 4$	$\mathtt{V}_1 \leftarrow 4$
$\quad \leftarrow \mathtt{V} + 5$	$\quad \leftarrow \mathtt{V}_1 + 5$
$\mathtt{V} \leftarrow 6$	$\mathtt{V}_2 \leftarrow 6$
$\quad \leftarrow \mathtt{V} + 7$	$\quad \leftarrow \mathtt{V}_2 + 7$

Figure 2. Straight-line code and its single assignment version.

Most programs, however, have branch and join nodes. At the join nodes, we add a special form of assignment called a *ϕ-function*. In Figure 3, the operands to the ϕ-function indicate which assignments to `V` reach the join point. Subsequent uses of `V` become uses of $\mathtt{V}_3$. The old variable `V` is thus replaced by new variables $\mathtt{V}_1, \mathtt{V}_2, \mathtt{V}_3, \ldots$ and each use of $\mathtt{V}_\mathtt{i}$ is reached by just one assignment to $\mathtt{V}_\mathtt{i}$. Indeed, there is only one assignment to $\mathtt{V}_\mathtt{i}$ in the entire program. This simplifies the record keeping for several optimizations. An especially clear example is constant propagation based on SSA form [50], so the next subsection sketches this application.

```
if P                              if P
   then V ← 4                        then V1 ← 4
   else V ← 6                        else V2 ← 6
                                  V3 ← φ(V1, V2)
/* Use V several times. */        /* Use V3 several times. */
```

Figure 3. An `if-then-else` and its single assignment version.

1.1 Constant Propagation

Figure 4 is a more elaborate version of Figure 3. The `else` branch includes a test of `Q`, and propagating the constant `true` to this use of `Q` tells a compiler that the `else` branch never falls through to the join point. If `Q` is the constant `true`, then all uses of `V` after the join point are really uses of the constant `4`. Such possibilities can be taken into account without SSA form, but the processing is either costly [48] or difficult to understand [49]. With SSA form, the algorithm is fast and simple.

Initially, the algorithm assumes that each edge is *unexecutable* (i.e., never followed at run time) and that each variable is constant with an as-yet unknown value (denoted $\top$). Worklists are initialized appropriately, and the assumptions are corrected until they stabilize. Suppose the algorithm finds that variable `P` is *not* constant (denoted $\perp$) and hence that either branch of the outer conditional may be taken. The outedges of the test of `P` are marked executable and the statements they reach are processed. When $V_1 \leftarrow 4$ is processed, the assumption about V_1 is changed from $\top$ to `4` and all uses of V_1 are notified that they are uses of `4`. (Each use of V_1 is indeed a use of this value, thanks to the single assignment property.) In particular, the ϕ-function combines `4` with V_2. The second operand of the ϕ-function, however, is associated with the inedge of the join point that corresponds to falling through the `else` branch. So long as this edge is considered unexecutable, the algorithm uses $\top$ for the second operand, no matter what is currently assumed about V_2. Combining `4` with $\top$ yields `4`, so uses of V_3 are still tentatively assumed to be uses of `4`. Eventually, the assumption about `Q` may change from $\top$ to a known constant or to $\perp$. A change to either `false` or $\perp$ would lead to the discovery that the second inedge of the join point can be followed, and then the `6` at V_2 combines with the `4` at V_1 to yield $\perp$ at V_3. A change to `true` would have no effect on the assumption at V_3. Traditional constant propagation algorithms, on the other hand, would see that assignments of two different

<pre>if P
then do V ← 4
     end
else do V ← 6
        if Q then return
     end
 	
... ← V + 5</pre>
	<pre>if P
	then do V₁ ← 4
	     end
	else do V₂ ← 6
	        if Q then return
	     end
	V₃ ← φ(V₁, V₂)
	... ← V₃ + 5</pre>

Figure 4. An example of constant propagation.

constants to V seem to "reach" all uses after the join point. They would thus decide that V is not constant at these uses.

The algorithm just sketched is linear in the size of the SSA program it sees [50], but this size might be nonlinear in the size of the original program. In particular, it would be safe but inefficient to place a ϕ-function for every variable at every join point. This paper shows how to obtain SSA form efficiently. When ϕ-functions are placed carefully, nonlinear behavior is still possible but is unlikely in practice. The size of the SSA program is typically linear in the size of the original program; the time to do the work is essentially linear in the SSA size.

1.2 Where to Place ϕ-Functions

At first glance, careful placement might seem to require enumerating pairs of assignment statements for each variable. Checking whether there are *two* assignments to V that reach a common point might seem to be intrinsically nonlinear. In fact, however, it is enough to look at the *dominance frontier* of each node in the control flow graph. Leaving the technicalities to later sections, we sketch the method here.

Suppose that a variable V has just one assignment in the original program, so that any use of V will be either a use of the value V_0 at entry to the program or a use of the value V_1 from the most recent execution of the assignment to V. Let X be the basic block of code that assigns to V, so X will determine the value of V when control flows along any edge $X \rightarrow Y$ to a basic block Y. When entered along $X \rightarrow Y$, the code in Y will see V_1 and be unaffected by V_0. If $Y \neq X$ but all paths to Y must still go through X (in which case X is said to *strictly dominate* Y), then the code in Y will always see V_1. Indeed, any node strictly dominated by X will always see V_1, no matter how far from X it may be. Eventually, however, control may be able to reach a node Z not strictly dominated by X. Suppose Z is the first such node on a path, so that Z sees V_1 along one inedge but may see V_0 along another inedge. Then Z is said to be *in the dominance frontier* of X and is clearly in need of a ϕ-function for V. In general, no matter how many assignments to V may appear in the original program and no matter how complex the control flow may be, we can place ϕ-functions for V by finding the dominance frontier of every node that assigns to V, then the dominance frontier of every node where a ϕ-function has already been placed, and so on.

The same concept of dominance frontiers used for computing SSA form can also be used to compute control dependences [20, 24], which 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 [2], program optimization, and program analysis [28].

1.3 Outline of the Rest of the Paper

Section 2 reviews the representation of control flow by a directed graph. Section 3 explains SSA form and sketches how to construct it. This section also considers variants of SSA form as defined here. Our algorithm can be adjusted to deal with these variants. Section 4 reviews the dominator tree concept and formalizes dominance frontiers. Then we show how to compute SSA form (§5) and the control dependence graph (§6) efficiently. Section 7 explains how to translate out of SSA form. Section 8 shows that our algorithms are linear in the size of programs restricted to certain control structures. We also give evidence of general linear behavior by reporting on experiments with FORTRAN programs. Section 9 summarizes the algorithms and time bounds, compares our technique with other techniques, and presents some conclusions.

2 Control Flow Graphs

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 [1, 36]. Basic blocks are indicated by the column of numbers in parentheses in Figure 5. A *control flow graph* is a directed graph whose nodes are the basic blocks of a program and two additional nodes, `Entry` and `Exit`. There is an edge from `Entry` to any basic block at which the program can be entered, and there is an edge to `Exit` from any basic block that can exit the program. For reasons related to the representation of control dependences and explained in Section 6, there is also an edge from `Entry` to `Exit`. The other edges of the graph 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 $X \to Y$ in the graph, and $Succ(X)$ is the set of all successors of X; similarly for predecessors. A node with more than one successor

```
I ← 1                       (1)
J ← 1                       (1)
K ← 1                       (1)
L ← 1                       (1)
repeat                      (2)
   if (P)                   (2)
      then do               (3)
         J ← I              (3)
         if (Q)             (3)
            then L ← 2      (4)
            else L ← 3      (5)
         K ← K + 1          (6)
      end                   (6)
      else K ← K + 2        (7)
   print(I,J,K,L)           (8)
   repeat                   (9)
      if (R)                (9)
         then L ← L + 4     (10)
   until (S)                (11)
   I ← I + 6                (12)
until (T)                   (12)
```

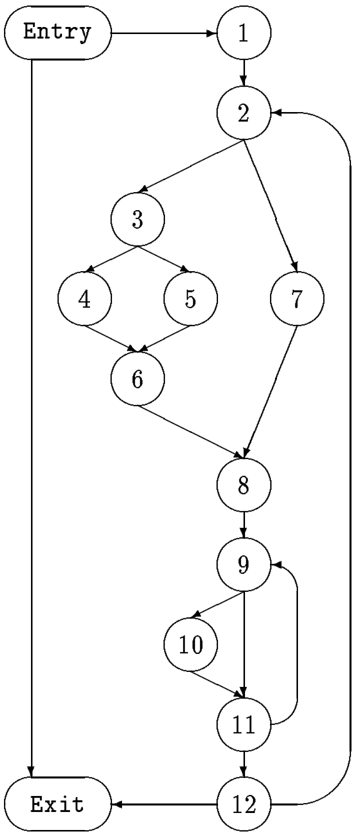

Figure 5. A simple program and its control flow graph.

is a *branch* node; a node with more than one predecessor is a *join* node. 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. Throughout this paper, CFG denotes the control flow graph of the program under discussion.

For any nonnegative integer J, a *path of length* J in CFG consists of a sequence of $J+1$ nodes (denoted $X_0, ..., X_J$) and a sequence of J edges (denoted $e_1, ..., e_J$) such that e_j runs from X_{j-1} to X_j for all j with $1 \le j \le J$. (We write $e_j : X_{j-1} \to X_j$.) As usual with sequences, one item (node or edge) may occur several times. The *null* path, with $J = 0$, is allowed. We write $p : X_0 \xrightarrow{*} X_J$ for an unrestricted path p, but $p : X_0 \xrightarrow{+} X_J$ if p is known to be nonnull.

Nonnull paths $p : X_0 \xrightarrow{+} X_J$ and $q : Y_0 \xrightarrow{+} Y_K$ are said to *converge* at a node Z if

$$X_0 \neq Y_0; \tag{1}$$

$$X_J = Z = Y_K; \tag{2}$$

$$(X_j = Y_k) \implies (j = J \textbf{ or } k = K). \tag{3}$$

Intuitively, the paths p and q start at different nodes and are almost node-disjoint, but they come together at the end. The use of **or** rather than **and** in (3) is deliberate. One of the paths may happen to be a cycle $Z \xrightarrow{+} Z$, but we still need to consider the possibility of convergence.

3 Static Single Assignment Form

This section initially assumes that programs have been expanded to an intermediate form in which each statement evaluates some expressions and uses the results either to determine a branch or to assign to some variables. (Other program constructs are considered in §3.1.) An assignment statement A has the form $LHS(A) \leftarrow RHS(A)$, where the left-hand side $LHS(A)$ is a tuple of distinct target variables $\langle U, V, ...\rangle$ and the right-hand side $RHS(A)$ is a tuple of expressions, with the same length as the LHS tuple. Each target variable in $LHS(A)$ is assigned the corresponding value from $RHS(A)$. In the examples already discussed, all tuples are 1-tuples and there is no need to distinguish between V and $\langle V\rangle$. Section 3.1 sketches the use of longer tuples in expanding other program constructs (such as procedure calls) so as to make explicit the variables used and/or changed by the statements in the source program. Such explicitness is a prerequisite for most optimizations anyway. The only novelty in our use of tuples is the fact that they provide a simple and uniform way to fold the results of analysis into the intermediate text. Practical concerns about the size of the intermediate text are addressed in §3.1.

Translating a program into static single assignment (*SSA*) form is a two-step process. In the first step, some trivial ϕ-functions $V \leftarrow \phi(V, V, ...)$ are inserted at some of the join nodes in the program's control flow graph. In the second step, new variables V_i (for $i = 0, 1, 2, ...$) are generated. Each mention of a variable V in the program is replaced by a mention of one of the new variables V_i, where a *mention* may be in a branch

```
I ← 1
J ← 1
K ← 1
L ← 1
repeat




   if (P)
      then do
         J ← I
         if (Q)
            then L ← 2
            else L ← 3

         K ← K + 1
      end
      else K ← K + 2



   print(I,J,K,L)
   repeat

      if (R)
         then L ← L + 4

   until (S)
   I ← I + 6
until (T)
```

```
I_1 ← 1
J_1 ← 1
K_1 ← 1
L_1 ← 1
repeat
   I_2 ← φ(I_3,I_1)
   J_2 ← φ(J_4,J_1)
   K_2 ← φ(K_5,K_1)
   L_2 ← φ(L_9,L_1)
   if (P)
      then do
         J_3 ← I_2
         if (Q)
            then L_3 ← 2
            else L_4 ← 3
         L_5 ← φ(L_3,L_4)
         K_3 ← K_2 + 1
      end
      else K_4 ← K_2 + 2
   J_4 ← φ(J_3,J_2)
   K_5 ← φ(K_3,K_4)
   L_6 ← φ(L_2,L_5)
   print(I_2,J_4,K_5,L_6)
   repeat
      L_7 ← φ(L_9,L_6)
      if (R)
         then L_8 ← L_7 + 4
      L_9 ← φ(L_8,L_7)
   until (S)
   I_3 ← I_2 + 6
until (T)
```

Figure 6. A simple program and its SSA form.

expression or on either side of an assignment statement. Throughout this paper, an assignment statement may be either an ordinary assignment or a ϕ-function.

A ϕ-function at entrance to a node X has the form $V \leftarrow \phi(R, S, \ldots)$, where $V, R, S, \ldots$ are variables. The number of operands $R, S, \ldots$ is the number of control flow predecessors of X. The predecessors of X are listed in some arbitrary fixed order, and the j-th operand of ϕ is associated with the j-th predecessor. If control reaches X from its j-th predecessor, then the run-time support[1] remembers j while executing the ϕ-functions

[1] When SSA form is used only as an intermediate form (Figure 1), there is no need actually to provide this support. For us, the semantics of ϕ are only important when assessing the correctness of intermediate steps in a sequence of program transformations beginning and ending with code that has no ϕ-functions. Others [6, 11, 52] have found it useful to give ϕ another parameter that incidentally encodes j, under various restrictions on the control flow.

in X. The value of $\phi(R, S, ...)$ is just 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 entering X. Any ϕ-functions in X are executed before the ordinary statements in X. Some variants of ϕ-functions as defined here are useful for special purposes. For example, each ϕ-function can be tagged with the node X where it appears [5]. When the control flow of a language is suitably restricted, each ϕ-function can be tagged with information about conditionals or loops [5, 6, 11, 52]. The algorithms in this paper apply to such variants as well.

Static single assignment form may be considered as a property of a single program or as a relation between two programs. A single program is defined to be *in* SSA form if each variable is a target of exactly one assignment statement in the program text. Translation *to* SSA form replaces the original program by a new program with the same control flow graph. For every original variable V, the following conditions are required of the new program:

1. If two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ converge at a node Z, and nodes X and Y contain assignments to V (in the original program), then a trivial ϕ-function $V \leftarrow \phi(V, ..., V)$ has been inserted at Z (in the new program).

2. Each mention of V in the original program or in an inserted ϕ-function has been replaced by a mention of a new variable V_i, leaving the new program in SSA form.

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

Translation to *minimal* SSA form is translation to SSA form with the proviso that the number of ϕ-functions inserted is as small as possible, subject to Condition 1 above. 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. However, extraneous ϕ-functions can cause information to be lost, and they always add unnecessary overhead to the optimization process itself. Thus it is important to place ϕ-functions only where they are required. One variant of SSA form [52] would sometimes forego placing a ϕ-function at a convergence point Z, so long as there are no more uses for V in or after Z. The ϕ-function could then be omitted without any risk of losing Condition 3. This has been called *pruned* SSA form [16], and it is sometimes preferable to our placement at all convergence points. However, as Figure 16 in §7 illustrates, our form is sometimes preferable to pruned form. When desired, pruned SSA form can be obtained by a simple adjustment of our algorithm [16, §5.1].

For any variable V, the nodes at which 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 a convergence point for two paths that originate at two different nodes, both 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 a convergence point for two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ that start at nodes X and Y already 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 convergence points of nonnull paths originating at nodes with assignments to V. The set of nodes needing ϕ-functions could thus be found by iterating an observation/insertion cycle.[2] The algorithm presented here obtains the same end results in much less time.

3.1 Other Program Constructs

If the source program computes expressions using constants and scalar variables to assign values to scalar variables, then it is straightforward to derive an intermediate text that is ready to be put into SSA form (or otherwise analyzed and transformed by an optimizing compiler). However, source programs commonly use other constructs as well: some variables are not scalars; some computations do not explicitly indicate which variables they use or change. This section sketches how to map some important constructs to an explicit intermediate text suitable for use in a compiler.

3.1.1 Arrays

It will suffice to consider one-dimensional arrays here. Arrays of higher dimension involve more notation but no more concerns.

If `A` and `B` are array variables, then array assignment statements like `A` $\leftarrow$ `B` or `A` $\leftarrow$ `0`, if allowed by the

[2] In typical cases, paths originating at ϕ-functions add nothing beyond the contributions from paths originating at ordinary assignments. The straightforward iteration is still too slow, even in typical cases.

source language, can be treated just like assignments to scalar variables. Many of the mentions of A in the source program, however, will be mentions of A(i) for some index i, which may be taken to be an integer variable. Treating A(i) as a variable would be awkward, both because an assignment to A(i) may or may not change the value of A(j) and because the value of A(i) could be changed by assigning to i rather than to A(i). An easier approach is illustrated in Figure 7. The entire array is treated like a single scalar variable, which may be one of the operands of Access or Update.[3] The expression Access(A,i) evaluates to the i-th component of A; the expression Update(A,j,V) evaluates to an array value that is of the same size as A and has the same component values, except for V as the value of the j-th component. Assigning a scalar value V to A(j) is equivalent to assigning an array value to the entire array A, where the new array value depends on the old one, as well as on the index j and on the new scalar value V. The translation to SSA form is unconcerned with whether the values of variables are large objects or what the operators mean.

$\leftarrow$ A(i)	$\leftarrow$ Access(A,i)	$\leftarrow$ Access(A_8, i_7)
A(j) $\leftarrow$ V	A $\leftarrow$ Update(A,j,V)	A_9 $\leftarrow$ Update(A_8, j_6, V_5)
$\leftarrow$ A(k) + 2	T $\leftarrow$ Access(A,k)	T_1 $\leftarrow$ Access(A_9, k_4)
	$\leftarrow$ T + 2	$\leftarrow$ T_1 + 2

Figure 7. Source code with array component references (on the left) is equivalent to code with explicit Access and Update operators that treat the array A just like a scalar (in the middle). Transformation to SSA form proceeds as usual (on the right).

As with scalars, translation of array references to SSA form removes some anti- and output-dependences [32]. In the program on the left in Figure 7, dependence analysis may prohibit reordering the use of A(i) by the first statement and the definition of A(j) by the second statement. After translation to SSA form, the two references to A have been renamed and reordering is then possible. For example, the two statements can execute concurrently. Where optimization does not reorder the two references, §7.2 describes how translation out of SSA form reclaims storage that would otherwise be necessary to maintain distinct variables.

Consider the loop shown on the left in Figure 8, which assigns to every component of array A. The value assigned to each component A(i) does not depend (even indirectly) on any of the values previously assigned to components of A. In terms of its effect on A, the whole loop is like an assignment of the form A $\leftarrow$ (...), where A is not used in (...). Any assignment to a component of A that is not used before entering such an

[3] This notation is similar to *Select* and *Update* [24], which in turn is similar to notation for referencing aggregate structures in a data flow language [23].

```
integer A(1:100)          integer A0(1:100)               integer A0(1:100)
                          integer A1(1:100)               integer A1(1:100)
                          integer A2(1:100)               integer A2(1:100)
i ← 1                     i2 ← 1                          i2 ← 1
repeat                    repeat                          repeat
                             i2 ← φ(i1,i3)                   i2 ← φ(i1,i3)
                             A1 ← φ(A0,A2)                   A1 ← φ(A0,A2)
  A(i) ← i                   A2 ← Update(A1,i2,i2)           A2 ← HiddenUpdate(i2,i2)
  i ← i + 1                  i3 ← i2 + 1                     i3 ← i2 + 1
until i ≥ 100             until i3 ≥ 100                  until i3 ≥ 100
```

Figure 8. Source loop with array assignment (on the left) is equivalent to code with an `Update` operator that treats the array `A` just like a scalar (in the middle). As §7.2 explains, the eventual translation out of SSA form will leave just one array here. Using `HiddenUpdate` (on the right) is a purely formal way to summarize some results of dependency analysis, if available.

initialization loop is dead code that should be eliminated. With or without SSA form, the `Update` operator in the middle of Figure 8 makes `A` appear to be live at entry to the loop.

One reasonable response to the crudeness of the `Update` operator is to accept it. Address calculations and other genuine scalar calculations can still be optimized extensively. Another response is to perform dependence analysis [3, 10, 32, 51], which can sometimes determine that no subsequent accesses of `A` require values produced by any other assignment to `A`. Such is the case for each execution of the assignment to `A(i)` on the left in Figure 8. The assignment statement can then be viewed as an initialization of `A`. The problem for us, or for anyone who uses `Update` to make arrays look like scalars, is to communicate some of the results of dependence analysis to optimizations (like dead code elimination) that are usually formulated in terms of "scalar" variables. A simple solution to this formal problem is shown on the right in Figure 8, where the `HiddenUpdate` operator does not mention the assigned array operand. The actual code generated for an assignment from a `HiddenUpdate` expression is *exactly* the same as for an assignment from the corresponding `Update` expression, where the hidden operand is supplied by the target of the assignment.

3.1.2 Structures

A structure can be generally regarded as an array, where references to structure fields are treated as references to elements of the array. Thus, an assignment to a structure field is translated into an `Update` of the structure, and a use of a structure field is translated into an `Access` of the structure. In the prevalent case of simple structure field references, this treatment results in arrays whose elements are indexed by constants.

Dependence analysis can often determine independence among such accesses, so that optimizations may move an assignment to one field far from an assignment to another field. If analysis or language semantics reveals a structure whose n fields are always accessed disjointly, then the structure can be decomposed into n distinct variables. The elements of such structures are united in the source program only for organizational reasons, and expressing the structure's decomposition in SSA form makes the program's actual use of the structure more apparent to subsequent optimization.

3.1.3 Implicit References to Variables

Constructing SSA form requires knowing those variables modified and used by a statement. In addition to those variables explicitly referenced, a statement may use or modify variables not mentioned by the statement itself. Examples of such *implicit* references are global variables modified or used by a procedure call, aliased variables, and dereferenced pointer variables. To obtain SSA form, we must account for implicit as well as explicit references, either by conservative assumptions or by analysis. Heap storage can be conservatively modeled by representing the entire heap as a single variable that is both modified and used by any statement that may change the heap. More refined modeling is also possible [15], but the conservative approach is already strong enough to support optimization of code that does not involve the heap but is interspersed with heap-dependent code.

For any statement S, three types of references affect translation into SSA form:

- $MustMod(S)$ is the set of variables that *must* be modified by execution of S.
- $MayMod(S)$ is the set of variables that *may* be modified by execution of S.
- $MayUse(S)$ is the set of variables whose values prior to execution of S *may* be used by S.

We represent implicit references for any statement S by transformation to an assignment statement A, where all the variables in $MayMod(S)$ appear in $LHS(A)$ and all the variables in $MayUse(S) \cup (MayMod(S) - MustMod(S))$ appear in $RHS(A)$.

An optimizing compiler may or may not have access to the bodies of all procedures called (directly or indirectly) or to summaries of their effects. If no specific information is available, it is customary to make conservative assumptions. A call might change any global variable or any parameter passed by reference, but it is reasonable to assume that variables local to the caller will *not* change. Such assumptions limit the extent that transfromations can be performed. Techniques are available to extract more detailed information: to determine parameter aliasing and effects of procedures on global variables, see [7, 8, 9, 19, 37, 42]; to determine pointer aliasing, see [14, 15, 27, 29, 33, 34, 44].

When a sophisticated analysis technique is applied, the usual result is that there are few side effects and the tuples (both LHS and RHS) are small. Small tuples can be represented directly. Sophisticated analysis, however, is often unavailable. Many compilers do no interprocedural analysis at all. Consider a call to an external procedure that has not been analyzed. Both tuples for the call must contain all global variables. The compiler's own representation of the tuples can still be compact. The representation can be a structure that includes a flag (set to indicate that all globals are in the tuple) plus a direct representation of the few local variables that are in the tuple because of parameter transmission. The intuitive explanations and theoretical analyses of optimization techniques (with or without SSA form) are conveniently formulated in terms of explicit tuples; compact representations can still be used in the implementations.

3.2 Overview of the SSA Algorithm

Translation to minimal SSA form is done in three steps:

1. The *dominance frontier* mapping is constructed from the control flow graph (§4.2).

2. Using the dominance frontiers, the locations of the ϕ-functions for each variable in the original program are determined (§5.1).

3. The variables are *renamed* (§5.2) by replacing each mention of an original variable V by an appropriate mention of a new variable V_i.

4 Dominance

Section 4.1 reviews the dominance relation [47] between nodes in the control flow graph and how to summarize this relation in a dominator tree. Section 4.2 introduces the *dominance frontier* mapping and gives an algorithm for its computation.

4.1 Dominator Trees

Let X and Y be nodes in the control flow graph CFG of a program. 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 $X \ggeq Y$ for domination. If 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. The root of a dominator tree is `Entry`, and any node Y other than `Entry` has $idom(Y)$ as its parent in the tree. The dominator tree for CFG from Figure 5 is shown in Figure 9. Let N and E be the numbers of nodes and edges in CFG. The dominator tree can be constructed in $O(E\alpha(E,N))$ time [35] or (by a more difficult algorithm) in $O(E)$ time [26]. For all practical purposes, $\alpha(E,N)$ is a small constant,[4] so this paper will consider the dominator tree to have been found in linear time.

The dominator tree of CFG has exactly the same set of nodes as CFG but 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.

4.2 Dominance Frontiers

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:

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

[4] Under the definition of α used in analyzing the dominator tree algorithm [35, p. 123], $N \leq E$ implies that $\alpha(E,N) = 1$ when $\log_2 N < 16$ and $\alpha(E,N) = 2$ when $16 \leq \log_2 N < 2^{16}$.

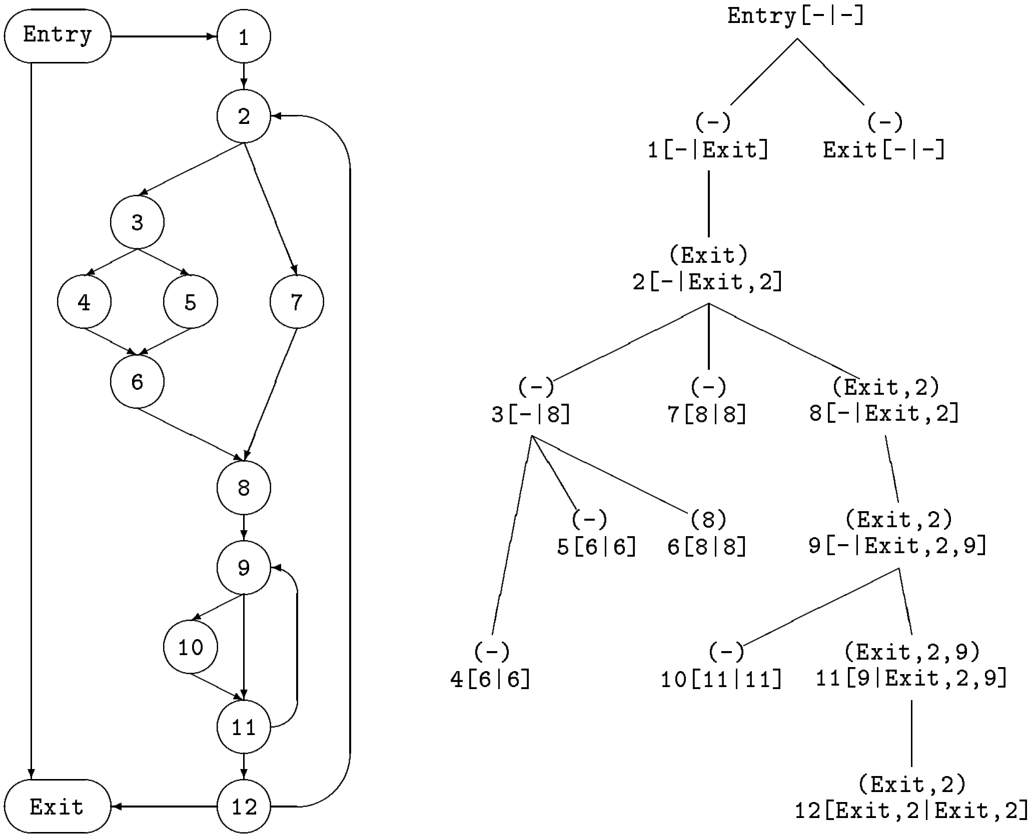

Figure 9. Control flow graph and dominator tree of the simple program.
The sets of nodes listed in ( ) and [ ] brackets summarize the dominance frontier calculation from §4.2.
Each node X is annotated with two sets $[DF_{local}(X) \mid DF(X)]$ and a third set $(DF_{up}(X))$.

Computing $DF(X)$ directly from the definition would require searching much of the dominator tree. The total time 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 $\sum_X |DF(X)|$ of the mapping, we define two intermediate sets DF_{local} and DF_{up} for each node such that the following equation holds:

$$DF(X) \;=\; DF_{local}(X) \;\cup \bigcup_{Z \in Children(X)} DF_{up}(Z). \tag{4}$$

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

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

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

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

Lemma 1 *The dominance frontier equation* (4) *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 $U \to Y$ 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

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

The $\Longleftarrow$ part is true because the immediate dominator is defined to be a *strict* dominator. For the $\Longrightarrow$ 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 $X \to Y$, 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

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

The $\Longleftarrow$ part is true because strict dominance is the transitive closure of immediate dominance. For the $\Longrightarrow$ 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 $U \to Y$, 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 for computing dominance frontiers given in Figure 10. The `/*local*/` line effectively computes $DF_{local}(X)$ on the fly and uses it in (4) 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 visiting each of its children. To illustrate the working of this algorithm, we have annotated the dominator tree in Figure 9 with the information in `[ ]` and `( )` brackets.

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

Figure 10. Calculation of $DF(X)$ for each CFG node X.

Theorem 1 *The algorithm in Figure 10 is correct.*

Proof. Direct from the preceding lemmas. □

Let CFG have N nodes and E edges. The loop over $Succ(X)$ in Figure 10 examines each edge just once, so all executions of the `/*local*/` line are complete in time $O(E)$. Similarly, all executions of the

/*up*/ line are complete in time $O(\text{size}(DF))$. The overall time is thus $O(E + \text{size}(DF))$, which amounts to a worst-case complexity of $O(E + N^2)$. However, §8 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 [2].

4.3 Relating Dominance Frontiers to Joins

We start by stating more formally the nonrecursive characterization of where the ϕ-functions should be located. Given a set $\mathcal{S}$ of CFG nodes, the set $J(\mathcal{S})$ of *join* nodes 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 converge at Z. 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 the efficient worklist algorithm in Figure 11; the formulation here is convenient for relating iterated dominance frontiers to iterated joins. If the set $\mathcal{S}$ is the set of assignment nodes for a variable V, then we will show that

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

(this equation depends on the fact that `Entry` is in $\mathcal{S}$) 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 Figure 11.

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. Moreover, unless X dominates every node on p, the node X' can be chosen in $DF^+(\{X\})$.*

Proof. If X dominates every node on p, then we just choose $X' = X$ to get all the claimed properties of X'. We may assume that some of the nodes on p are *not* dominated by X. Let the sequence of nodes on p be $X = X_0, ..., X_J = Z$. For the smallest i such that X does not dominate X_i, the predecessor X_{i-1} is dominated by X and so puts X_i into $DF(X)$. Thus there are choices of j with $X_j \in DF^+(\{X\})$. Consider $X' = X_j$ for the largest j with $X_j \in DF^+(\{X\})$. We will show that $X' \geq\!\!\!\!> Z$. Suppose not. Then $j < J$ and there is a first k with $j < k \leq J$ such that X' does not dominate X_k. The predecessor X_{k-1} is dominated by X' and so puts X_k into $DF(X')$. Thus $X_k \in DF(DF^+(\{X\})) = DF^+(\{X\})$, contradicting the choice of j. $\square$

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 converge at Z. Then $Z \in DF^+(\{X\}) \cup DF^+(\{Y\})$.*

Proof. We consider three cases that are obviously exhaustive. In the first two cases, we prove that $Z \in DF^+(\{X\}) \cup DF^+(\{Y\})$. Then we show that the first two cases are (unobviously) exhaustive because the third case leads to a contradiction. Let X' be from Lemma 4 for the path p, with sequence of nodes $X = X_0, ..., X_J = Z$. Let Y' be from Lemma 4 for the path q, with sequence of nodes $Y = Y_0, ..., Y_K = Z$.

Case 1. We suppose that X' is on q and show that $Z \in DF^+(\{X\})$. By the definition of convergence (specifically, (3) in §2), $X' = Z$. We may now assume that $Z = X$ and X dominates every node on p. (Otherwise, Lemma 4 already asserts $Z \in DF^+(\{X\})$.) Because X dominates the predecessor X_{J-1} of Z

but does not *strictly* dominate Z, we have $Z \in DF(X) \subseteq DF^{+}(\{X\})$.

Case 2. We suppose that Y' is on p and show that $Z \in DF^{+}(\{Y\})$, reasoning just as in Case 1.

Case 3. We derive a contradiction from the suppositions that X' is *not* on q and Y' is *not* on p. Because $X' \ggeq Z$ but X' is not on q, $X' \gg Y_K = Z$ and therefore dominates all predecessors of Y_K. In particular, $X' \ggeq Y_{K-1}$. But $X' \neq Y_{K-1}$, so $X' \gg Y_{K-1}$ and we continue inductively to show that $X' \gg Y_k$ for all k. In particular, $X' \gg Y'$. On the other hand, by similar reasoning from the supposition that $Y' \ggeq Z$ but Y' is not on p, we can show that $Y' \gg X'$. Two nodes cannot strictly dominate each other, so Case 3 is impossible. □

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 *none* of the nodes are dominated by X. The paths therefore converge at 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 nodes with assignments to V.*

Proof. By Lemma 6 and induction on i in the definition of J^{+}, we can show that

$$J^{+}(\mathcal{S}) \subseteq DF^{+}(\mathcal{S}). \tag{5}$$

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

$$DF^+(\mathcal{S}) \subseteq J^+(\mathcal{S}). \tag{6}$$

The induction step is as follows:

$$\begin{array}{rcccl} DF_{i+1} & = & DF(\mathcal{S} \cup DF_i) & \subseteq & DF(\mathcal{S} \cup J^+(\mathcal{S})) \\ & \subseteq & J(\mathcal{S} \cup J^+(\mathcal{S})) & = & J^+(\mathcal{S}). \end{array}$$

The set of nodes that need ϕ-functions for V is precisely $J^+(\mathcal{S})$, so (5) and (6) prove the theorem. □

5 Construction of Minimal SSA Form

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

The algorithm in Figure 11 inserts trivial ϕ-functions. The outer loop of this algorithm is performed once for each variable V in the program. Several data structures are used:

- W is the worklist of CFG nodes 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 node X in the worklist ensures that each node Y in $DF(X)$ receives a ϕ-function. Each iteration terminates when the worklist becomes empty.

- $Work(*)$ is an array of flags, one flag for each node, where $Work(X)$ indicates whether X has ever been added to W during the current iteration of the outer loop.

- $HasAlready(*)$ is an array of flags, one for each node, where $HasAlready(X)$ indicates whether a ϕ-function for V has already been inserted at X.

The flags $Work(X)$ and $HasAlready(X)$ are independent. We need two flags because the property of assigning to V is independent of the property of needing a ϕ-function for V. The flags could have been implemented with just the values `true` and `false`, but this would require additional record keeping to reset any `true` flags between iterations, without the expense of looping over all the nodes. It is simpler to devote an integer to each flag and to test flags by comparing them with the current iteration count.

```
IterCount ← 0
for each node X do
   HasAlready(X) ← 0
   Work(X) ← 0
end
W ← ∅
for each variable V do
   IterCount ← IterCount + 1
   for each X ∈ 𝒜(V) do
      Work(X) ← IterCount
      W ← W ∪ {X}
   end
   while W ≠ ∅ do
      take X from W
      for each Y ∈ DF(X) do
         if HasAlready(Y) < IterCount
            then do
               place ⟨ V ← φ(V, ..., V) ⟩ at Y
               HasAlready(Y) ← IterCount
               if Work(Y) < IterCount
                  then do
                     Work(Y) ← IterCount
                     W ← W ∪ {Y}
                  end
            end
      end
   end
end
```

Figure 11. Placement of ϕ-functions.

Let each node X have $A_{orig}(X)$ original assignments to variables, where each ordinary assignment statement $LHS \leftarrow RHS$ contributes the length of the tuple LHS to $A_{orig}(X)$. Counting assignments to variables is one of several measures of program size.[5] By this measure, the program expands from size $A_{orig} = \sum_X A_{orig}(X)$ to size $A_{tot} = \sum_X A_{tot}(X)$, where each ϕ-function placed at X contributes 1 to $A_{tot}(X) = A_{orig}(X) + A_{\phi}(X)$. There is a similar expansion in the number of mentions of variables, from $M_{orig} = \sum_X M_{orig}(X)$ to $M_{tot} = \sum_X M_{tot}(X)$, where each ϕ-function placed at X contributes 1 plus the indegree of X to $M_{tot}(X) = M_{orig}(X) + M_{\phi}(X)$.

[5] The various measures relevant here are reviewed when the whole SSA translation process is summarized in §9.1.

Placing a ϕ-function at Y in Figure 11 has cost linear in the indegree of Y, so there is an $O(\sum_X M_\phi(X)$ contribution to the running time from the work done when $HasAlready(Y) < IterCount$. Replacing mentions of variables will contribute at least $O(M_{tot})$ to the running time of any SSA translation algorithm, so the cost of placement can be ignored in analyzing the contribution of Figure 11 to the $O(...)$ bound for the whole process. The $O(N)$ cost of initialization can be similarly ignored because it is subsumed by the cost of the dominance frontier calculation. What cannot be ignored is the cost of managing the worklist W. The statement `take` X `from` W in Figure 11 is performed $A_{tot}(X)$ times, and each time incurs a cost linear in $|DF(X)|$ because all $Y \in DF(X)$ are polled. The contribution of Figure 11 to the running time of the whole process is therefore $O(\sum_X (A_{tot}(X) \times |DF(X)|))$. Sizing the output in the natural way as A_{tot}, we can also describe the contribution as $O(A_{tot} \times avrgDF)$, where the weighted average

$$avrgDF \stackrel{\text{def}}{=} \left(\sum_X (A_{tot}(X) \times |DF(X)|)\right) / \left(\sum_X A_{tot}(X)\right) \tag{7}$$

emphasizes the dominance frontiers of nodes with many assignments. As §8 shows, the dominance frontiers are small in practice and Figure 11 is effectively $O(A_{tot})$. This in turn is effectively $O(A_{orig})$.

5.2 Renaming

The algorithm in Figure 12 renames all mentions of variables. New variables denoted V_i, where i is an integer, are generated for each variable V. Figure 12 begins a top-down traversal of the dominator tree by calling `SEARCH` at the root node `Entry`. 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 11, we need a loop over all variables only 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 variable 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 a tuple of expressions and the left-hand side $LHS(A)$ is a tuple of distinct target variables. These tuples change as mentions of variables are renamed, but the original target tuple is still remembered as $oldLHS(A)$. To minimize notation, a conditional branch is treated like an ordinary assignment to a special variable whose value indicates which edge the branch should follow. A real implementation would recognize that a conditional branch involves a little less work than a genuine assignment: the LHS part of the processing can be omitted.

The processing of an assignment statement A considers the mentions of variables in A. The simplest case is that of a target variable V in the tuple $LHS(A)$. We need a new variable V_i. By keeping a count $C(V)$ of the number of assignments to V that have already been processed, we can find an appropriate new variable by using $i = C(V)$ and then incrementing $C(V)$. To facilitate renaming uses of V in the future, we also push i (which identifies V_i) onto a stack $S(V)$ of (integers that identify) new variables replacing V.

A subtler computation is needed for the right-hand side $RHS(A)$. Consider any variable V used in $RHS(A)$ — i.e., V appears in at least one of the expressions in the tuple. We want to replace V by V_i, where V_i is the target of the assignment that produces the value for V actually used in $RHS(A)$. There are two subcases because A may be either an ordinary assignment or a ϕ-function. Both subcases get V_i from the top of the stack $S(V)$, but they inspect $S(V)$ at different times in Figure 12. Lemma 10 later in this section shows that V_i is correctly chosen in both subcases.

The correctness proof for renaming depends on results from §4 and on three more lemmas. We start by showing that it makes sense to speak of "the" assignment to a variable in the transformed program.

```
for each variable V do
   C(V) ← 0
   S(V) ← EmptyStack
end
call SEARCH(Entry)

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

Figure 12. Renaming mentions of variables.
The list of uses of V_i grows with each replacement of V by V_i in an RHS.

Lemma 8 *Each new variable V_i in the transformed program is a 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. To show that there is at least one assignment to V_i, we consider the two ways V_i can be mentioned. If V_i is mentioned on the LHS on an assignment, then there is nothing more to show. If the value of V_i is used, on the other hand, then $i = Top(S(V))$ was true at the time when the algorithm renamed the old use of V to a use of V_i. At the earlier time when i was pushed onto $S(V)$, it was pushed because an assignment to V had just been changed to an assignment to V_i. □

A node X may contain assignments to the variable V that appeared in the original program or were introduced to receive the value of a ϕ-function for V. Let $TopAfter(V, X)$ denote the new variable in effect for V after *all* statements in node X have been considered by the first loop in Figure 12. Specifically, we consider the top of each stack $S(V)$ at the end of this loop and define

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

If there are no assignments to V in X, then the top-down traversal ensures that $TopAfter(V, X)$ is inherited from the closest dominator of X that assigns to V.

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

$$TopAfter(V, X) = TopAfter(V, idom(Y)). \tag{8}$$

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

By Lemma 2, $Y \in DF_{local}(X) \subseteq DF(X)$ and thus X does not assign to a new variable derived from V. Let U be the first node in the sequence $idom(X), idom(idom(X)), \ldots$ that assigns to such a variable. Then

$$TopAfter(V, X) = TopAfter(V, U). \tag{9}$$

Because U assigns to a new variable derived from V, $Y \notin DF(U)$. But U dominates a predecessor of Y, so U strictly dominates Y. For any Z with $U \gg Z \ggeq idom(Y)$, we get $Z \ggeq X$ because $Z \gg Y$ and there is an edge $X \to Y$. By the choice of U, $U \gg Z \ggeq X$ implies that Z does not assign to a new variable derived from V. Therefore

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

and (8) follows from (9). □

The preceding lemma will help establish Condition 3 in the definition of SSA form, but first we must extend *TopAfter* so as to specify which new variable corresponds to V just before and just after each statement A. There is one iteration of the first loop in Figure 12 for A. We consider the top of each stack just before and just after this iteration to define

$$\begin{aligned} TopBefore(V, A) &\stackrel{\text{def}}{=} V_i \quad \text{where} \quad i = Top(S(V)) \text{ before processing } A; \\ TopAfter(V, A) &\stackrel{\text{def}}{=} V_i \quad \text{where} \quad i = Top(S(V)) \text{ after processing } A. \end{aligned}$$

In particular, if A happens to be the last statement in block X, then $TopAfter(V, A) = TopAfter(V, X)$. If, on the other hand, A is followed by another statement B, then $TopAfter(V, A) = TopBefore(V, B)$.

Lemma 10 *Consider any control flow path in the transformed program and the same path in the original program. Consider any variable V and any occurrence of a statement A along the path. If A is from the original program, then the value of V just before executing A in the original program is the same as the value of $TopBefore(V, A)$ just before executing A in the transformed program.*

Proof. We use induction along the path. We consider the k-th statement executed along the path and assume that, for all $j < k$, the j-th statement T is either not from the original program[6] or has each variable V agreeing with $TopBefore(V, T)$ just before T. We assume that the k-th statement A is from the original program and show that V agrees with $TopBefore(V, A)$ just before A.

Case 1. Suppose that A is not the first original statement in its basic block Y. Let T be the statement just before A in Y. By the induction hypothesis and the semantics of assignment, V agrees with $TopAfter(V, T)$

[6] It could be a nominal `Entry` assignment or a ϕ-function.

just after T. Thus V agrees with $TopBefore(V, A)$ just before A.

Case 2. Suppose that A is the first original statement in its basic block Y. Let $X \to Y$ be the edge followed by the control path. We claim that V agrees with $TopAfter(V, X)$ when control flows along this edge.

If X = `Entry`, then $TopAfter(V, X)$ is V_0 and does hold the entry value of V. If $X \neq$ `Entry`, let T be the last statement in X. By the induction hypothesis and the semantics of assignment, V agrees with $TopAfter(V, T)$ just after T and hence with $TopAfter(V, X)$ when control flows along $X \to Y$.

We must still bridge the gap between knowing that V agrees with $TopAfter(V, X)$ along $X \to Y$ and knowing that V agrees with $TopBefore(V, A)$ just before doing A in Y. As Figure 13 illustrates, there may or may not be a ϕ-function for V ahead of A in the transformed program.

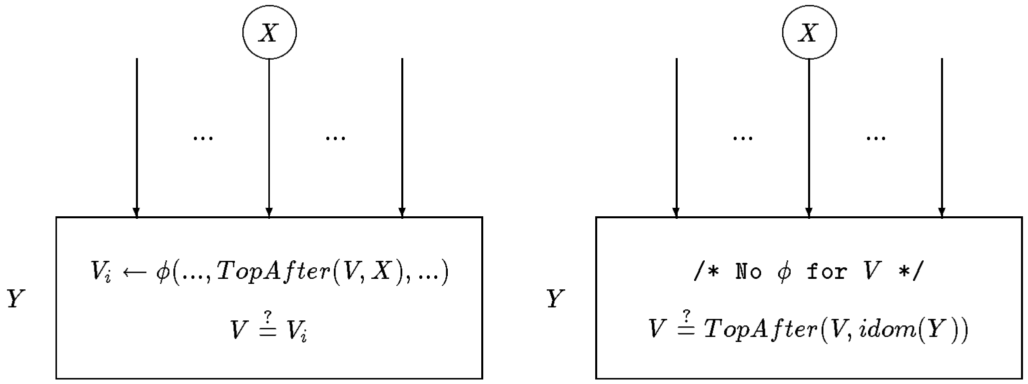

Figure 13. In the proof of Lemma 10, the node Y may or may not have a ϕ-function for V.

Case 2.1. Suppose that V has a ϕ-function $V_i \leftarrow \phi(...)$ in Y. The ϕ operand corresponding to $X \to Y$ is $TopAfter(V, X)$, thanks to the loop over successors of X in Figure 12. This is the operand whose value is assigned to V_i, which is $TopBefore(V, A)$. Thus V agrees with $TopBefore(V, A)$ just before doing A in Y.

Case 2.2. Suppose that V does not have a ϕ-function in Y. By Lemma 9, V agrees with $TopAfter(V, X) = TopAfter(V, idom(Y)) = TopBefore(V, A)$ just before doing A in Y. $\square$

Theorem 3 *Any program can be put into minimal SSA form by computing the dominance frontiers and then applying the algorithms in Figure 11 and Figure 12. Let A_{tot} be the total number of assignments to variables in the resulting program.*[7] *Let M_{tot} be the total number of mentions of variables in the resulting program. Let E be the number of edges in CFG. Let $avrgDF$ be the weighted average (7) of dominance frontier sizes. Then the running time of the whole process is*

$$O(E + \sum_X |DF(X)| + (A_{tot} \times avrgDF) + M_{tot}).$$

Proof. Figure 11 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 translation to SSA form with the fewest possible ϕ-functions. We must still show that renaming is done correctly by Figure 12. Condition 2 in the definition follows from Lemma 8. Condition 3 follows from Lemma 10.

Let N be the number of nodes in CFG. The dominator tree has $O(N)$ edges and Figure 12 runs in $O(N + E + M_{tot})$ time. The $N + E$ term is subsumed by the $O(E + \sum_X |DF(X)|)$ cost of computing the dominance frontiers, so Figure 12 contributes $O(M_{tot})$. As was explained at the end of §5.1, Figure 11 contributes $O(A_{tot} \times avrgDF)$. $\square$

6 Construction of Control Dependences

In this section we show that control dependences [24] 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.[8] 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.

[7] Each ordinary assignment $LHS \leftarrow RHS$ contributes the length of the tuple LHS to A_{tot}, and each ϕ-function contributes 1 to A_{tot}.

[8] The postdominance relation in [24] 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 of the dominance relation.

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.

In other words, there is some edge from X that definitely causes Y to execute, and there is also some path from X that avoids executing Y. We associate with this control dependence from X to Y the label on the control flow edge from X that causes Y to execute. Our definition of control dependence here can easily be shown to be equivalent to the original definition [24].

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. $\square$

The *reverse control flow graph RCFG* has the same nodes as the 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 14 applies this result to compute control dependences. After building the dominator tree for $RCFG$ by the standard method [35] in time $O(E\alpha(E, N))$, we spend $O(\text{size}(RDF))$ finding dominance frontiers and then inverting them. The total time is thus $O(E + \text{size}(RDF))$ for all practical purposes. By applying the algorithm in Figure 14 to the control flow graph in Figure 5, we obtain the control dependences in Figure 15. 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 10 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 14. 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	2,8,9,11,12

Figure 15. Control dependences of program in Figure 5.

7 Translating From SSA Form

Many powerful analysis and transformation techniques can be applied to programs in SSA form. Eventually, however, a program must be executed. The ϕ-functions have precise semantics, but they are generally not represented in existing target machines. This section describes how to translate out of SSA form by replacing each ϕ-function with some ordinary assignments. Naive translation could yield inefficient object code, but efficient code can be generated if two already useful optimizations are applied: dead code elimination and storage allocation by coloring.

Naively, a k-input ϕ-function at entrance to a node X can be replaced by k ordinary assignments, one at the end of each control flow predecessor of X. This is always correct, but these ordinary assignments sometimes perform a good deal of useless work. If the naive replacement is preceded by dead code elimination and then followed by coloring, however, the resulting code is efficient.

7.1 Dead Code Elimination

The original source program may have *dead code* (i.e., code that has no effect on any program output). Some of the intermediate steps in compilation (such as procedure integration) may also introduce dead code. Code that once was live may become dead in the course of optimization. With so many possible sources for dead code, it is natural to perform (or repeat) dead code elimination late in the optimization process, rather than burden many intermediate steps with concerns about dead code.

Translation to SSA form is one of the compilation steps that may introduce dead code. Suppose that V is assigned and then used along each branch of an `if ... then ... else ...`, but that V is never used after the join point. The original assignments to V are live, but the added assignment by the ϕ-function is dead. Often such dead ϕ-functions are useful, as in the equivalencing and redundancy elimination algorithms that are based on SSA form [5, 43]. One such use is shown in Figure 16. Although others have avoided placement of dead ϕ-functions in translating to SSA form [16, 52], we prefer to include the dead ϕ-functions to increase optimization opportunities.

There are many different definitions of dead code in the literature. Dead code is sometimes defined to be unreachable code and sometimes defined (as it is here) to be ineffectual code. In both cases, it is desirable

```
if P1                              if P1
   then do                            then do
      Y1 ← 1                             Y1 ← 1
      use of Y1                          use of Y1
   end                                end
   else do                            else do
      Y2 ← X1                            Y2 ← X1
      use of Y2                          use of Y2
   end                                end
Y3 ← φ(Y1,Y2)                      Y3 ← φ(Y1,Y2)
   ...                                ...
if P1
   then  Z1 ← 1
   else  Z2 ← X1
Z3 ← φ(Z1,Z2)
use of Z3                          use of Y3
```

Figure 16. On the left is an unoptimized program containing a dead ϕ-function that assigns to Y_3. The value numbering technique in [5] can determine that Y_3 and Z_3 have the same value. Thus, Z_3 and many of the computations that produce it can be eliminated. The dead ϕ-function is brought to life by using Y_3 in place of Z_3.

to use the broadest possible definition, subject to the correctness condition that "dead" code really can be safely removed.[9] A procedural version of the definition is more intuitive than a recursive version, so we prefer the procedural style. Initially, all statements are tentatively marked *dead*. Some statements, however, need to be marked *live* because of the conditions listed below. Marking these statements live may cause others to be marked live. When the natural worklist eventually empties, any statements that are still marked dead are truly dead and can be safely removed from the code.

A statement will be marked live if and only if at least one of the following holds:

1. The statement is one that should be assumed to affect program output, such as an I/O statement, an assignment to a reference parameter, or a call to a routine that may have side-effects.

2. The statement is an assignment statement and there are statements already marked live that use some of its outputs.

3. The statement is a conditional branch and there are statements already marked live that are control dependent on this conditional branch.

[9] The definition used here is broader than the usual one [1, p. 595] and similar to that of "faint" variables [25, p. 489].

Several published algorithms eliminate dead code in the narrower sense that requires every conditional branch to be marked live [30, 31, 38].[10] Our algorithm, given in Figure 17, goes one step further in eliminating dead conditional branches. (An unpublished algorithm by R. Paige does this also.) The following data structures are used.

- $Live(S)$ indicates that statement S is live.
- $PreLive$ is the set of statements whose execution is initially assumed to affect program output. Statements that result in I/O or side-effects outside the current procedure scope are typically included.
- $WorkList$ is a list of statements whose liveness has been recently discovered.
- $Definers(S)$ is the set of statements that provide values used by statement S.
- $Last(B)$ is the statement that terminates basic block B.
- $Block(S)$ is the basic block containing statement S.
- $ipdom(B)$ is the basic block that immediately postdominates block B.
- $CD^{-1}(B)$ is the set of nodes that are control dependence predecessors of the node corresponding to block B. This is the same as $RDF(B)$ from Figure 14.

After the algorithm discovers the live statements, all those still not marked live are deleted.[11] Other optimizations (such as code motion) may leave empty blocks, so there is already ample reason for a late optimization to remove them. The empty blocks left by dead code elimination can be removed along with any other empty blocks.

The fact that statements are considered dead until marked live is crucial for condition (2). Statements that depend (transitively) on themselves are never marked live unless required by some other live statement. Condition (3) is handled by the loop over $CD^{-1}(Block(S))$. A basic block whose termination controls a block with live statements is itself live.

[10] The more readily accessible [31] contains a typographical error; the earlier technical report [30] should be consulted for the correct version.

[11] A conditional branch can be deleted by transforming it to an unconditional branch to any one of its prior targets.

```
for each statement S do
   if S ∈ PreLive
      then Live(S) ← true
      else Live(S) ← false
end
WorkList ← PreLive

while (WorkList ≠ ∅) do
   take S from WorkList

   for each D ∈ Definers(S) do
      if Live(D) = false
         then do
            Live(D) ← true
            WorkList ← WorkList ∪ { D }
         end
   end

   for each block B in CD⁻¹(Block(S)) do
      if Live(Last(B)) = false
         then do
            Live(Last(B)) ← true
            WorkList ← WorkList ∪ { Last(B) }
         end
   end
end

for each statement S do
   if Live(S) = false
      then delete S from Block(S)
end
```

Figure 17. Dead code elimination.

7.2 Allocation by Coloring

At first, it might seem possible simply to map all occurrences of V_i back to V and delete all of the ϕ-functions. However, the new variables introduced by translation to SSA form cannot always be eliminated, because optimizations may have capitalized on the storage-independence of the new variables. The useful persistence of the new variables introduced by translation to SSA form can be illustrated by the code motion example in Figure 18. The source code (on the left) assigns to V twice and uses it twice. The SSA form (in the middle) can be optimized by moving the invariant assignment out of the loop, yielding a program with separate variables for separate purposes (on the right). The dead assignment to V_3 will be eliminated. These optimizations leave a region in the program where V_1 and V_2 are simultaneously live. Thus, both variables are required: the original variable V cannot substitute for both renamed variables.

```
while (...) do


   read V
   W ← V + W
   V ← 6
   W ← V + W
end
```

```
while (...) do
   W3 ← φ(W0, W2)
   V3 ← φ(V0, V2)
   read V1
   W1 ← V1 + W3
   V2 ← 6
   W2 ← V2 + W1
end
```

```
V2 ← 6
while (...) do
   W3 ← φ(W0, W2)
   V3 ← φ(V0, V2)
   read V1
   W1 ← V1 + W3

   W2 ← V2 + W1
end
```

Figure 18. Program that really uses two instances for a variable after code motion.
The source program is on the left; the unoptimized SSA form is in the middle;
the result of code motion is on the right.

Any graph coloring algorithm [12, 13, 17, 18, 21] can be used to reduce the number of variables needed and thereby remove most of the associated assignment statements. The choice of coloring technique should be guided by the eventual use of the output. If the goal is to produce readable source code, then it is desirable to consider each original variable V separately, coloring just the SSA variables derived from V. If the goal is machine code, then all of the SSA variables should be considered at once. In both cases, the process of coloring will change most of the assignments that were inserted to model the ϕ-functions into identity assignments, i.e., assignments of the form $V \leftarrow V$. These identity assignments can all be deleted.

Storage savings are especially noticable for arrays. If optimization does not perturb the order of the first two statements in Figure 7, then arrays A_8 and A_9 can be assigned the same color and hence can share the same storage. The array $\texttt{A}_9$ is then assigned an `Update` from an identically colored array. Such operations can be implemented inexpensively by assigning to just one component if the arrays share storage. In particular, the actual operation performed by `HiddenUpdate` is always of this form.

8 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 to `V`. Program structure alone determines dominance frontiers and the number of control dependences. 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. (We assume expressions and predicates perform no internal branching.) Such programs can be described by the grammar given in Figure 19. 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 19. 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 19. 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. When a production expands a nonterminal parse tree node S, a new subgraph is inserted into CFG in place of S. In this new subgraph, each node corresponds to a symbol on the right-hand side of the production.

We will show that applying any of the productions preserves the following invariants:

- Each CFG node S corresponding to an unexpanded `<statement>` symbol has at most one node in its dominance frontier.
- Each CFG node T corresponding to a terminal symbol has at most two nodes in its dominance frontier.

We consider the productions in turn.

(1) This production adds a CFG node S and edges $\texttt{Entry} \to S \to \texttt{Exit}$, 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 a 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. $\square$

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 5. For each loop, the dominance frontier of the entrance to 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 it seems that the 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 algorithms for constructing dominance frontiers and placing ϕ-functions in the PTRAN system, which already offered the required local data flow and control flow analysis [2]. We ran these algorithms on 61 library procedures from EISPACK [46] and 160 procedures from two "Perfect" [39] benchmarks. Some summary statistics of these procedures are shown in Figure 20. These Fortran programs

Package name	Statments in all procedures	Statements per procedure			Description
		min	median	max	
EISPACK	7034	22	89	327	Dense matrix Eigen–vectors and –values
FLO52	2054	9	54	351	Flow past an airfoil
SPICE	14093	8	43	753	Circuit simulation
Totals:	23181	8	55	753	221 Fortran procedures

Figure 20. Summary statistics of our experiment

were chosen because they contain irreducible intervals and other unstructured constructs. As the plot in Figure 21 shows, the size of the dominance frontier mapping appears to vary linearly with program size. The ratio of these sizes ranged from 0.6 (the `Entry` node has an empty dominance frontier) to 2.1.

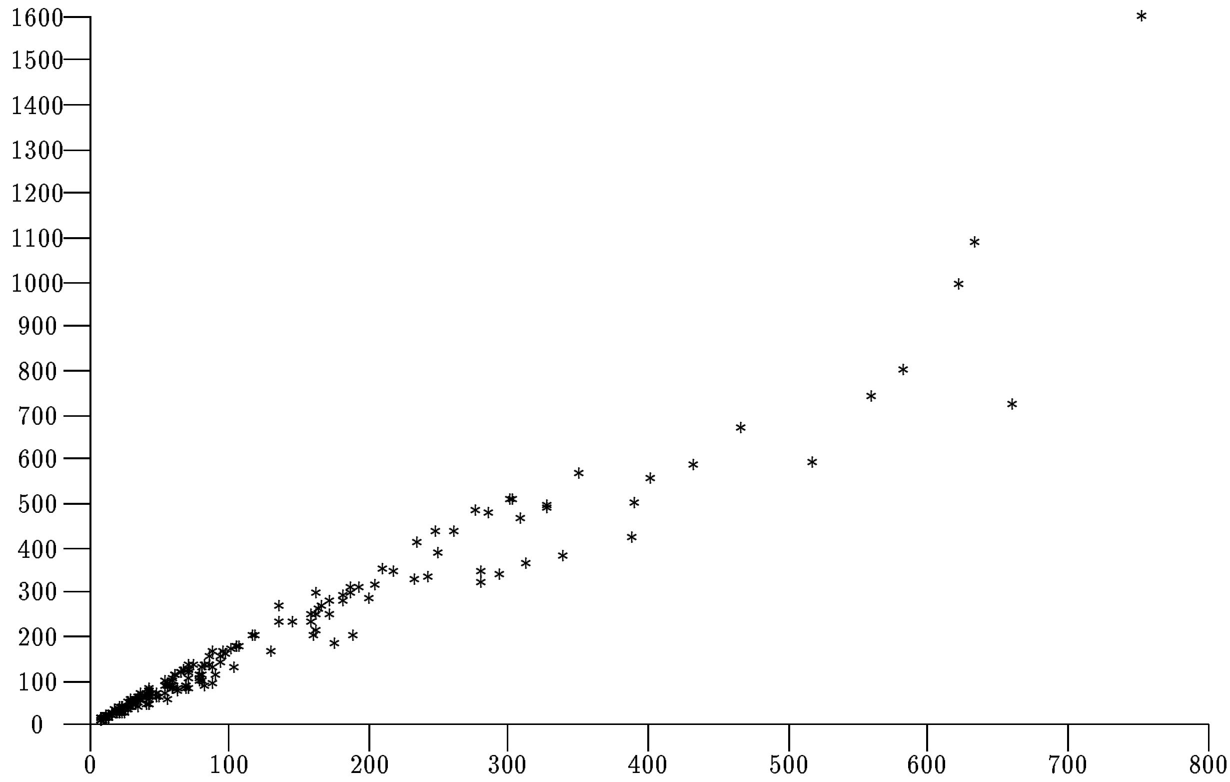

Figure 21. Size of dominance frontier mapping vs. number of program statements.

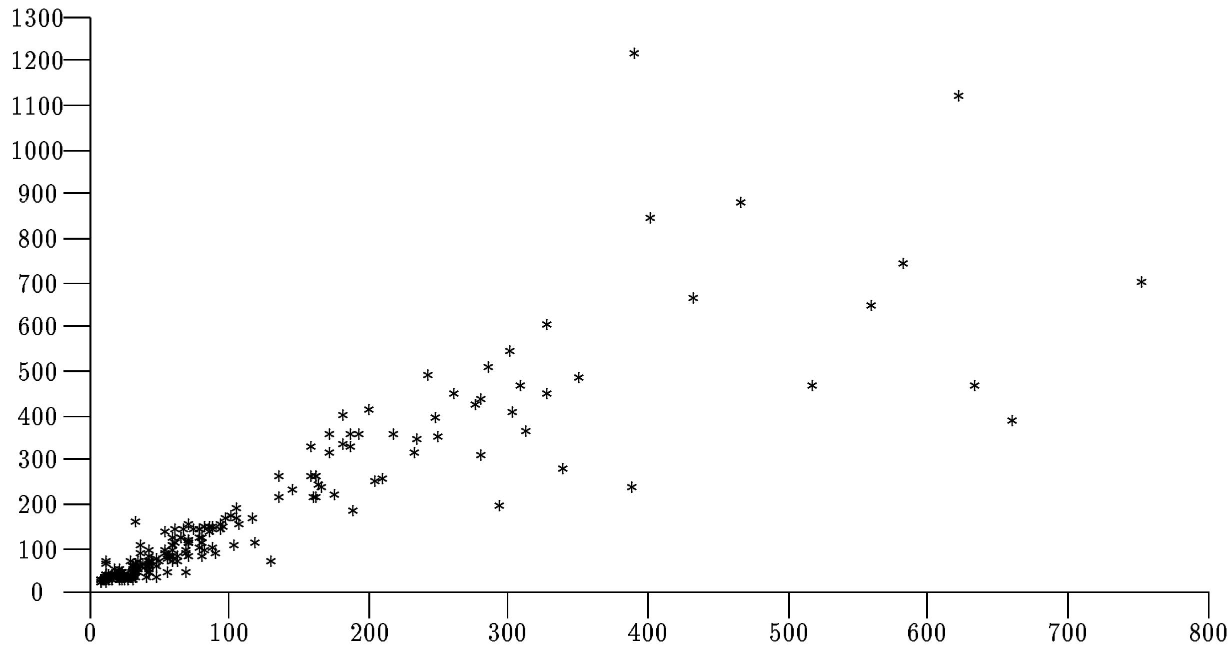

Figure 22. Number of ϕ-functions vs. number of program statements.

For the programs we tested, the plot in Figure 22 shows that the number of ϕ-functions is also linear in the size of the original program. The ratio of these sizes ranged from 0.5 to 5.2. The largest ratio occurred

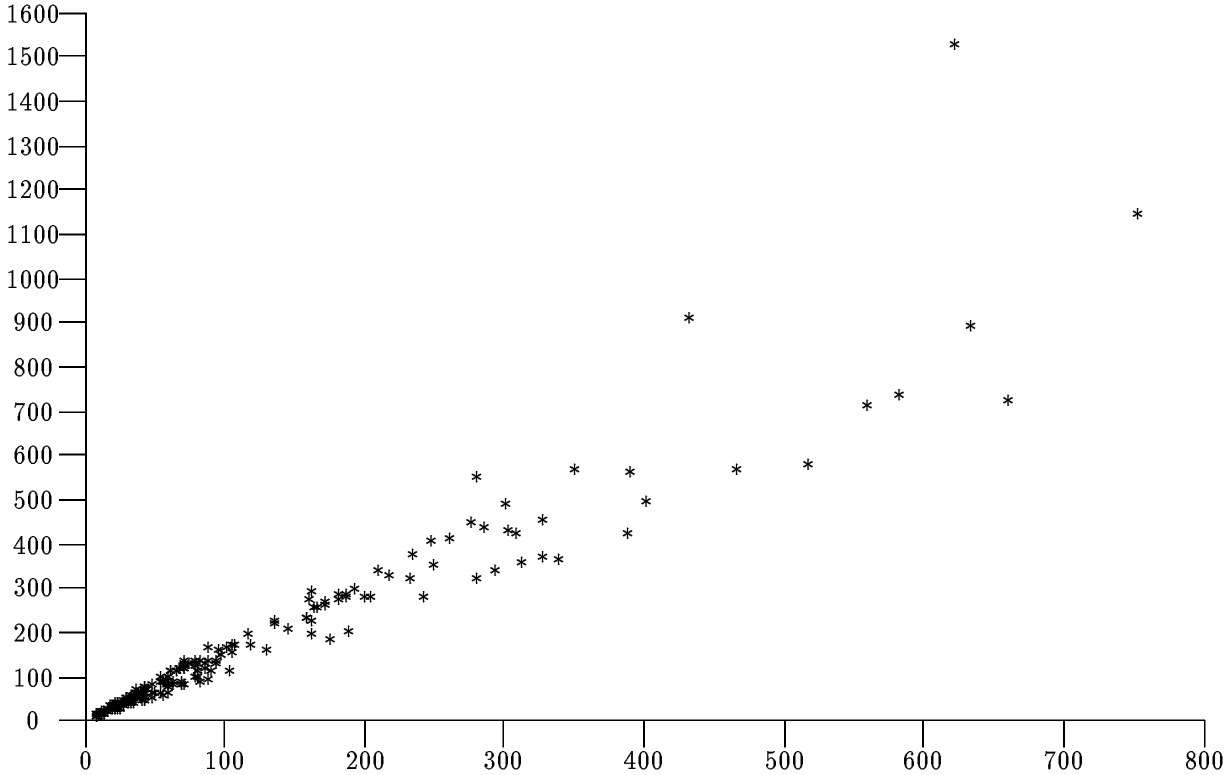

Figure 23. Size of control dependence graph vs. number of program statements.

for a procedure of only 12 statements, and 95% of the procedures had a ratio under 2.3. All but one of the remaining procedures contained fewer than 60 statements. Finally, the plot in Figure 23 shows that the size of the control dependence graph is linear in the size of the original program. The ratio of these sizes ranged from 0.6 to 2.4, which is very close to the range of ratios for dominance frontiers.

The ratio $avrgDF$ (defined by (7) in §5.1) measures the cost of placing ϕ-functions relative to the number of assignments in the resulting SSA form program. This ratio varied from 1 to 2, with median 1.3. There was no correlation with program size.

We also measured the expansion A_{tot}/A_{orig} in the number of assignments when translating to SSA form. This ratio varied from 1.3 to 3.8. Finally, we measured the expansion M_{tot}/M_{orig} in the number of mentions (assignments or uses) of variables when translating to SSA form. This ratio varied from 1.6 to 6.2. For both of these ratios, there was no correlation with program size.

9 Discussion

9.1 Summary of Algorithms and Time Bounds

The conversion to SSA form is done in three steps:

1. The *dominance frontier* mapping is constructed from the control flow graph CFG (§4.2). Let CFG have N nodes and E edges. Let DF be the mapping from nodes to their dominance frontiers. The time to compute the dominator tree and then the dominance frontiers in CFG is $O(E + \sum_X |DF(X)|)$.

2. Using the dominance frontiers, the locations of the ϕ-functions for each variable in the original program are determined (§5.1). Let A_{tot} be the *total* number of assignments to variables in the resulting program, where each ordinary assignment statement $LHS \leftarrow RHS$ contributes the length of the tuple LHS to A_{tot}, and each ϕ-function contributes 1 to A_{tot}. Placing ϕ-functions contributes $O(A_{tot} \times avrgDF)$ to the overall time, where $avrgDF$ is the weighted average (7) of the sizes $|DF(X)|$.

3. The variables are renamed (§5.2). Let M_{tot} be the total number of mentions of variables in the resulting program. Renaming contributes $O(M_{tot})$ to the overall time.

To state the time bounds in terms of fewer parameters, let the overall size R of the original program be the maximum of the relevant numbers: N nodes, E edges, A_{orig} original assignments to variables, and M_{orig} original uses of variables. In the worst case, $avrgDF = \Omega(N) = \Omega(R)$ and k ordinary assignments can require $\Omega(kR)$ insertions of ϕ-functions. Thus $A_{tot} = \Omega(R^2)$ at worst. In the worst case, a ϕ-function has $\Omega(R)$ operands. Thus $M_{tot} = \Omega(R^3)$ at worst. The one-parameter worst-case time bounds are thus $O(R^2)$ for finding dominance frontiers and $O(R^3)$ for translation to SSA form.

However, the data in §8 suggest that the entire translation to SSA form will be linear in practice. The dominance frontier of each node in CFG is small, as is the number of ϕ-functions added for each variable. In effect, $avrgDF$ is constant, $A_{tot} = O(A_{orig})$, and $M_{tot} = O(M_{orig})$. The entire translation process is effectively $O(R)$.

Control dependences are read off from the dominance frontiers in the reverse graph $RCFG$ (§6) in time $O(E + \text{size}(RDF))$. Since the size of RDF is the size of the output of the control dependence calculation, this algorithm is linear in the size of the output. The only quadratic behavior is caused by the output being $\Omega(R^2)$ in the worst case. The data in §8 suggest that the control dependence calculation is effectively $O(R)$.

9.2 Related Work

Minimal SSA form is a refinement of Shapiro and Saint's [45] notion of a pseudo-assignment. The *pseudo-assignment nodes* for V are exactly the nodes that need ϕ-functions for V. A closer precursor [22] of SSA form associated new names for V with pseudo-assignment nodes and inserted assignments from one new name to another. Without explicit ϕ-functions, however, it was difficult to manage the new names or reason about the flow of values.

Suppose the control flow graph CFG has N nodes and E edges for a program with Q variables. One algorithm [41] requires $O(E\alpha(E, N))$ bit vector operations (where each vector is of length Q) to find all the pseudo-assignments. A simpler algorithm [43] for reducible programs computes SSA form in time $O(E \times Q)$. With lengths of bit vectors taken into account, both of these algorithms are essentially $O(R^2)$ on programs of size R, and the simpler algorithm sometimes inserts extraneous ϕ-functions. The method presented here is $O(R^3)$ at worst, but §8 gives evidence that it is $O(R)$ in practice. The earlier $O(R^2)$ algorithms have no provision for running faster in typical cases; they appear to be intrinsically quadratic.

For CFG with N nodes and E edges, previous general control dependence algorithms [24] can take quadratic time in $(N + E)$. This analysis is based on the worst-case $\Omega(N)$ depth of the (post)dominator tree [24, p. 326]. Section 6 shows that (inverse) control dependences can be determined by computing dominance frontiers in the reverse graph $RCFG$. In general, our approach can also take quadratic time, but the only quadratic behavior is caused by the output being $\Omega(N^2)$ in the worst case. In particular, suppose a program is comprised only of straight-line code, `if-then-else`, and `while-do` constructs. By Corollary 2, our algorithm computes control dependences in linear time. We obtain a better time bound for such programs because our algorithm is based on dominance frontiers, whose sizes are not necessarily related to the depth of the dominator tree. For languages that offer only these constructs, control dependences can also be computed from the parse tree [28] in linear time, but our algorithm is more robust. It handles all cases in quadratic time and typical cases in linear time.

9.3 Conclusions

Previous work has shown that SSA form and control dependences can support powerful code optimization 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 for early encouragement and Fran Allen, Trina Avery, Julian Padget, Bob Paige, Tom Reps, Randy Scarborough, and Jin-Fan Shaw for various helpful comments. We are especially grateful to the referees, whose thorough reports led to many improvements in this paper.

References

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

[2] 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.

[3] J. R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformations.* PhD thesis, Dept. of Computer Sci., Rice U., Houston, TX, April 1983.

[4] J. R. Allen and S. Johnson. Compiling C for vectorization, parallelization and inline expansion. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, pages 241–249, June 1988. Published as *SIGPLAN Notices* Vol. 23, No. 7.

[5] 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*, pages 1–11, January 1988.

[6] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand driven interpretation of languages. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 257–271, June 1990. Published as *SIGPLAN Notices* Vol. 25, No. 6.

[7] J. B. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *Conf. Rec. Sixth ACM Symp. on Principles of Programming Languages*, pages 29–41, January 1979.

[8] J. M. Barth. An interprocedural data flow analysis algorithm. *Conf. Rec. Fourth ACM Symp. on Principles of Programming Languages*, pages 119–131, January 1977.

[9] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Trans. on Programming Languages and Systems*, 12(3):341–395, July 1990.

[10] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *Proc. SIGPLAN'86 Symp. on Compiler Construction*, pages 162–175, June 1986. Published as *SIGPLAN Notices* Vol. 21, No. 7.

[11] R. Cartwright and M. Felleisen. The semantics of program dependence. *Proc. SIGPLAN'89 Symp. on Compiler Construction*, pages 13–27, July 1989. Published as *SIGPLAN Notices* Vol. 24, No. 7.

[12] G. J. Chaitin. Register allocation and spilling via graph coloring. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, pages 98–105, June 1982. Published as *SIGPLAN Notices* Vol. 17, No. 6.

[13] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[14] D. R. Chase. Safety considerations for storage allocation optimizations. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, pages 1–10, June 1988. Published as *SIGPLAN Notices* Vol. 23, No. 7.

[15] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 296–310, June 1990. Published as *SIGPLAN Notices* Vol. 25, No. 6.

[16] J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conf. Rec. Eighteenth ACM Symp. on Principles of Programming Languages*, pages 55–66, January 1991.

[17] F. C. Chow. A portable machine-independent global optimizer – design and measurements. Technical Report 83-254 (PhD Thesis), Computer Systems Laboratory, Stanford U. Stanford, CA, December 1983.

[18] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12(4):501–536, October 1990.

[19] K. D. Cooper. *Interprocedural Data Flow Analysis in a Programming Environment*. PhD thesis, Dept. of Mathematical Sciences, Rice U., Houston, TX, 1983.

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

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

[22] 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.

[23] J. B. Dennis. First version of a data flow procedure language. Technical Report Comp. Struc. Group Memo 93 (MAC Tech. Memo 61), Massachussetts Institute of Technology, Cambridge, MA, May 1975.

[24] 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.

[25] R. Giegerich. A formal framework for the derivation of machine-specific optimizers. *ACM Trans. on Programming Languages and Systems*, 5(3):478–498, July 1983.

[26] 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.

[27] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *Proc. SIGPLAN'89 Symp. on Compiler Construction*, June 1989. Published as *SIGPLAN Notices* Vol. 24, No. 7.

[28] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Trans. on Programming Languages and Systems*, 11(3):345–387, July 1989.

[29] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[30] K. W. Kennedy. Global dead computation elimination. Technical Report SETL Newsletter No. 111, Courant Institute of Mathematical Sciences, New York U., New York, NY, August 1973.

[31] K. W. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*. Prentice-Hall, 1981.

[32] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley & Sons, 1978.

[33] J. R. Larus. Restructuring symbolic programs for concurrent execution on multiprocessors. Technical Report UCB/CSD 89/502, Computer Sci. Dept., U. of California at Berkeley, Berkeley, CA, May 1989.

[34] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, pages 21–34, July 1988. Published as *SIGPLAN Notices* Vol. 23, No. 7.

[35] T. Lengauer 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.

[36] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis*. Prentice-Hall, 1981.

[37] E. W. Myers. A precise interprocedural data flow algorithm. *Conf. Rec. Eighth ACM Symp. on Principles of Programming Languages*, pages 219–230, January 1981.

[38] K. J. Ottenstein. *Data-flow Graphs as an Intermediate Form.* PhD thesis, Dept. of Computer Sci., Purdue U., W. Lafayette, IN, August 1978.

[39] L. Pointer. Perfect report: 1. Technical Report CSRD 896, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, July 1989.

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

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

[42] B. K. Rosen. Data flow analysis for procedural languages. *J. ACM*, 26(2):322–344, April 1979.

[43] 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*, pages 12–27, January 1988.

[44] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 285–293, January 1988.

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

[46] 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, NY, NY, 1976.

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

[48] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. on Software Engineering*, SE-1(3):270–285, September 1975.

[49] 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.

[50] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 1991. To appear.

[51] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Sci., U. of Illinois at Urbana-Champaign, Urbana IL, 1982.

[52] W. Yang, S. Horwitz, and T. Reps. Detecting program components with equivalent behaviors. Technical Report 840, Dept. of Computer Sci., U. of Wisconsin at Madison, Madison, WI, April 1989.