A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs

VUGRANAM C. SREEDHAR
Hewlett-Packard Company
and
GUANG R. GAO
University of Delaware
and
YONG-FONG LEE
Intel Corporation

In this article, we present a new framework for elimination-based exhaustive and incremental data flow analysis using the DJ graph representation of a program. Unlike previous approaches to elimination-based incremental data flow analysis, our approach can handle arbitrary structural and nonstructural changes to program flowgraphs, including irreducibility. We show how our approach is related to dominance frontiers, and we exploit this relationship to establish the complexity of our exhaustive analysis and to aid the design of our incremental analysis.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*; E.1 [**Data**]: Data Structures—*graphs; trees*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: DJ graphs, exhaustive and incremental data flow analysis, irreducible flowgraphs, reducible flowgraphs, Tarjan's interval

1. INTRODUCTION

Traditional elimination methods consist of three steps [Ryder and Paull 1986]: (1) reducing the flowgraph to a single node, (2) eliminating variables in the data flow equations by substitution, and (3) once the solution to the single node is determined, propagating the solution to other nodes to determine their respective

This research was supported by the National Sciences and Engineering Research Council (NSERC) and the Canadian Centers of Excellence (IRIS). V. C. Sreedhar's work on this article was done when he was at McGill University.
A preliminary version of this article has appeared in the Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation.
Authors' addresses: V. C. Sreedhar, Hewlett-Packard Company, 11000 Wolfe Road, Cupertino, CA 95014; G. R. Gao, Department of Electrical and Computer Engineering, University of Delaware, 140 Evans Hall, Newark, DE 19716; Y. Lee, Intel Corporation, RN6-18, 2200 Mission College Blvd., Santa Clara, CA 95052.

solutions. Elimination methods for data flow analysis have been studied by many authors [Allen and Cocke 1976; Burke 1990; Graham and Wegman 1976; Tarjan 1981; Ullman 1973]. An excellent survey can be found in Ryder and Paull [1986].

In this article we present a new framework for both *exhaustive* and *incremental* data flow analysis based on elimination methods. At the heart of our framework is the DJ graph representation [Sreedhar 1995; Sreedhar and Gao 1995; 1996]. A DJ graph is just the dominator tree of a flowgraph augmented with *join* edges from the flowgraph (see Figure 1). A flowgraph edge is a join edge (J edge) if it has no corresponding edge in the dominator tree. A dominator tree edge is called a D edge. Unlike previous elimination methods that typically reduce a flowgraph to a single node, our approach only eliminates J edges from a DJ graph in a bottom-up fashion on the dominator tree (which may be compressed in the process). It performs variable elimination either eagerly or in a delayed manner. At the end of the bottom-up elimination phase, all the J edges are eliminated, and the data flow solution at every node, except for the root, is expressed only in terms of its parent's solution in the dominator tree. Once we determine the solution for the root node, we propagate this information on the dominator tree in a top-down fashion to compute the solution for every other node.

Incremental analyses have many applications in program development environments (e.g., Matlab) and aggressive optimizing compilers, especially those performing interprocedural optimizations. Unlike previous elimination-based incremental methods, our approach can handle arbitrary program changes, including structural changes that cause irreducibility. A novel aspect of our framework is that we found a surprisingly simple relationship between (iterated) dominance frontiers and incremental data flow analysis. Since we use dominance frontiers in our incremental analysis, we also show how to incrementally compute the dominance frontier relation.

Finally, we will show how to extend our framework to an interprocedural setting similar to that proposed by Burke [1990].

The next section introduces some useful notation and definitions. Sections 3 to 9 describe our exhaustive analysis, and Sections 10 to 14 describe our incremental analysis. Section 16 shows how to extend our framework to an interprocedural setting. Finally, Section 17 compares our approach with other related work.

2. BACKGROUND AND NOTATION

A **flowgraph** is a directed graph $G = (N, E, \text{START})$, where N is the set of nodes; E is the set of control flow edges; and $\text{START} \in N$ is the distinguished start node with no incoming edges. A flowgraph need not be connected; that is, some nodes may not be reachable from the START node. We define the reachable subgraph $\mathsf{REACH}(\text{START})$ to be a subgraph of G such that all nodes in $\mathsf{REACH}(\text{START})$ are reachable from START.

If S is a set, we use $|S|$ to represent the cardinality of S.

In the reachable subgraph $\mathsf{REACH}(\text{START})$, a node x **dominates** another node y, denoted as x **dom** y, if and only if all paths from START to y pass through x. We write by x **!dom** y when x does not dominate y. If x **dom** y and $x \neq y$, then x **strictly dominates** y, and it is denoted by x **stdom** y. A node x **immediately dominates** another node y, denoted by $x = idom(y)$, if x **stdom** y

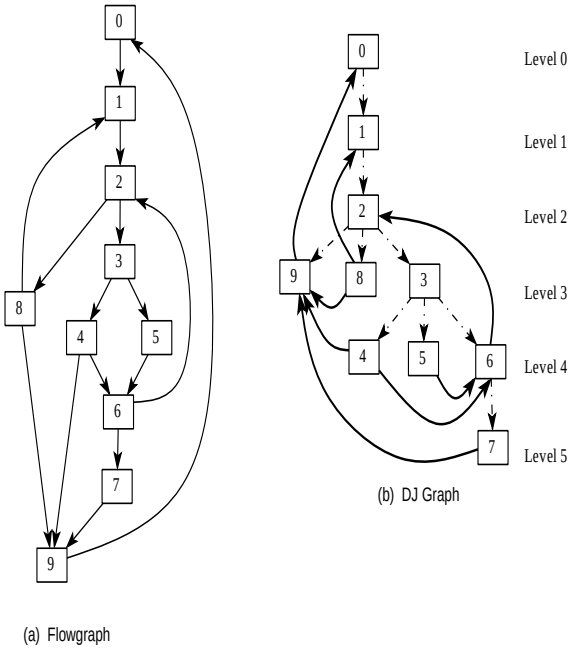

Fig. 1. A flowgraph and its DJ graph.

and if there is no other node z such that x **stdom** z **stdom** y. Given the notion of the immediate dominance relation, we define the **dominator tree** of G as a rooted tree $T = (N_d, E_d, \text{START}_d)$, where $N_d = N$, an edge $x_d \rightarrow y_d \in E_d$ if and only if $x = idom(y)$ in G, and START_d is the root node.

For each node x in the dominator tree, we associate with it a *level number*, denoted as $x.level$, which is the depth of the node from the root of the tree.

The DJ graph of a flowgraph consists of the same set of nodes as in the flowgraph and two types of edges called D edges and J edges. D edges are dominator tree edges. An edge $x \rightarrow y$ in a flowgraph is a *join edge* (or *J edge*) if $x \neq idom(y)$. If $x \rightarrow y$ is a J edge, then $x.level \geq y.level$ [Sreedhar 1995; Sreedhar and Gao 1996].

To construct the DJ graph for a flowgraph, first build the dominator tree of the flowgraph. Then, for each J edge $x \rightarrow y$ in the flowgraph, insert a corresponding edge $x \rightarrow y$ into the dominator tree. Note that the dominance relation of a DJ graph is exactly captured by the underlying dominator tree of the DJ graph. Figure 1 shows an example flowgraph and its DJ graph.

The **dominance frontier** $DF(x)$ of a node x is the set of all nodes y such that x dominates a predecessor of y but does not strictly dominate y [Cytron et al. 1991]. It can be extended to mean a set of edges $z \rightarrow y$ such that x dominates z, but does not strictly dominate y.

We can extend the definition of dominance frontier to a set S of nodes:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

We define **iterated dominance frontier** $IDF(S)$ for a set S of nodes as the limit of the increasing sequence:

$$IDF_1(S) = DF(S), \tag{1}$$
$$IDF_{i+1}(S) = DF(S \cup IDF_i(S)) \tag{2}$$

Throughout this article, all flowgraph properties such as dominance, dominance frontiers, etc., are defined only for the reachable subgraph REACH(START). Therefore, whenever we use the phrase "a flowgraph and its dominator tree" we are referring to "the reachable subgraph REACH(START) and its dominator tree." This convention applies to other properties as well (including data flow properties and DJ graphs).

3. EXHAUSTIVE DATA FLOW ANALYSIS: BASICS

Data flow analysis is a process of estimating facts about a program statically. These facts, or data flow information, can be modeled by elements of a lattice $\mathcal{L}$. Associated with each flowgraph node x is a flow function f_x that maps input information to output information [Kildall 1973; Marlowe 1989; Marlowe and Ryder 1990b].[1]

Let $I_x \in \mathcal{L}$ be the information at the entry of a node x, and let $O_x \in \mathcal{L}$ be the information at the exit of the node. Then the input-output relationship can be expressed as

$$O_x = f_x(I_x).$$

We can rewrite this equation as follows for forward union problems:[2]

$$O_x = f_x(I_x) = P_x I_x + G_x \tag{3}$$

where $P_x, G_x \in \mathcal{L}$, $+$ is the union operation, and juxtaposition is the intersection operation.[3] We can interpret the above equation as follows: the *O*utput flow information at a node's exit is either (1) what is *G*enerated within the node or (2) what arrives at its *I*nput and is *P*reserved through the node.

We need another set of equations to relate the output information at a node y to the input information at x when an edge $y \to x$ exists. The input information I_x is the merge of all the output information O_y of nodes y in $Pred_f(x)$, which is the set of predecessors of x on the flowgraph, i.e.,

$$I_x = \bigwedge_{y \in Pred_f(x)} O_y \tag{4}$$

$\bigwedge$ is usually a union or intersection operation, depending upon the data flow problem being solved. Combining Eq's. (3) and (4), we obtain the following equation,

[1] For some problems, it is more convenient to associate flow functions with flowgraph edges instead of nodes (see Section 16).

[2] Throughout this article we will assume a forward union problem to illustrate our approach. Forward intersection problems have a dual formulation, and backward problems can be solved on the "reverse" flowgraph.

[3] In general, G_x may not be a constant [Marlowe 1989; Marlowe and Ryder 1990b]. It can also depend on I_x. That is, the information that is generated at a node depends on (1) the local data flow information at the node and (2) the input data flow information to the node.

denoted by H_x, for each $x \in N$:

$$H_x : \quad O_x = f_x(\bigwedge_{y \in Pred_f(x)} O_y) \tag{5}$$

$$H_x : \quad O_x = P_x(\bigwedge_{y \in Pred_f(x)} O_y) + G_x \tag{6}$$

Since there is one equation for each node, we have a total of $|N|$ equations. Notice that the first variation (Eq. (5)) is more general than the second (Eq. (6)), since for some data flow problems the information generated within a node x is not independent of I_x [Marlowe 1989; Marlowe and Ryder 1990b]. We use the term **output variable** to name the variable O_x that appears on the left-hand side (LHS) of equation H_x. Any variable appearing on the right-hand side (RHS) of the equation is called an **input variable**. Furthermore, P_x and G_x are called the **parameters** of the equation.

Given any two equations H_x and H_y, we say that H_x **depends on** H_y if the output variable of H_y appears on the RHS of H_x. That is, we need the solution of O_y in order to compute the solution of O_x. Also, if H_x depends on H_y, then there is an edge from y to x in the corresponding flowgraph.

Next we discuss the concept of *variable elimination*, which is fundamental to all elimination methods. Consider the following two equations:

$$H_y : \quad O_y = P_y(\bigwedge_{z \in Pred_f(y)} O_z) + G_y$$

$$H_x : \quad O_x = P_x O_y + G_x.$$

In the case above, equation H_x depends only on H_y. This also means that node x has only one incoming edge $y \rightarrow x$ in the corresponding flowgraph. To eliminate the variable O_y from the RHS of H_x, we can replace it with the RHS of H_y. The resulting H_x equation thus becomes

$$H_x : \quad O_x = P_x(P_y(\bigwedge_{z \in Pred_f(y)} O_z) + G_y) + G_x.$$

Here we eliminate the dependence of H_x on H_y, but introduce dependences from H_z to H_x for each predecessor z of y. In the corresponding flowgraph, we also eliminate the edge $y \rightarrow x$ and introduce a new edge $z \rightarrow x$ for each $z \in Pred_f(y)$.

For the more general variation in Eq. (5), variable elimination corresponds to function composition. To illustrate this, let us consider the following equations:

$$H_y : \quad O_y = f_y(\bigwedge_{z \in Pred_f(y)} O_z)$$

$$H_x : \quad O_x = f_x(O_y)$$

After eliminating O_y in H_x we get

$$H_x : \quad O_x = f_x(f_y(\bigwedge_{z \in Pred_f(y)} O_z)).$$

Table I. Data Flow Equations for the Example

x	f_x	O_x	$I_x = \wedge_{y \in Pred_f(x)} O_y$	$f_x(I_x) = O_x = P_x I_x + G_x$
0	f_0	O_0	O_9	$P_0 O_9 + G_0$
1	f_1	O_1	$O_0 \wedge O_8$	$P_1(O_0 \wedge O_8) + G_1$
2	f_2	O_2	$O_1 \wedge O_6$	$P_2(O_1 \wedge O_6) + G_2$
3	f_3	O_3	O_2	$P_3 O_2 + G_3$
4	f_4	O_4	O_3	$P_4 O_3 + G_4$
5	f_5	O_5	O_3	$P_5 O_3 + G_5$
6	f_6	O_6	$O_4 \wedge O_5$	$P_6(O_4 \wedge O_5) + G_6$
7	f_7	O_7	O_6	$P_7 O_6 + G_7$
8	f_8	O_8	O_2	$P_8 O_2 + G_8$
9	f_9	O_9	$(O_4 \wedge O_7 \wedge O_8)$	$P_9(O_4 \wedge O_7 \wedge O_8) + G_9$

Finally, we define the closure operation for recursive equations. If at any node y, the data flow equation is of the form

$$H_y : \qquad O_y \ = \ mO_y + k = f_y(O_y), \tag{7}$$

where m and k are terms that do not contain O_y, then the closure operation of this equation is

$$H_y : \qquad O_y \ = \ f_y^*(O_y) \tag{8}$$

where f^* is a fixed-point operator of the recursive equation. Assume that the flow function associated with each node is monotone and that the lattice does not contain infinite descending chains (every chain in the lattice is finite) [Marlowe and Ryder 1990b]. Then the fixed point finally reached will be the maximal fixed point with respect to the function lattice $(\mathcal{L} \rightarrow \mathcal{L})$.

For many of the classical problems, such as Reaching Definitions and Available Expressions, f^* can be computed quickly, essentially in constant time [Marlowe and Ryder 1990b]. Ryder and Paull call such fixed-point operations the loop-breaking rules [Ryder and Paull 1986].

Example. The data flow equation for each node in Figure 1 is summarized in Table I.

To solve a system of data flow equations, we propose two methods in our exhaustive analysis: eager elimination (Section 4) and delayed elimination (Section 6). Both methods consist of two phases: (1) bottom-up DJ graph reduction and variable elimination, called the **elimination phase**, and (2) top-down propagation of data flow solutions, called the **propagation phase**.

4. EAGER ELIMINATION METHOD

Our eager elimination method uses the E1 and E2 rules, together called the $\mathcal{E}$-*rules*, to reduce a DJ graph in a bottom-up fashion. Figure 2 shows the main driver for our framework. The $\mathcal{E}$-*rules* are always applied to a J edge $y \rightarrow z$ such that y is a nonjoin node, and such that there is no J edge with its source node at a level greater than $y.level$ in the reduced DJ graph. In this article we adopt the following relaxed definition of nonjoin nodes: A node y is a **nonjoin** node if and only if $Pred(y)$ contains no nodes other than y and $idom(y)$.[4] We use $\mathcal{G}^0$ to represent

[4]That is, a self-loop $y \rightarrow y$ does not prevent y from being a nonjoin node.

```
MainDFA()
{
1:    Initialize PriorityList;
2:    for i = NumLevel − 1 downto 1 do
3:       ReduceLevel(i) ;
4:       if(PriorityList[i] is not empty) then
5:          CollapseIrreducible(i) ;
6:       endif
7:    endfor
8:    DomTDPropagate() ;
}
Procedure ReduceLevel(i)
{
9:    while((y = GetNode(i)) ≠ NULL) do
10:      foreach outgoing J edge y → z do
11:         if(z == y) then /* self-loop */
12:            Eager1(y → y) ;
13:         else
14:            if(y.level == z.level) then
15:               Eager2a(y → z) ;
16:            else
17:               Eager2b(y → z) ;
18:            endif
19:         endif
20:      endfor
21:   endwhile
}
```

Fig. 2. Algorithm outline for our framework.

the original (zeroth) DJ graph. Each application of the $\mathcal{E}$-*rules* transforms some reduced DJ graph $\mathcal{G}^i$ to $\mathcal{G}^{i+1}$ until the resulting DJ graph "degenerates" into its dominator tree.[5] Figure 3 gives a graphical illustration of these rules.

Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. We denote by $\mathcal{G}^{i+1}$ the DJ graph obtained by applying a reduction rule to the DJ graph $\mathcal{G}^i$. We denote by $E1(< \mathcal{G}^i, N, E, y \xrightarrow{J} y >)$ the application of E1 graph reduction rule to the J edge $y \xrightarrow{J} y$ in the DJ graph $\mathcal{G}^i = (N, E)$ (we use similar notation for E2 rules). We denote by $E1(< O_y = f_y(O_y) >)$ the application of E1 variable elimination rule for the equation at node y. We denote by $\Longrightarrow$ the transformation operator of the DJ graph. Finally, $-$ denotes the set difference operator.

The E1 rule eliminates a self-loop, and computes the closure of a recursive equation.

Definition 4.1 (*E1 Rule*). Let y be a nonjoin node such that the self-loop $y \rightarrow y$ exists. Let $O_y = f_y(O_y)$ be the flow equation H_y at node y. Then

(1) *Graph reduction*:

$$E1(< \mathcal{G}^i, N, E, y \xrightarrow{J} y >) \Longrightarrow < \mathcal{G}^{i+1}, N, E - \{y \xrightarrow{J} y\} >$$

[5]Note that the dominator tree of the flow graph is same as the dominator tree of the DJ graph.

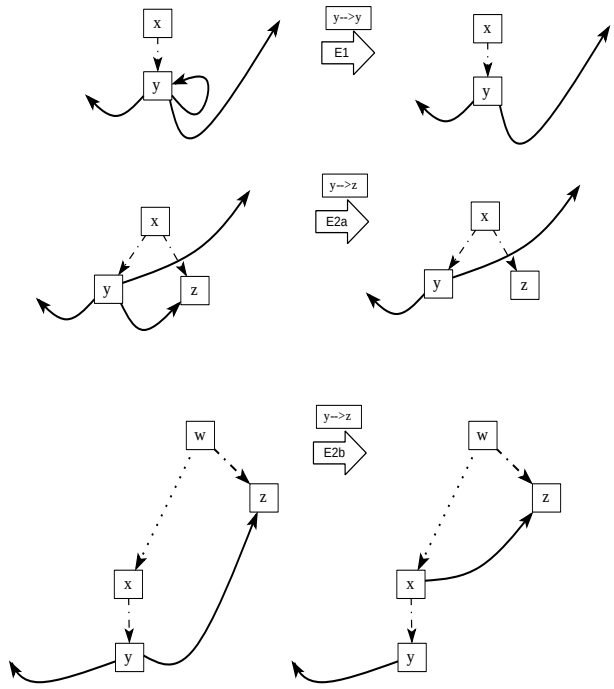

Fig. 3. Graph reduction due to the $\mathcal{E}$-*rules*.

(2) *Variable elimination*:

$$E1(< O_y = f_y(O_y) >) \Longrightarrow < O_y = f_y^*(O_y) >$$

An E2 rule is applied to a J edge $y \rightarrow z$ only if y is a nonjoin node and $y \rightarrow y$ does not exist. Recall that $y.level \geq z.level$ for any J edge $y \rightarrow z$. We distinguish between two types of E2 rules depending on the levels of y and z. If $y.level = z.level$, we apply E2a; otherwise we apply E2b.

Definition 4.2 (*E2 Rules*). Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a nonjoin node such that $y \rightarrow y$ does not exist. Let $y \rightarrow z$ be a J edge, and let $x = idom(y)$. Let $O_y = kO_x + m$ be the equation H_y at node y, such that the parameters k and m do not contain any variables. Finally, let $O_z = aO_y + b$ be the equation H_z at node z, where a and b do not contain the variable O_y. There are two cases:

E2a Rule. If $y.level = z.level$, then

(1) *Graph reduction:*

$$E2a(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \Longrightarrow < \mathcal{G}^{i+1}, N, E - \{y \xrightarrow{J} z\} >$$

(2) *Variable elimination:*

$$E2a(< O_z = aO_y + b >) \Longrightarrow < O_z = a(kO_x + m) + b >$$

```
Procedure Eager1(y → y)
{
22:   Compute the closure H_y : O_y = f*_y(O_y).
23:   Delete the edge y -J-> y ;
}
Procedure Eager2a(y → z)
{
24:   Eliminate O_y in H_z by replacing it with the RHS of H_y.
25:   Delete the edge y -J-> z ;
26:   if(z becomes a nonjoin node ) then
27:      Put z at the head of PriorityList[z.level] list ;
28:   endif
}
Procedure Eager2b(y → z)
{
29:   Eliminate O_y in H_z by replacing it with the RHS of H_y.
30:   x = idom(y) ;
31:   Delete the edge y -J-> z ;
32:   if(x → z does not exist) then
33:      Insert a new J edge x -J-> z ;
34:   endif
}
```

Fig. 4. Procedures for $\mathcal{E}$-*rules*.

E2b Rule. If $y.level \neq z.level$, then

(1) *Graph reduction:*

$$E2b(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \Longrightarrow < \mathcal{G}^{i+1}, N, (E - \{y \xrightarrow{J} z\}) \cup \{x \xrightarrow{J} z\} > \text{ [6]}$$

(2) *Variable elimination:*

$$E2b(< O_z = aO_y + b >) \Longrightarrow < O_z = a(kO_x + m) + b >$$

The newly inserted edge $idom(y) \rightarrow z$ in E2b is called a **derived edge** of $y \rightarrow z$. An important point to note is that before an E2 rule is applied to an edge $y \rightarrow z$, we first eliminate the self-loop $y \rightarrow y$, if it exists, by applying the E1 rule. This will ensure that there is no self-loop at node y when E2 rules are applied to outgoing edges from y. The difference between E2a and E2b is minor, but this distinction is useful when we discuss delayed elimination and when we handle irreducibility.

The algorithm outline for eager elimination is given in Figure 2. $NumLevel$ is the total number of levels in the DJ graph. $PriorityList[i]$ maintains an ordered list of nodes at level i to be processed by procedure **ReduceLevel**(i), such that nonjoin nodes always appear earlier than join nodes. We first initialize $PriorityList$ to contain all the nodes in the DJ graph.

MainDFA() calls **ReduceLevel**() in a bottom-up fashion. When the procedure **ReduceLevel**(i) terminates, $PriorityList[i]$ will be empty if the flowgraph is reducible. For an irreducible flowgraph, procedure **CollapseIrreducible**(i) is

[6] We do not insert $x \rightarrow z$ in $\mathcal{G}^{i+1}$ if it is already present in $\mathcal{G}^i$.

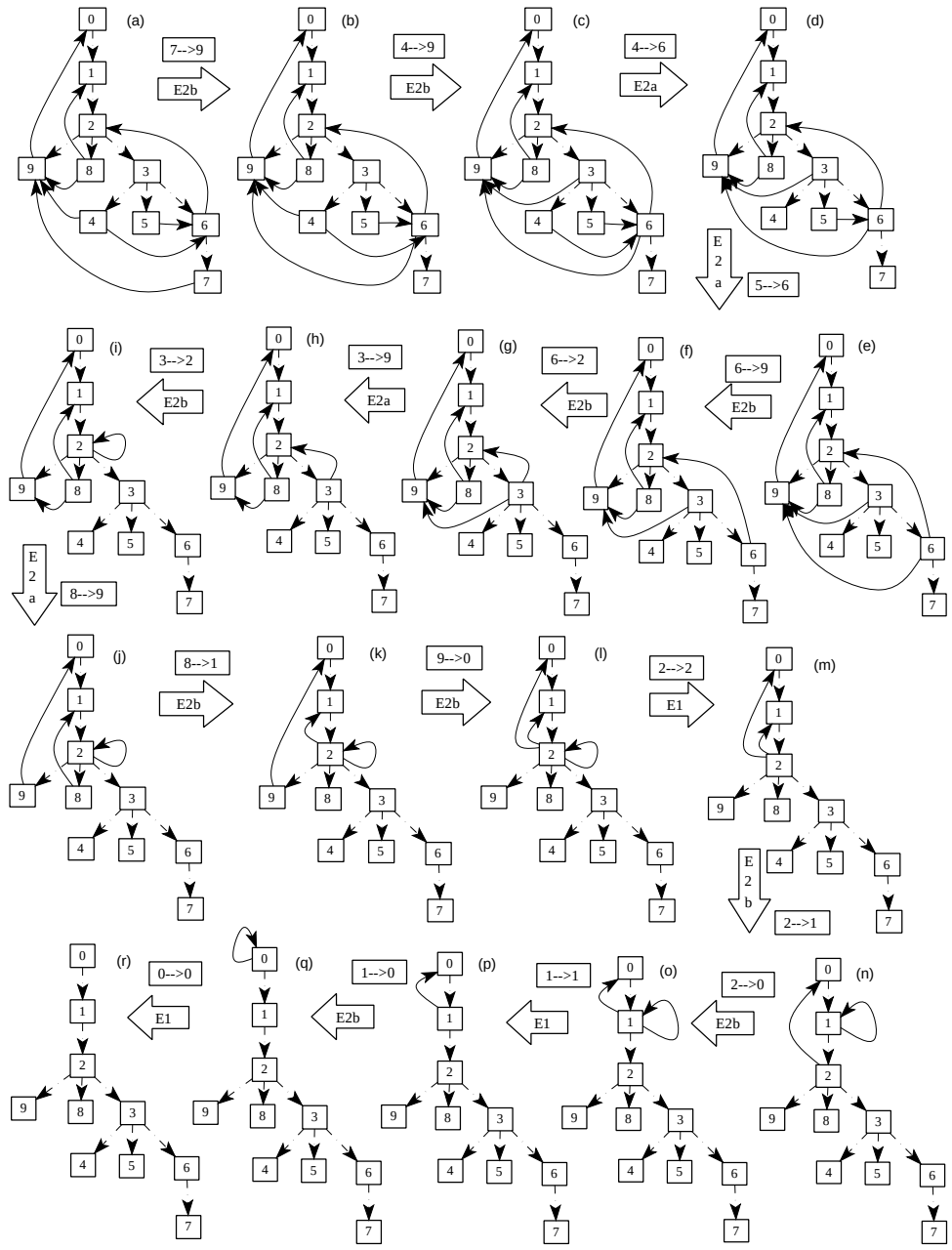

Fig. 5. DJ graph reduction with eager elimination (e.g., from (a) to (b), we apply the E2b rule to J edge $7 \rightarrow 9$).

Table II. Partial Trace of Variable Elimination (the corresponding DJ graphs in Figure 5)

	Rule	$y \rightarrow z$	$\mathcal{G}^i\dagger$	$\mathcal{G}^{i+1}\dagger$	O_z^{i+1}
1	E2b	$7 \rightarrow 9$	(a)	(b)	$P_9O_4 + P_9O_8 + P_9P_7O_6 + P_9G_7 + G_9$
2	E2b	$4 \rightarrow 9$	(b)	(c)	$P_9P_4O_3 + P_9G_4 + P_9O_8 + P_9P_7O_6 + P_9G_7 + G_9$
3	E2a	$4 \rightarrow 6$	(c)	(d)	$P_6P_4O_3 + P_6G_4 + P_6O_5 + G_6$
4	E2a	$5 \rightarrow 6$	(d)	(e)	$(P_6P_4 + P_6P_5)O_3 + P_6G_4 + P_6G_5 + G_6$
5	E2b	$6 \rightarrow 9$	(e)	(f)	$(P_9P_4 + P_9P_7P_6P_4 + P_9P_7P_6P_5)O_3 + P_9O_8 + P_9G_4 + P_9P_7P_6G_4 + P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + G_9$

invoked to handle irreducible regions at level i; details are given in Section 8. At the end of the elimination phase, the data flow solution at every node (except for the root) depends only on the solution of its immediate dominator. At this point, **MainDFA()** calls procedure **DomTDPropagate()** to propagate solutions down the dominator tree and finishes the entire data flow solution process. Notice that the rules applied in a bottom-up fashion on the DJ graph. This ordering ensures that when a rule is applied to a J edge $y \rightarrow z$, there are no more J edges at levels below $y.level$, and the equations at all nodes below $y.level$ will depend only on the equation of their immediate dominator node.

Procedure **ReduceLevel**(i) eliminates J edges whose source nodes are at level i (for a reducible flowgraph) by applying the $\mathcal{E}$-*rules*. We ensure that if the self-loop edge $y \rightarrow y$ exists, it will be the first outgoing edge from node y. Consequently, when an E2 rule is applied to an outgoing J edge from y, there will be no self-loop at y.

Procedure **GetNode**(i) returns a nonjoin node at level i whenever one exists. Otherwise, it returns $NULL$ under two circumstances: (1) $PriorityList[i]$ is empty and (2) $PriorityList[i]$ is not empty, but all its remaining nodes are join nodes. The second case indicates the existence of irreducibility.

Procedures **Eager1**(), **Eager2a**(), and **Eager2b**() implement the E1, E2a, and E2b rules, respectively, and their associated variable eliminations (see Figure 4).

Although the worst-case time complexity of eager elimination is $O(|E| \times |N|)$, we will demonstrate in Section 9 that the algorithm is expected to behave linearly in practice.

Example. Consider the example flowgraph shown in Figure 1. Figure 5 gives a trace of the graph reduction process. Assume a forward data flow problem with union as the meet operation. Table II gives a partial trace of the variable elimination corresponding to the graph reduction shown in Figure 5.

5. CORRECTNESS AND COMPLEXITY OF EAGER ELIMINATION

In this section we prove the correctness of the eager elimination method and analyze its time complexity.

5.1 Correctness

The main theorem which establishes the correctness of the eager elimination method is Theorem 5.1.6. To prove the main theorem,

—we have to show that the $\mathcal{E}$-*rules* when applied in a bottom-up fashion reduce a reducible DJ graph to its dominator tree and

—we have to show that variable elimination and top-down propagation are correct.

We first define the reducibility of DJ graphs as well as flowgraphs [Hecht and Ullman 1974].

Definition 5.1.1. A DJ graph $\mathcal{G}$ is reducible if and only if we can partition the edges into two disjoint groups, called the **forward** edges and **back** edges, with the following two properties:

(1) the forward edges form an acyclic graph in which every node can be reached from the START node of $\mathcal{G}$.
(2) the back edges consist only of edges whose destination nodes dominate their source nodes.

We can easily see that a DJ graph is reducible if and only if the corresponding flowgraph is also reducible [Sreedhar 1995].

We begin the proof chain by showing that if $\mathcal{G}$ is reducible, there always exists at least one nonjoin node at the maximum level.

LEMMA 5.1.2. *Let $\mathcal{G}$ be a DJ graph, and let k be a level number such that there are no J edges originating at levels greater than k. If $\mathcal{G}$ is reducible then there exists at least one nonjoin node at level k.*

PROOF. Suppose to the contrary that there were no nonjoin nodes at level k of the DJ graph. Let M be the set of nodes at this level. Without loss of generality we will remove all the self-loops in M.

We then show that there must exist a nontrivial cycle at level k, concluding that the DJ graph is irreducible. Since every node in M is not a nonjoin node, it must have at least one incoming J edge, and the source node of this J edge must be at k level too, according to the property of DJ graphs. (Remember that every node can have at most one incoming D edge.) If this is the case we can traverse backward over the J edges and still stay at the same level. Since there are only a finite number of nodes in M, we will eventually visit a node twice by this backward traversal. This implies the existence of a simple cycle consisting only of nodes from M. Notice that we have removed self-loops. By Definition 5.1.1, the DJ graph is not reducible. But this contradicts the assumption that it is reducible. Therefore the lemma is true, and there must exist a nonjoin node at level k. □

The next lemma shows that applying *$\mathcal{E}$-rules* preserves reducibility of the DJ graph.

LEMMA 5.1.3. *Let $\mathcal{G}$ be a reducible DJ graph. Let one of $\mathcal{E}$-rules be applied to $\mathcal{G}$, resulting in $\mathcal{G}^1$. Then $\mathcal{G}^1$ is also reducible.*

PROOF. The proof of the lemma is based on the following observation. In the three *$\mathcal{E}$-rules* we delete an edge, but only in E2b rule we also insert an edge.

First we consider the deletion case. From Definition 5.1.1 we know that if $\mathcal{G}$ is reducible we can partition the edges into two sets, called forward edges and back edges. The source node of a back edge dominates its destination node. It is obvious to see that deleting an edge from $\mathcal{G}$ does not violate the Definition 5.1.1, i.e., $\mathcal{G}^1$ will still be reducible.

Now for the insertion case we notice that a new edge is inserted only when E2b rule is applied. By applying E2b rule we delete an edge $y \rightarrow z$ and insert $x \rightarrow z$, where $x = idom(y)$. We will first show that if $y \rightarrow z$ is a forward edge so is $x \rightarrow z$, and if $y \rightarrow z$ is a back edge so is $x \rightarrow z$. Let $y \rightarrow z$ be a back edge. From Definition 5.1.1 we know that z **dom** y. Now since $x = idom(y)$ and $y.level > z.level$, z should also dominate x. Therefore $x \rightarrow z$ is a back edge too.

We can similarly argue for the case when $y \rightarrow z$ is a forward edge. Let $y \rightarrow z$ be a forward edge. From Definition 5.1.1 we know that z !**dom** y. Now since $x = idom(y)$ and $y.level > z.level$, z will also not dominate x. Therefore $x \rightarrow z$ is also a forward edge.

Since by adding $x \rightarrow z$ we never violate the definition of reducibility, $\mathcal{G}^1$ should also be reducible. □

Based on Lemmas 5.1.2 and 5.1.3, we can always apply *$\mathcal{E}$-rules* when there are still J edges in the DJ graph. Furthermore, at each stage of the reduction the rules preserve reducibility. Given these we will next show that our bottom-up reduction will always reduce a reducible DJ graph to its dominator tree, and any variable elimination performed during the reduction is correct. We state these results in the following lemma.

LEMMA 5.1.4. *In eager elimination, when* **ReduceLevel**(i) *is called at step* **3** *and the call terminates, then*

(*1*) *all the J edges whose source nodes are at levels greater than or equal to i are eliminated and*

(*2*) *the RHS of the flow equation at each node y, whose level number is greater than or equal to i, contains only the flow variable $O_{idom(y)}$.*

PROOF. First of all observe that when one of *$\mathcal{E}$-rules* is applied to a J edge $y \rightarrow z$, the edge $y \rightarrow z$ is eliminated (and a new edge $idom(y) \rightarrow z$ is possibly inserted). Also, in the procedure **ReduceLevel**(i) we apply the *$\mathcal{E}$-rules* for each outgoing edge of the nonjoin node returned by **GetNode**(i) (steps **9** and **10**). Finally, when one of *$\mathcal{E}$-rules* is applied to the J edge $y \rightarrow z$, O_y is eliminated in H_z, (either by substitution, as in E2a rule and E2b rule, or by computing the closure, as in E1 rule).

We will prove the rest of the lemma using induction on the loop index i at step **2**.

Base Case (Loop Index $i = NumLevel-1$). When **ReduceLevel**($NumLevel-1$) is called we apply the appropriate *$\mathcal{E}$-rules* for each J edge $y \rightarrow z$, such y is a nonjoin node at level $NumLevel - 1$ (step **10**). Since *$\mathcal{E}$-rules* eliminates J edges $y \rightarrow z$, and eliminates O_y in H_z, the two assertions of the lemma are true for the base case.

Induction Hypothesis. Assume that the two assertions of the lemma are true for some loop index $i = k+1$ less than the maximum level. This means that all J edges whose source nodes at levels greater than or equal to $k+1$ are eliminated, and the flow equation of each node y with $y.level \geq k$ contains, on their RHS's, only the flow variable $O_{idom(y)}$.

Induction Step. Given the hypothesis, we will show that the two assertions of the lemma are true for loop index $i = k$. From Lemma 5.1.3 we know that $\mathcal{E}$-*rules* preserve reducibility, and from Lemma 5.1.2 we know that there exists at least one nonjoin node at level k. Let one of $\mathcal{E}$-*rules* be applied to some $\mathcal{G}^j$ resulting in $\mathcal{G}^{j+1}$. Assume that the maximum level of $\mathcal{G}^j$ and $\mathcal{G}^{j+1}$ are the same. Since $\mathcal{G}^{j+1}$ is also reducible, there exists a nonjoin node at the maximum level (according to Lemma 5.1.2 again). Therefore, **GetNode**(k) when called at step **9** with the current level k, will return a nonjoin node. The procedure **GetNode**(k) will return $NULL$ only when there are no more nonjoin nodes at level k. Since the given DJ graph is assumed to be reducible, this situation can happen only when all the outgoing J edges from level k are eliminated. Hence when **ReduceLevel**(k) terminates, all the outgoing J edges from level k are eliminated.

Next to see that variables are appropriately eliminated, we examine the $\mathcal{E}$-*rules*. We know that each of the $\mathcal{E}$-*rules*, when applied to $y \rightarrow z$ will eliminate O_y from H_z by substituting it with a linear function of $O_{idom(y)}$. Therefore, when the procedure **ReduceLevel**(k) returns, two assertions of the lemma will be true. □

In Lemma 5.1.5, we will argue that when the **DomTDPropagate()** is invoked at step **8**, the flow equation at each node depends only on the flow variable of its immediate dominator, except the root node which depends on none.

LEMMA 5.1.5. *When* **DomTDPropagate()** *is invoked at step* **8***, the flow equation at each node depends only on the flow variable of its immediate dominator.*

PROOF. From Lemma 5.1.4 we know that when the call to **ReduceLevel**(i) at step **3** terminates, all J edges whose source nodes are at level i are eliminated. Also, the **foreach** loop at step **2** calls **ReduceLevel**(i) in the decreasing order of i. Therefore when the loop terminates, all the J edges have been eliminated.

Also, the procedure **DomTDPropagate()** is invoked at step **8** only when the call to **ReduceLevel**(1) is terminated. □

Finally, we show the main theorem for correctness.

THEOREM 5.1.6. *The algorithm for eager elimination correctly computes the solutions to a set of data flow equations for a reducible flowgraph.*

PROOF. From Lemma 5.1.4 we know that the $\mathcal{E}$-*rules*, when applied in a bottom-up fashion, reduces a DJ graph to its dominator tree. At this point, from Lemma 5.1.5 we know that the equation at each node contains, on its RHS, only the flow variable of the node's immediate dominator, while the equation at the START node depends on no one, meaning that the RHS of the equation is constant. Therefore, we can propagate the solution of the START node down the dominator tree to compute the solutions at all other nodes. Recall that the flow equation at each node depends on its immediate dominator. Therefore our top-down propagation yields a correct solution at every node. □

5.2 Complexity

It is easy to see that each time E1 and E2a rules are applied we eliminate one J edge, and when E2b rule is applied we merely transform an edge $y \rightarrow z$ to $idom(y) \rightarrow z$. From this we can see that E1 and E2a are applied at most $O(|E|)$ times, whereas

E2b rule can be applied as many as $O(|E| \times |N|)$ times. We state this in the following lemma.

LEMMA 5.2.1. *The E1 and E2a rules will be applied at most $O(|E|)$ times, and E2b rule will be applied at most $O(|E| \times |N|)$ times.*

PROOF. Each application of E1 and E2a rules removes one edge, and since none of the *$\mathcal{E}$-rules* increase the number of edges in the DJ graph, we can apply these two rules at most $O(|E|)$.

Each application of E2b rule to an edge $y \rightarrow z$ removes this edge and introduces $idom(y) \rightarrow z$. Since there are at most $O(|N|)$ levels, the derived edge of $y \rightarrow z$ moves up at most $O(|N|)$ times. Finally, since there are only $O(|E|)$ edges, we can apply E2b rule at most $O(|E| \times |N|)$ times. □

Next it is also easy to see that for each application of E1 and E2 rules, we eliminate one flow variable. Since E2b rule is applied $O(|E| \times |N|)$, time complexity of eager elimination is $O(|E| \times |N|)$ function operation.

THEOREM 5.2.2. *The number of steps required to transform a DJ graph $\mathcal{G}^0$ to its dominator tree $\mathcal{G}^M$ using the $\mathcal{E}$-rules is bounded by $O(|E| \times |N|)$ steps.*

PROOF. Follows from Lemmas 5.1.4 and 5.2.1. □

It takes $O(|N|)$ steps to propagate data flow information down the dominator tree. So the time complexity of the entire eager elimination method is bounded by $O(|E| \times |N|)$ steps.

Finally, it is important to note that we have assumed constant time for propagating data flow information at each node, but this is not necessarily true in practice. To model reaching definitions requires propagating information for $|V|$ program variables, and the number of program variables could potentially be $O(|N|)$. So we should approriately accommodate the time complexity of propagating data flow information for a particular data flow problem in determining the actual time complexity of solving the data flow problem.

6. DELAYED ELIMINATION METHOD

To improve the worst-case quadratic behavior of eager elimination, we propose the delayed elimination method, whose worst-case time complexity is $O(|E| \times \log(|N|))$. The key intuition behind our delayed elimination method is based on the following observation. Consider the J edge $7 \rightarrow 9$ in Figure 5(a). The effect of our eager elimination triggered by this edge is as follows:

(1) E2b derives $6 \rightarrow 9$ from $7 \rightarrow 9$ (Figure 5(b)).
(2) E2b derives $3 \rightarrow 9$ from $6 \rightarrow 9$ (Figure 5(g)).
(3) E2a removes $3 \rightarrow 9$ (Figure 5(h)).

Rather than moving the source nodes of derived edges one level up at a time, we can directly generate just one derived edge such that its source and destination nodes are at the same level. We can do so if we know exactly which node is that source node. For each J edge $y \rightarrow z$, its **top node** is $x = \mathbf{Top}_{y \rightarrow z}$ such that x dominates y and such that $x.level = z.level$. For example, node 3 is the top node

for the J edge $7 \rightarrow 9$. In a preprocessing step we can easily compute the top nodes of *all* J edges in linear time [Sreedhar 1995]. An interesting point to note is that the destination node 9 of edges $7 \rightarrow 9$, $6 \rightarrow 9$, and $3 \rightarrow 9$ is in the dominance frontiers of the source nodes 7, 6, and 3. Thus, we use the term **dominance frontier interval path**, denoted as $x \stackrel{d+}{\rightarrow} y$, of the J edge $y \rightarrow z$ to mean the dominator tree path from top node x to node y.

Now we can define the D2b rule as follows:

Definition 6.1 (*D2b Rule*). Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a nonjoin node, let $y \rightarrow z$ be an outgoing edge, and let $x = \mathbf{Top}_{y \rightarrow z}$. If $idom(y) \neq idom(z)$ then

$$D2b(< \mathcal{G}^i, N, E, y \stackrel{J}{\rightarrow} z, x >) \implies < \mathcal{G}^{i+1}, N, (E - \{y \stackrel{J}{\rightarrow} z\}) \cup x \stackrel{J}{\rightarrow} z > .$$

In D2b rule we do not perform variable elimination, but we intentionally delay it. For DJ graph reduction, we carry over the definition of the E1 and E2a rules to our delayed elimination method, and call them the D1 and D2a rules, respectively. The associated variable elimination performed during D1 and D2a is different from that of their counterpart rules E1 and E2a; in D1 and D2a we also eliminate variables that were delayed by D2b. We will call the D1, D2a, and D2b rules collectively the $\mathcal{D}$-*rules*. The algorithm outline given in Figure 2 for eager elimination can be used for delayed elimination, by replacing the calls to **Eager1**(), **Eager2a**(), and **Eager2b**() in procedure **ReduceLevel**(i) by the calls to **Delayed1**(), **Delayed2a**(), and **Delayed2b**() (which implement the $\mathcal{D}$-*rules*), respectively. The procedures for implementing the $\mathcal{D}$-*rules* are given in Figure 6.

To eliminate variables delayed by D2b, we perform path compression akin to Tarjan's simple path compression during D1 and D2a rules. The path compression on a dominator tree is performed whenever we invoke the procedure **CompPath**(x, y), where x and y are nodes on the (compressed) dominator tree such that x is an ancestor of y. **CompPath**(x, y) performs the following operations on a dominator tree:

Delayed Substitution. For each node w on the (compressed) dominator path $x \stackrel{d+}{\rightarrow} y$, excluding x, express variable O_w as a linear function of O_x, by top-down traversal of the path.

Path Compression. Make all the nodes on the path $x \stackrel{d+}{\rightarrow} y$, the children of x.

These two operations are performed in the path order: for any two nodes u and v in the path, if u is an ancestor of v in the tree then we process u earlier than v. It is important to note that when a node w on the path $x \stackrel{d+}{\rightarrow} y$ is made a child of x, all children of w except the one on the path are still w's children. In other words, only one of w's children (i.e., the one on the path $x \stackrel{d+}{\rightarrow} y$) will change its parent from w to x.

The key point in delayed elimination is to eliminate variables only when applying D1 or D2a rule, but not when applying D2b rule. When **Delayed2b**($y \rightarrow z$) is invoked we keep track of **Bottom** node y in $x = \mathbf{Top}_{y \rightarrow z}$ in a list $x{->}jedges$ (step **51**). We also delete the edge $y \rightarrow z$ and insert $x \rightarrow z$ (if it does not already exist). When **Delayed1**($x \rightarrow x$) or **Delayed2a**($x \rightarrow z$) is later invoked, we first perform the path compression.

```
Procedure Delayed1(y → y)
{
35:   while(((u → w = GetJedge(y)) ≠ NULL)) do
      /* Replace node equations on y -d+-> u as a function of O_y  */
36:      CompPath(y,u) ;
37:      Eliminate O_u in H_w by replacing it with RHS of H_y.
38:   endwhile
39:   Compute f*(O_y).
40:   Delete the edge y → y ;
}
Procedure Delayed2a(y → z)
{
41:   while(((u → w = GetJedge(y)) ≠ NULL)) do
42:      CompPath(y,u) ;
43:      Eliminate O_u in H_w by replacing it with RHS of H_y.
44:   endwhile
45:   Eliminate O_y in H_z by replacing it with RHS of H_x,
        where x = parent(y) on the compressed dominator tree.
46:   Delete the edge y → z ;
47:   if(z becomes a nonjoin node) then
48:      Put z at the head of PriorityList[z.level] list ;
49:   endif
}
Procedure Delayed2b(y → z)
{
50:   x = Top_{y→z} ;
51:   Insert edge y → z in a list x->jedges.
52:   Delete the edge y → z ;
53:   if(x → z exists) then
54:      z.indegree = z.indegree − 1 ;
55:   else
56:      Insert a new J edge x → z ;
57:   endif
} foo bar
```

Fig. 6. Procedures for $\mathcal{D}$-*rules*.

In the procedure **Delayed1**$(y \rightarrow y)$, the function **GetJedge**(y) returns J edge $u \rightarrow w$ for which y is the **Top** node. For each such edge we invoke **CompPath**(y, u) to perform path compression and delayed variable substitution (step **36**). Finally, we compute the closure of the recursive equation H_y to break the loop (step **39**). At step **40** we eliminate the edge (as in **Eager2a**()). It is important to remember that the destination node w of the J edge $u \rightarrow w$, returned by **GetJedge**(y), will not necessarily be the same as y. This is because the y can be a **Top** node for many different J edges that were inserted in y->$jedges$ by some previous invocations of **Delayed2b**(). We will eliminate all such J edges from y->$jedges$ the first time the procedure **Delayed1**$(y \rightarrow y)$ is invoked on y.

The steps in **Delayed2a**() are similar to **Delayed1**() except that we do not compute closure (since there are no self-loops). Note that w can never be same as y; if it were then we should have invoked **Delayed1**() prior to invoking **Delayed2a**(). The operations from step **46** to step **49** are the same as in **Eager2a**() and essentially perform a DJ graph reduction.

In the procedure **Delayed2b**($y \to z$) we do not eliminate variables. At step **51** we save the J edge $y \to z$ such that $x = \mathbf{Top}_{y \to z}$, in the list $x{->}jedges$ (step **51**). These J edges are returned by the function **GetJedge**()) during D1 and D2a rules. step **41**, respectively. The operations from step **52** to step **57** are the same as in **Eager2b**() and essentially do DJ graph reduction.

The top-down propagation phase in the delayed elimination method is similar to the propagation phase in the eager elimination method, except that we propagate the data flow information on the compressed dominator tree.

Example. Here we illustrate the delayed elimination method for our example DJ graph. The complete DJ graph reduction process is shown in Figure 7.

(1) For the DJ graph in Figure 7(a) we apply D2b rule, invoking **Delayed2b**($7 \to 9$). Since variables are not eliminated during D2b reduction, no flow equation changes in response to the graph reduction from Figure 7(a) to Figure 7(c). But we insert the J edge $7 \to 9$ in the list $3{->}jedges$ (since node 3 is the **Top** of the J edge $7 \to 9$). For the DJ graph in Figure 7(b) we again apply D2b rule, invoking **Delayed2b**($4 \to 9$). Once again we insert the J edge $4 \to 9$ in the list $3{->}jedges$).

(2) For the DJ graph in Figure 7(c) we apply D2a rule, invoking **Delayed2a**($4 \to 6$). At node 4, we have $4{->}jedges = \emptyset$; therefore the procedure **GetJedge**(4) would return $NULL$ (step **41**). At step **45** we eliminate O_4 on the RHS of H_6 by replacing it with a linear function of O_3. Once this is done, the flow equation at node 6 becomes

$$O_6 = P_6(P_4O_3 + G_4 + O_5) + G_6.$$

(3) For the DJ graph in Figure 7(d) we apply D2a rule, invoking **Delayed2a**($5 \to 6$). At node 5, we again have $5{->}jedges = \emptyset$; therefore **GetJedge**(5) would return $NULL$ (step **41**). At step **45** we eliminate O_5 on the RHS of H_6 by replacing it with a linear function of O_3. Once this is done, the flow equation at node 6 becomes

$$\begin{aligned} O_6 &= P_6(P_4O_3 + G_4 + P_5O_3 + G_5) + G_6 \\ &= (P_6P_4 + P_6P_5)O_3 + P_6G_4 + P_6G_5 + G_6. \end{aligned}$$

Notice that at this point O_6 depends only on O_3 (where node 3 is a parent of 6 on the dominator tree).

(4) Next we apply D2b rule to the edge $6 \to 2$, transforming the DJ graph of Figure 7(e) to Figure 7(f). We invoke **Delayed2b**($6 \to 2$), in which we store $6 \to 2$ in $2{->}jedges$ (step **51**).

(5) Next we apply D2a rule to the edge $3 \to 9$, transforming the DJ graph of Figure 7(f) to Figure 7(g). We invoke **Delayed2a**($3 \to 9$). At this point $3{->}jedges$ contains edges $4 \to 9$ and $7 \to 9$. So at step **41** the procedure **GetJedge**(y) returns these two edges, one after another. Let $4 \to 9$ be the first edge returned, and so at step **42** we invoke **CompPath**(3, 4). In the procedure **CompPath**() we express the flow equation at every node on the path $3 \xrightarrow{d+} 4$, excluding 3, as a function of O_3. Since 4 is the only node on

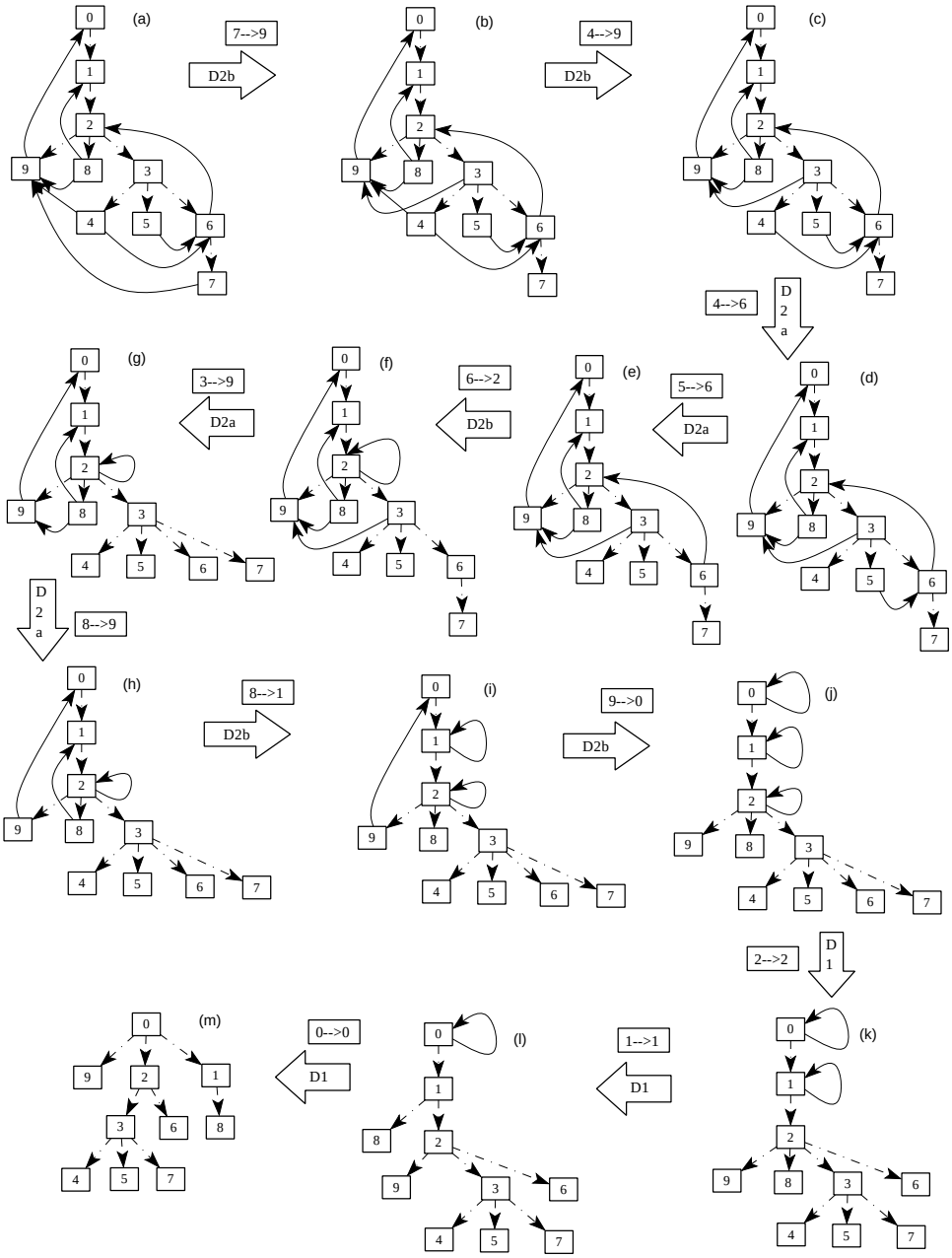

Fig. 7. A trace of the DJ graph reduction using $\mathcal{D}$-*rules*.

this path, and its equation is already in the required form, we return from the procedure **CompPath**$(3, 4)$. At step **45** we replace O_4 on the RHS of H_9 as a linear function of O_3. The resulting equation of node 9 is

$$O_9 = P_9(P_4O_3 + G_4 + O_7 + O_8) + G_9.$$

Next **GetJedge**(y) would return the J edge $7 \rightarrow 9$ from the list $3\text{->}jedges$. Therefore we invoke **CompPath**$(3, 7)$. Within the procedure **CompPath**() we first express the flow equation of every node on the path $3 \xrightarrow{d+} 7$, excluding 3, as a function O_3 in a top-down fashion. The path $3 \xrightarrow{d+} 7$ contains nodes 6 and 7 (excluding 3). The equation at node 6 is already expressed as a function of O_3. But the equation at node 7 is expressed in terms of O_6. Therefore we replace O_6 in H_7 by a linear function of O_3. By doing so the new equation at node 7 becomes

$$\begin{aligned} O_7 &= P_7O_6 + G_7 \\ &= P_7((P_6P_4 + P_6P_5)O_3 + P_6G_4 + P_6G_5 + G_6) + G_7 \\ &= (P_7P_6P_4 + P_7P_6P_5)O_3 + P_7P_6G_4 + P_7P_6G_5 + P_7G_6 + G_7. \end{aligned}$$

During path compression we also make node 7 a child of node 3, and the resulting compressed dominator tree is shown in Figure 7(g). Next, at step **43** we eliminate O_7 on the RHS of H_9 by replacing it with a linear function O_3. By doing so the new equation at node 9 becomes

$$\begin{aligned} O_9 &= P_9(P_4O_3 + G_4 + O_7 + O_8) + G_9 \\ &= P_9P_4O_3 + P_9O_7 + P_9O_8 + P_9G_4 + G_9 \\ &= P_9P_4O_3 + P_9((P_7P_6P_4 + P_7P_6P_5)O_3 + P_7P_6G_4 + \\ &\quad P_7P_6G_5 + P_7G_6 + G_7) + P_9O_8 + P_9G_4 + G_9 \\ &= P_9P_4O_3 + (P_9P_7P_6P_4 + P_9P_7P_6P_5)O_3 + P_9P_7P_6G_4 + \\ &\quad P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + P_9O_8 + P_9G_4 + G_9 \\ &= (P_9P_4 + P_9P_7P_6P_4 + P_9P_7P_6P_5)O_3 + P_9O_8 + P_9P_7P_6G_4 + \\ &\quad P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + P_9G_4 + G_9. \end{aligned}$$

Finally at step **43** we eliminate O_3 on the RHS of H_9 by replacing it with a linear function of O_2. By doing so the equation at node 9 becomes

$$\begin{aligned} &= (P_9P_4 + P_9P_7P_6P_4 + P_9P_7P_6P_5)(P_3O_2 + G_3) + P_9O_8 + P_9P_7P_6G_4 \\ &\quad + P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + P_9G_4 + G_9. \end{aligned}$$

(6) We can continue to eliminate variables as described above at other nodes. We encourage interested readers to do so.

7. COMPLEXITY OF DELAYED ELIMINATION

In this section we analyze the complexity of the delayed elimination method. The correctness of the delayed elimination method can be established along the same line as in the eager elimination method.

The complexity of the delayed elimination method can be established using the *Rising Roots Condition* property of path compression [Lucas 1990]. Let the top

node in a path compression be called the *root* of the compression. Let $(C_1, \ldots, C_e)$ be a sequence of path compressions that transforms an initial (dominator) tree T_0 to the final compressed tree T_e (with C_i transforming T_{i-1} into T_i). The rising roots condition for a sequence of path compressions, as defined by Lucas [1990], is given below:

Definition 7.1. A sequence of path compressions $(C_1, \ldots, C_m)$ satisfies the *Rising Roots Condition (RRC)* if and only if for every node x, if x appears as a nonroot node in any compression C_i, then for every $j > i$, x appears as a nonroot node in C_j if C_j is a compression from y and y is a descendant of x in T_{j-1}.

Lucas then showed that a sequence of path compressions on a tree that satisfies the RRC corresponds to some sequence of intermixed *union* and *find* operations used in the disjoint set union problem; and conversely, a sequence of intermixed *union* and *find* corresponds to some sequence of path compressions satisfying the RRC. Tarjan and van Leeuwen [1984] showed that the time complexity for a sequence of e intermixed *union* and *find* operations is $O(e \times \log n)$, where n is the number of elements in the set. Consequently, if we can show that the sequence of e path compression performed during our delayed elimination does satisfy the RRC, it immediately follows that the total cost of our e path compression is also $O(e \times \log |N|)$, where $|N|$ is the number of nodes in the dominator tree.

LEMMA 7.2. *The sequence of path compressions performed during our bottom-up reduction satisfies the RRC.*

PROOF. Our path compression are ordered by the levels of their roots (i.e., the **Top** nodes). Therefore, once a node has been a nonroot node in a compression, it will never be a root node in any future compression. □

THEOREM 7.3. *The time complexity of the delayed elimination method is* $O(|E| \times \log |N|)$.

PROOF. Follows from Lemma 7.2 and the fact that the number e of path compression is bounded by $|E|$. □

More recently, Buchsbaum et al. established the lower bound of $\Theta(e \times \log n)$ for a sequence of e (order-preserving) path compressions satisfying the RRC on an initial tree of n nodes [Buchsbaum et al. 1995]. For most flowgraphs, $|E| = O(|N|)$, so the above bound is tight.

8. HANDLING IRREDUCIBILITY

In this section, we describe our strategy for handling irreducibility in flowgraphs using eager elimination.[7] Figure 8 shows an irreducible flowgraph and its DJ graph.

In our elimination methods we detect irreducibility whenever during the bottom-up reduction **GetNode**(i) returns $NULL$ and there are still J edges to be processed at this level. At step **4**, if $PriorityList[i]$ is not empty, then there are still nodes to be processed at this level, but none are non-join nodes. This condition is sufficient to signal irreducibility, and we invoke the procedure **CollapseIrreducible**(i) to

[7]We can similarly handle irreducibility in delayed elimination.

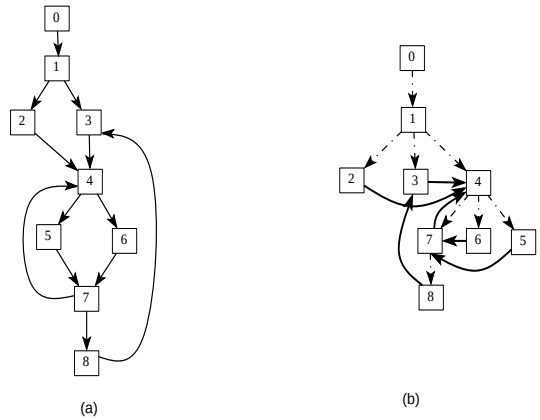

Fig. 8. An irreducible flowgraph and its DJ graph.

Procedure CollapseIrreducible(i)
{
`/* See also the description in the main text */`
58: Determine the SCCs for the nodes at level i, and construct dag(s) of SCCs;
59: Process each SCC S in each dag in topological order as follows:
60: If S contains a cycle then compute the closure for the equations of all the nodes in the cycle. Also express the RHS of the equation at each node in terms of its immediate dominator's flow variable.
61: If a node $s \in S$ has an edge $s \rightarrow t$ to node $t \notin S$ then replace O_s on the RHS of equation H_t by the RHS of H_s, which is expressed as a function of $O_{idom(s)}$.
`/* After all the SCCs are processed, */`
`/* the equation of every node at level` i `*/`
`/* is expressed as a function of its */`
`/* immediate dominator's flow variable. */`
62: Remove all J edges whose source and destination are at level i.
63: Apply the E2b rule to every J edge whose source is at level i but whose destination is at level less than i.
}

Fig. 9. Algorithm for handling irreducibility with eager elimination.

handle irreducibility at level i. The description of **CollapseIrreducible**(i) under eager elimination is given in Figure 9.

We first apply Tarjan's Strongly Connected Component (SCC) algorithm on nodes at level i (step **58**). It is important to ensure that during Tarjan's algorithm, we never visit any node whose level is not i. This generates dag(s) of SCCs. Notice

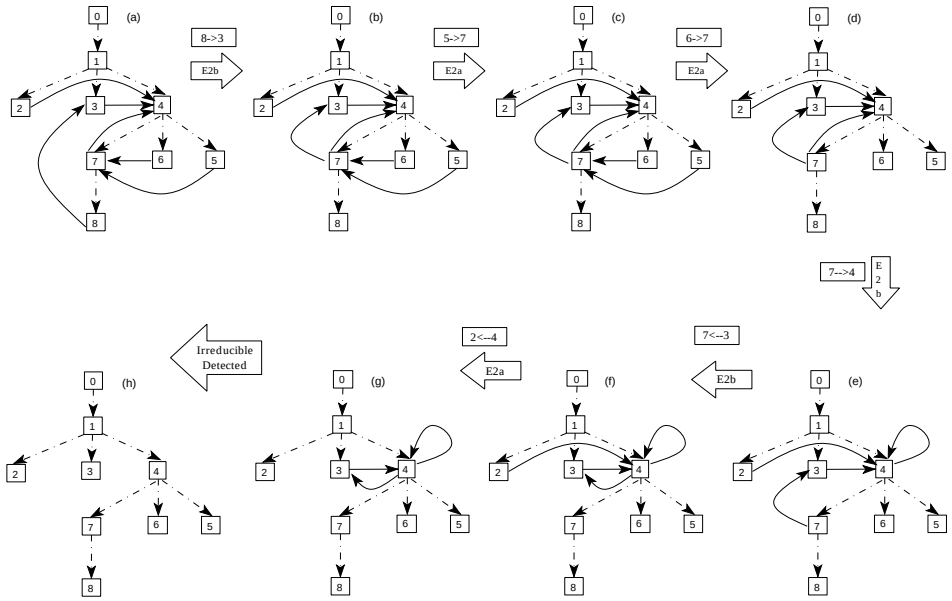

Fig. 10. A trace of DJ graph reduction for the example irreducible flowgraph.

that there can be more than one disjoint dag at a level. For every dag, we process each SCC S in the dag in topological order (step **59**). We compute the closure of the equations for all nodes in S if S is nontrivial (step **60**). We also express the equation at every node at level i in terms of its immediate dominator's flow variable (step **61**). This is possible because we are eliminating the flow variables in SCCs' topological order. Finally, we eliminate all the J edges lying at level i (step **62**). Figure 10 gives a trace of the DJ graph reduction using the $\mathcal{E}$-*rules* for the DJ graph shown in Figure 8.

For irreducible flowgraphs, the cost of our elimination methods depends on Tarjan's SCC algorithm and on finding the closure of the set of mutually recursive equations of the nodes involved. The time complexity of finding all SCCs in a graph with e edges is $O(e)$. Recall that before we apply Tarjan's SCC algorithm at a level, we have processed J edges outgoing from nodes at the level with the $\mathcal{E}$-*rules* or $\mathcal{D}$-*rules*. Consequently, any remaining J edge has both the source and the destination node at the level. It is sufficient for Tarjan's SCC algorithm to process these remaining edges. Since each J edge can be processed by the SCC algorithm at most once through itself or its last derived edge, the total cost of Tarjan's SCC algorithm performed to handle irreducibility is $O(|E|)$ for both eager and delayed elimination.

Another additional cost of handling irreducibility is performing fixed-point iteration over all the nodes/edges involved (step **60**). In the worst case, there are $O(|N|)$ such nodes or $O(|E|)$ such edges, so our methods will be only as efficient as iterative methods. However, we do not expect this worst-case scenario to happen

in practice.

9. EMPIRICAL OBSERVATION OF EXHAUSTIVE METHODS

Although both eager and delayed elimination are theoretically worse than linear, they can behave as if they were *linear in practice*. To establish this statement, we first state two claims to relate our elimination methods to dominance frontiers.[8]

CLAIM 9.1. *The number of the $\mathcal{E}$-rules applied during the entire elimination phase is bounded by the total size of dominance frontiers of the flowgraph.*

CLAIM 9.2. *In delayed elimination, the total length of the paths compressed during the D1 and D2a rules is bounded by the total size of the dominance frontier relation.*

Cytron et al. [1991] have presented that, in practice, the size of the dominance frontier relation is linear in the size of its flowgraph. Therefore, the time complexity of our elimination methods is expected to be linear in practice. We will also show that this is indeed true using empirical results.

To demonstrate the effectiveness of our approach, we implemented our methods for solving intraprocedural reaching definitions problem. We used as test data a set of 40 Fortran procedures taken from SPEC92, RiCEPS, LAPACK, and GATOR. We performed our experiments on a SPARC-20 workstation. The results of experiments are summarized below:

—Of the 40 procedures we tested, five procedures have irreducible loops. Three of these five procedures have only one irreducible loop.

—The maximum size of SCC, found when Tarjan's SCC algorithm is applied during **CollapseIrreducible**(), is 4 (found in procedure `coef`). This suggests that our approach is indeed very efficient in practice for handling irreducible loops when compared to other related approaches (see details later in the section).

—As pointed out by Cytron et al. [1991] the size of the dominance frontier relation is linear in practice. We found the average ratio $|DF|/|E_f| = 1.1$. This ratio suggests that the average size of dominance frontiers (represented as a set of edges) is about $1.1 - 1/1.1 = 9.1\%$ more than the size of the flowgraph. This confirms to the claim made by Cytron et al. that the size of dominance frontiers is indeed linear in practice.

—As expected, the number of $\mathcal{E}$-*rules* applied is bounded by the size of the dominance frontier relation. We found the average ratio $nE/|DF|$ to be 0.76, where nE is the number of $\mathcal{E}$-*rules* applied, and $|DF|$ is the size of dominance frontiers. This suggests that eager elimination is expected to behave linearly in practice.

—As expected, the number of $\mathcal{D}$-*rules* applied nD is less than the size of the flowgraph $|E_f|$, with the average ratio of $nD/|E_f| = 0.52$.

—We found that the average ratio of the total length of the dominance frontier interval paths without path compression C' to the total length of the dominance frontier interval paths with path compression C to be 1.3 (i.e, $C'/C = 1.3$).

[8]One can easily prove the two lemmas using Lemma 15.1; we will leave it as an exercise for readers to prove the results.

Table III. Notation Used in Tables IV and V

Name	Procedure names
$\|N\|$	Number of flowgraph nodes
$\|E_f\|$	Number of flowgraph edges
I	Number of times the procedure **CollapseIrreducible**() is called
S	Maximum size of SCCs detected during **CollapseIrreducible**()
$R1$	Number of times E1 (or D1) rule applied
$R2a$	Number of times E2a (or D2a) rule applied
$E2b$	Number of times E2b rule applied
nE	Total number of $\mathcal{E}$-*rules* applied
$D2b$	Number of times D2b rule applied
nD	Total number of $\mathcal{D}$-*rules* applied
C'	Total length of the dominance frontier interval paths without path compression
C	Total length of the dominance frontier interval paths with path compression
C'/C	Ratio of C' to C
$\|DF\|$	Size of dominance frontiers in the flowgraph
$\|G\|$	Total number of downward exposed definitions for scalar variables
T_e	Execution time in seconds for the eager elimination method
T_d	Execution time in seconds for the delayed elimination method
$S_{e/d}$	T_e/T_d

This ratio indicates that all the *dominance frontier interval paths* have about $1.3 - 1.0/1.3 = 23.1\%$ of their edges overlapped.

—The value of $C'/C = 1.3$ also suggests that, in an ideal situation, the delayed elimination method can be about 1.3 times as fast as eager elimination. In practice, however, delayed method incurs overhead from bookkeeping during path compression.

—We found that eager elimination is almost as fast as delayed elimination. On average delayed elimination is only about 1.15 times faster than eager elimination.

Tables IV and V give a summary of our empirical results. The notation used in these tables are given in Table III.

Table IV gives the empirical results for our test procedures. The second and third columns give the number of nodes (basic blocks) and the number of edges in the flowgraph for each procedure, respectively.

The column I shows the number of times the procedure **CollapseIrreducible**() is invoked. Recall that this procedure is invoked only if irreducibility is detected at a particular level. Column S shows the maximum size of nontrivial (size > 1) strongly connected components detected and processed by this procedure. This column quantifies the number of nodes whose data flow equations are involved in fixed-point iteration. Our results indicate that the maximum size of SCCs is 4 for one procedure (`coef`), 3 for two procedures (`card` and `comlr`), and 2 for two procedures (`dcdcmp` and `readin`). Previous approaches perform iteration over a normally much larger region when an irreducible region is encountered [Burke 1990]. For example, Burke's approach to handling irreducibility identifies the smallest single-entry region that completely encloses the irreducible portion. We measured the size of such single-entry regions and found it to be around 25 for the procedure `coef`, 33 for `comlr`, 78 for `card`, and 28 for `dcdcmp`.[9] From this it is evident that our

[9]These numbers were generated using *Sparse* compiler developed at Oregon Graduate Institute of Science and Technology.

Table IV. Empirical Results

Name	$\|N\|$	$\|E_f\|$	I	S	$R1$	$R2a$	$E2b$	nE	$D2b$	nD	C'	C	C'/C	$\|DF\|$
aerset	329	460	0	0	44	79	200	323	81	204	232	213	1.1	371
aqset	189	258	0	0	24	42	118	184	48	114	127	116	1.1	202
bjt	135	187	0	0	1	62	114	177	50	113	132	95	1.4	211
card	150	216	2	3	11	43	170	224	58	112	230	165	1.4	316
chemset	229	320	0	0	23	68	191	282	57	148	198	166	1.2	300
chgeqz	174	248	0	0	13	58	142	213	61	132	203	141	1.4	298
clatrs	214	308	0	0	17	79	129	225	76	172	152	142	1.1	276
coef	95	137	1	4	11	32	69	112	32	75	82	59	1.4	142
comlr	69	91	1	3	11	13	47	71	20	44	49	49	1.0	78
dbdsqr	228	327	0	0	19	74	176	269	75	168	275	188	1.4	391
dcdcmp	137	187	1	2	14	35	127	176	50	99	176	127	1.4	245
dcop	186	261	0	0	10	87	143	240	69	166	182	154	1.2	295
dctran	326	458	0	0	10	119	278	407	93	222	577	273	2.1	745
deseco	175	236	0	0	18	54	107	179	47	119	135	114	1.2	220
dgegv	160	232	0	0	11	51	102	164	55	117	200	120	1.7	287
dgesvd	321	470	0	0	6	102	206	314	111	219	406	257	1.6	585
dhgeqz	285	408	0	0	32	92	238	362	94	218	335	221	1.5	484
disto	133	191	0	0	7	53	98	158	58	118	107	102	1.1	186
dlatbs	167	238	0	0	12	60	102	174	57	129	127	113	1.1	220
dtgevc	321	459	0	0	50	93	213	356	114	257	274	230	1.2	439
dtrevc	248	353	0	0	21	85	157	263	75	181	192	173	1.1	318
elprnt	162	227	0	0	18	59	99	176	44	121	99	96	1.0	183
equilset	327	451	0	0	55	74	205	334	93	222	212	202	1.1	353
errchk	346	482	0	0	40	99	267	406	102	241	358	278	1.3	528
iniset	333	486	0	0	154	0	154	308	154	308	154	154	1.0	308
init	122	175	0	0	17	35	66	118	31	83	70	68	1.0	125
initgas	189	263	0	0	40	37	126	203	52	129	126	126	1.0	205
jsparse	281	403	0	0	52	70	185	307	76	198	185	183	1.0	313
modchk	306	419	0	0	25	111	205	341	74	210	240	213	1.1	390
moseq2	161	217	0	0	2	67	137	206	50	119	181	122	1.5	267
mosfet	214	295	0	0	1	91	231	323	76	168	307	184	1.7	427
noise	115	159	0	0	7	57	72	136	43	107	77	76	1.0	147
out	403	589	0	0	92	92	306	490	136	320	330	313	1.1	525
reader	182	235	0	0	8	53	28	89	15	76	28	28	1.0	89
readin	406	611	2	2	29	84	685	798	174	285	1457	574	2.5	1675
setupgeo	188	275	0	0	44	37	112	193	61	142	127	118	1.1	218
setuprad	195	286	0	0	55	38	124	217	68	161	124	124	1.0	220
smvgear	212	310	0	0	66	23	248	337	91	180	363	203	1.8	468
solveq	196	289	0	0	72	28	160	260	93	193	162	162	1.0	265
twldrv	168	243	0	0	16	59	127	202	49	124	189	133	1.4	280
Average	219	312	0.2	0.3	29	62	167	258	72	163	229	164	1.3	341

approach is very efficient in practice for handling irreducible flowgraphs.[10]

The columns $R1$ and $R2a$ indicate the number of times E1 (D1) and E2a (D2a) rules, respectively, were invoked during the eager (the delayed) elimination method. The number of E1 (D1) rules give the number of single-entry loops in a procedure. The column $E2b$ shows the number of E2b rules applied in eager elimination, whereas $D2b$ shows the number of D2b rules applied in delayed elimination. The

[10]For backward flow problems, such as live uses of variables, we expect to see must more irreducible regions, since the analysis is performed on the reverse flowgraph, and there are normally multiple loop exits in the forward flowgraph.

column nE represents the sum of the three columns $R1$, $R2a$, and $E2b$, and nD represents the sum of $R1$, $R2a$, and $D2b$.

The column C' gives the total length of all the *dominance frontier interval paths* without path compression performed. In comparison, the column C gives a similar number, but with path compression performed. Recall that the length of a dominance frontier interval path can progressively decrease after its overlapped subpaths are compressed. As we can see from the table there is not much difference between C' and C. The ratio C'/C gives an indication about how much improvement delayed elimination can achieve over eager elimination. The average ratio is 1.3, indicating that delayed elimination method should, ideally, be faster than eager elimination by a factor of 1.3 (if we ignore the overhead of bookkeeping in delayed elimination). This is evidenced by the data reported in Table V, where we can see that, on average, delayed elimination is about 1.15 times as fast as eager elimination.

As we can see from Table IV, the lengths C and C' are less than $|DF|$. From this result we can conclude that the time complexity of delayed elimination is expected to be linear in practice. Also, from the table we can see that the number of $\mathcal{E}$-*rules* applied (i.e., nE) is less than $|DF|$. Thus we can conclude that the time complexity of eager elimination should also be linear in practice (since the size of dominance frontiers is linear in practice). Finally, as expected, we can see that nD, the total number of $\mathcal{D}$-*rules* applied, is bounded by the number of flowgraph edges $|E_f|$.

In delayed elimination, although each D2b rule takes only constant time, the cost of applying D1 or D2a rule includes the path compression overhead. The total number of path compressions performed is equal to the number of D2b rules applied, but each compression can take $O(\log |N|)$ time (in the worst case). Therefore delayed elimination can suffer from the $\log |N|$ overhead factor due to a path compression. For the benefit of path compression to be fully exploited, there must be enough overlapping paths, so that future path compression could take less time. The number of dominance frontier interval paths passing through a node is bounded by the number of nodes in its dominance frontier. In other words, a node can participate in path compression only as many times as the size of its dominance frontier set, which is a small constant in practice (we can use use $|DF|$ and $|E_f|$ from Table IV to calculate $|DF|/|E_f|$ as an estimate). Therefore, we believe the benefit of path compression will normally not be fully utilized for real programs. In other words, to effectively exploit the benefits of path compression requires that the total length of the paths that are compressed be greater than $O(|E_f| \times \log |N|)$. An example flowgraph where path compression would beneficial is the ladder graph where the total length of paths compressed could be $O(|N|^2)$ [Cytron and Ferrante 1995; Sreedhar and Gao 1996].

Table V gives the timing data from our experiments. The columns T_e and T_d give the execution times in seconds for the eager and the delayed elimination method, respectively. The column $S_{e/d}$ gives the speedups of delayed elimination over eager elimination. We also observed that the execution time of both eager and delayed elimination is linearly proportional to the product $|E_f| \times |G|$, where $|G|$ is the number of downward exposed definitions for scalar variables (see Figure 11).

From the execution characteristics we can see that the eager elimination method is competitive with the delayed elimination method. Eager elimination even out-

Table V. Timings and Speedups

Name	$\|E_f\|$	$\|G\|$	T_e	T_d	$S_{e/d}$
aerset	460	166	0.41	0.38	1.08
aqset	258	74	0.11	0.10	1.10
bjt	187	213	0.38	0.31	1.23
card	216	63	0.26	0.20	1.30
chemset	320	94	0.24	0.21	1.14
chgeqz	248	115	0.23	0.18	1.28
clatrs	308	160	0.36	0.33	1.09
coef	137	117	0.25	0.22	1.14
comlr	91	63	0.12	0.11	1.09
dbdsqr	327	144	0.40	0.31	1.29
dcdcmp	187	75	0.20	0.15	1.33
dcop	261	127	0.25	0.22	1.14
dctran	458	151	0.54	0.26	2.08
deseco	236	240	0.39	0.42	0.93
dgegv	232	108	0.22	0.16	1.38
dgesvd	470	405	1.20	1.06	1.13
dhgeqz	408	261	0.79	0.53	1.49
disto	191	183	0.39	0.34	1.15
dlatbs	238	134	0.31	0.31	1.00
dtgevc	459	257	0.96	0.86	1.12
dtrevc	353	148	0.46	0.40	1.15
elprnt	227	138	0.27	0.24	1.13
equilset	451	179	0.58	0.55	1.05
errchk	482	154	0.59	0.52	1.13
iniset	486	280	0.87	0.97	0.90
init	175	129	0.17	0.20	0.85
initgas	263	114	0.27	0.29	0.93
jsparse	403	214	0.63	0.65	0.97
modchk	419	137	0.48	0.42	1.14
moseq2	217	248	0.45	0.30	1.50
mosfet	295	263	0.79	0.50	1.58
noise	159	135	0.19	0.22	0.86
out	589	404	1.49	1.43	1.04
reader	235	159	0.23	0.51	0.45
readin	611	240	1.46	1.00	1.46
setupgeo	275	195	0.37	0.41	0.90
setuprad	286	223	0.45	0.42	1.07
smvgear	310	224	0.70	0.48	1.46
solveq	289	186	0.49	0.53	0.92
twldrv	243	594	1.22	1.05	1.16
Average	312	188	0.50	0.44	1.15

performs delayed elimination in some cases. This is because there are not many overlapping paths in our test programs for delayed method to benefit from path compression. Theoretically, eager elimination is worse than its delayed counterpart in terms of time complexity. However, our empirical results demonstrate that eager elimination is very competitive when compared with delayed elimination. From a pragmatic point view, we thus recommend that one implement the eager method. Not only is it simple and easy to implement, but also it is amenable to incremental data flow analysis (see Section 10).

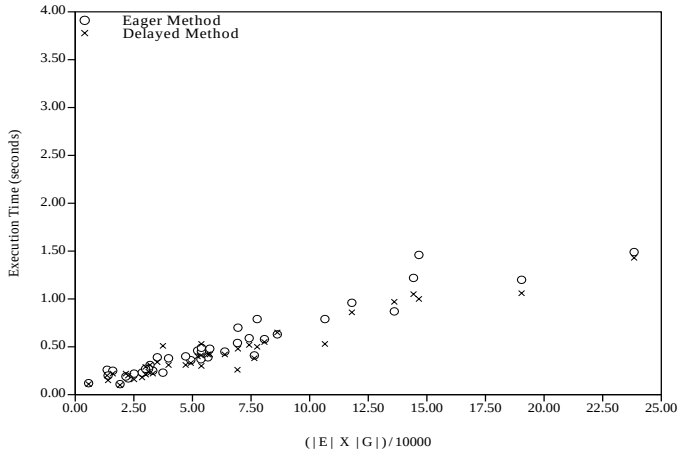

Fig. 11. Execution timing characteristics of eager and delayed elimination.

10. INCREMENTAL DATA FLOW ANALYSIS: PROBLEM FORMULATION

In the next several sections we present our incremental data flow analysis. This section lays the foundation of our incremental data flow analysis by introducing the concept of initial and final flow equations, and the representation called the DF graph, that are fundamental to our approach. Section 11 shows how to compute the set of nodes whose final flow equation change due to an incremental change. Section 12 shows how to update the final flow equations for nonstructural changes, and Section 13 shows how to handle structural changes. Here we also give an algorithm for updating the dominance frontier relation. Section 14 describes how to update the final data flow solutions for both structural and nonstructural changes.

10.1 Initial and Final Flow Equations

In our exhaustive eager elimination method, we showed how to express the RHS of a node's flow equation only in terms of its immediate dominator's flow variable by DJ graph reduction and variable elimination. We use the term **final flow equations** to name the set of equations at the end of the elimination phase. In contrast, we use the term **initial flow equations** for the set of flow equations prior to graph reduction and variable elimination.

Example. Consider a forward data flow problem with union as the merge operation (e.g., Reaching Definitions). The initial flow equations for the example flowgraph in Figure 8 are as follows (the *superscripts* 0 and F are used to indicate *initial* and *final*, respectively):

$$
\begin{aligned}
H_0^0: \quad & O_0 = G_0^0 \\
H_1^0: \quad & O_1 = P_1^0 O_0 + G_1^0 \\
H_2^0: \quad & O_2 = P_2^0 O_1 + G_2^0 \\
H_3^0: \quad & O_3 = P_3^0 (O_1 + O_8) + G_3^0
\end{aligned}
$$

$$
\begin{aligned}
H_4^0: \quad & O_4 = P_4^0(O_2 + O_3 + O_7) + G_4^0 \\
H_5^0: \quad & O_5 = P_5^0 O_4 + G_5^0 \\
H_6^0: \quad & O_6 = P_6^0 O_4 + G_6^0 \\
H_7^0: \quad & O_7 = P_7^0(O_5 + O_6) + G_7^0 \\
H_8^0: \quad & O_8 = P_8^0 O_7 + G_8^0
\end{aligned}
$$

The corresponding final flow equations are given below. Equations H_3^F and H_4^F are mutually recursive. Once their closure is determined, their final flow equations will be expressed only in terms of variable O_1 on the RHS.

$$
\begin{aligned}
H_0^F: O_0 &= G_0^0 \\
H_1^F: O_1 &= P_1^0 O_0 + G_1^0 \\
H_2^F: O_2 &= P_2^0 O_1 + G_2^0 \\
H_3^F: O_3 &= P_3^0 O_1 + (P_3^0 P_8^0 P_7^0 P_5^0 + P_3^0 P_8^0 P_7^0 P_6^0) O_4 \\
&\quad + P_3^0 P_8^0 P_7^0 G_6^0 + P_3^0 P_8^0 P_7^0 G_5^0 + P_3^0 P_8^0 G_7^0 \\
&\quad + P_3^0 G_8^0 + G_3^0 \\
H_4^F: O_4 &= P_4^0 P_2^0 O_1 + (P_4^0 P_3^0 P_8^0 P_7^0 P_5^0 + P_4^0 P_3^0 P_8^0 P_7^0 P_6^0) O_3 \\
&\quad + P_4^0 P_3^0 P_8^0 P_7^0 G_5^0 + P_4^0 P_3^0 P_8^0 P_7^0 G_6^0 + P_4^0 P_3^0 P_8^0 G_7^0 \\
&\quad + P_4^0 P_3^0 G_8^0 + P_4^0 G_3^0 + G_4^0 \\
H_5^F: O_5 &= P_5^0 O_4 + G_5^0 \\
H_6^F: O_6 &= P_6^0 O_4 + G_6^0 \\
H_7^F: O_7 &= (P_7^0 P_5^0 + P_7^0 P_6^0) O_4 + P_7^0 G_5^0 + P_7^0 G_6^0 + G_7^0 \\
H_8^F: O_8 &= P_8^0 O_7 + G_8^0
\end{aligned}
$$

10.2 Basic Steps

To simplify the presentation, we consider only two types of incremental changes. One can easily extend and implement other types of changes using our results.

(1) *Nonstructural change:* The parameters P_y^0 and G_y^0 of the initial flow equation H_y^0 at a node y are modified.

(2) *Structural change:* A flowgraph edge $x \rightarrow y$ is either inserted or deleted in the flowgraph.

It is important to remember that all incremental algorithms rely on having correct solutions at all nodes prior to incremental changes. Once a change occurs, incremental algorithms will update (ideally) only those solutions that are affected due to the change [Marlowe 1989; Marlowe and Ryder 1990a]. It is also important to remember that we maintain (and hence update) data flow solutions only for nodes in REACH(START). Let α_y^F denote the data flow solution at each node y prior to an incremental change. That is, α_y^F is the **final flow solution** at y before the change.

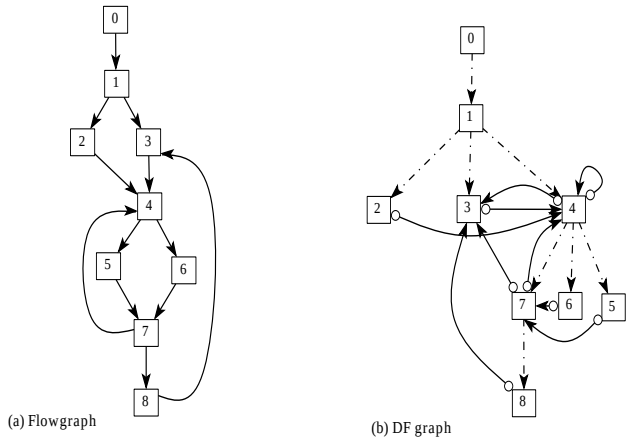

Fig. 12. A flowgraph and its DF graph.

Our approach annotates each node y with (1) the initial flow equation H_y^0, (2) the final flow equation H_y^F, and (3) the final flow solution α_y^F. Whenever an incremental change is introduced, our approach updates all three pieces of information. Updating the initial flow equations is straightforward. Our further discussion focuses on the strategy to update the final flow equations and the final flow solutions.

To effectively handle incremental updates, we use a representation called the **Dominance Frontier (DF) graph**. A DF graph is just the dominator tree of a flowgraph augmented with edges $x \rightarrow y$ such that $y \in DF(x)$.[11] We use DF graphs for updating the final flow equations. Figure 12 gives the DF graph for our example flowgraph.

11. FINDING THE SET OF AFFECTED NODES

Given an incremental change (either nonstructural or structural), the first step is to determine the set of nodes whose final flow equations are affected. To determine the set of affected nodes we made one key observation. Consider the final flow equation of node 7 of the example flowgraph shown in Figure 8:

$$\begin{aligned} H_7^F: \quad O_7 &= (P_7^0 P_5^0 + P_7^0 P_6^0) O_4 + P_7^0 G_5^0 + P_7^0 G_6^0 + G_7^0 \\ &= P_7^F O_4 + G_7^F \end{aligned}$$

We notice that the final parameters P_7^F and G_7^F on the RHS of the equation contain the initial parameters P_5^0, G_5^0, P_6^0, and G_6^0 from nodes 5 and 6. It is important to remember that the parameters of both the initial flow equation and the final flow equation at a node are constant values. We use the term *appear* to mean that the parameters of the final flow equations are computed from the parameters of the initial flow equations during the reduction and variable elimination phase of our eager elimination method. Given this notion, the key question is this: how are the

[11] In implementation, there is no need to construct a separate construction of the DF graph. To achieve what is needed in our approach, one can augment the DJ graph with additional edges.

parameters of the final flow equations related to those of the initial flow equations? We found a surprisingly simple relationship between them.

CLAIM 11.1. *Let P_w^F and G_w^F be the parameters of the final flow equations at node w. P_u^0 and G_u^0 will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$.*

$IDF(u)$ represents the iterated dominance frontier of u. The above claim has an important implication in our incremental algorithm. Suppose we make a nonstructural change at a node y, thereby affecting P_y^0 and G_y^0. From Claim 11.1 we may have to update the final flow equation at all nodes in $IDF(y)$. Since we are changing P_y^0 and G_y^0, the final flow equation H_y^F at node y should be updated. Let FEqAffected(y) be the set of all nodes whose final flow equations *potentially* change due to a nonstructural change at y; that is, FEqAffected(y) is the set of "possibly affected" nodes. By possibly affected we mean that if the final flow equation at a node w changes it will be in FEqAffected(y). To discover the set of possibily affected nodes, we use the following key result:

CLAIM 11.2. *Let a nonstructural change be introduced at node y. Then the final flow equation at a node w is* possibly affected (*i.e., w is in* FEqAffected(y)) *if and only if $w \in \{y\} \cup IDF(y)$.*

The situation is slightly different for structural updates. Let the edge $x \rightarrow y$ be the structural update (either inserted or deleted). To discover the set of possibily affected nodes, we use the following key result:

CLAIM 11.3. *Let $x \rightarrow y$ be the edge that is updated in the flowgraph. The final data flow equation at a node w is possibly affected if and only if $w \in \{y\} \cup IDF(y)$.*

The proofs of the above claims are given in Section 15. Notice that Claim 11.3 is very similar to Claim 11.2. The only difference between nonstructural and structural changes is that for structural changes we have to first update both the DJ graph and the dominance frontiers prior to computing the set of nodes whose final flow equations changed.

In the next section we describe our algorithm for updating final flow equations for nonstructural changes, and in Section 13 we describe how update the DJ graph, the dominance frontiers, and the final flow equations, for structural changes.

12. UPDATING FINAL DATA FLOW EQUATIONS: NONSTRUCTURAL CHANGES

Recall that for a nonstructural change introduced at node y, the final flow equation at a node w is *possibly affected* (i.e., w is in FEqAffected(y)) if and only if $w \in \{y\} \cup IDF(y)$. Thus the first step in updating the final flow equations is to compute the set $IDF(y)$. Computing $IDF(y)$ is much simpler using DF graphs than DJ graphs. We can easily show that a node w is in $IDF(y)$ if and only if there exists a path from y to w in the DF graph such that the path contains no D edges. Therefore, to compute $IDF(y)$ we find all the nodes reachable from y without visiting any D edges.

After we compute $IDF(y)$, we construct a **Projection Graph** $Proj(y)$ of the DF graph with respect to the nodes in $\{y\} \cup IDF(y)$. The projection graph $Proj(y)$

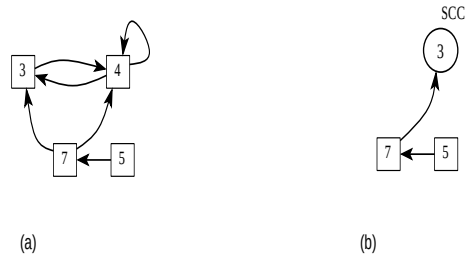

Fig. 13. The projection graph $Proj(5)$ and its dag.

contains nodes in $\{y\} \cup IDF(y)$, and we insert an edge $p \rightarrow q$ in $Proj(y)$ if and only if $q \in DF(p)$.

Given the projection graph, we next show how to update the final flow equations of the possibly affected nodes. Notice that a node w is possibly affected (because of a nonstructural change at node y) if and only if w is in $Proj(y)$. Furthermore, $Proj(y)$ need not be acyclic. So the first step is to apply Tarjan's Strongly Connected Component (SCC) algorithm and to provide a topological ordering over the dag of SCCs. Figure 13 shows $Proj(5)$ and its dag for our example DF graph. As for variable elimination, if an SCC contains a cycle, we compute the closure of the equations for all nodes in the SCC (as in the exhaustive case). We can easily show that if an SCC contains more than one node, all these nodes will be at the same level in the DF graph [Sreedhar 1995; Sreedhar et al. 1996].

We visit each SCC of the projection graph in topological order. When processing an SCC S, we have three cases to consider:

Case 1: S Is a Single Node and Has No Self-Loop. Let $Pred_f(S)$ be S's predecessors in the corresponding flowgraph. Assume that the final flow equation at every predecessor $p \in Pred_f(S)$ is correct (either previously updated or unaffected by the incremental change). To update the final flow equation H_S^F at node S, we start with its initial flow equation. That is, we first construct the following equation at node S:

$$H_S: \quad O_S = P_S^0\Big(\bigwedge_{t \in Pred_f(S)} O_t\Big) + G_S^0 \tag{9}$$

Starting with this equation, we eliminate variables from it in a bottom-up fashion, as in our exhaustive eager elimination, until the only variable on the RHS of H_S is $O_{idom(S)}$. At the end of the elimination process, the equation at node S will be in its final form. Notice that the above (incremental) update can be seen as a "selective" exhaustive eager elimination process and is restricted to the equation at node S.

Case 2: S Is a Single Node and Has a Self-Loop. Here, we assume that the final flow equation at every predecessor of node S, excluding S itself, is correct. We also perform a "selective" exhaustive variable elimination starting with the initial flow equation H_S of node S. At the end of this selective variable elimination, the only

variables that will remain in H_S are O_S and $O_{idom(S)}$. At this point, we compute the closure of the recursive equation to break the dependency of O_S on itself (*á la* the E1 rule) and obtain the updated final flow equation H_S^F.

Case 3: S Contains More Than One Node. In this case, we have irreducibility, so we must simultaneously determine the final flow equations for all nodes in S. Note that these nodes have the same immediate dominator. As before, we perform "selective" variable elimination for each equation at node w in S until the only variables remaining in the (sub-)system of equations are those from the nodes in S and their immediate dominator. Finally, we compute the closure for all mutually recursive equations simultaneously.

Claim 11.1 states that P_u^0 and G_u^0 will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$. We used this property in Claim 11.2 to determine the set of nodes whose final flow equations are possibly affected. Consequently, if we update the initial parameters P^0 and G^0 at a node y, we will have to update the final flow equations at all node in $\{y\} \cup IDF(y)$. Although it is safe to do so, we can improve the efficiency by limiting the set of nodes whose final flow equations need to be updated. We illustrate this by using the following example, where a node is assumed to represent a statement, and the data flow problem is Reaching Definitions:[12]

```
if (P) then
   noop;      /* node q */
   a = b + 1; /* node r */
   b = 20;    /* node t */
endif
c = b + 2;    /* node z */
```

Now, change `noop` to `b=10`. With Claim 11.2, we will update the final flow equation at node z, although it does not change since the effect of the change at node q does not propagate past node t, which kills the newly introduced definition. This example shows that $|IDF(y)|$ can be (much) larger than the set of nodes that are indeed affected. An improvement over our basic approach is as follows: instead of processing all SCCs in the projection graph, we can use a worklist queue to maintain the SCCs of two types: (1) those containing nodes of which the final flow equations will actually change and (2) those that are successors of the SCCs of the first type, but whose nodes final flow equations do not change. Initially, the SCC containing node y is inserted into the worklist. Whenever the worklist is not empty, we remove and process the SCC at its head. If the final flow equations do not change for all nodes in the SCC, there is no need to enqueue its successor SCCs. Otherwise, these successor SCCs are enqueued if they are not in the worklist. Notice that we still have to compute the dag, and hence visit all the nodes in $IDF(y)$ at least once.[13]

[12]This example is due to a PLDI reviewer.

[13]If the flowgraph is reducible, the $Proj(y)$ will contain no cycles, except for self-cycles. In this case, we can limit the updating of the final flow equations for the nodes in $Proj(y)$ during $IDF(y)$ computation.

In incremental analysis, there are usually trade-offs between space and time requirements [Marlowe 1989]. By keeping more intermediate information, one may avoid unnecessary work that is otherwise performed in the update process. For example, in our eager elimination, we do not maintain the sequence of the $\mathcal{E}$-*rules* applied to the (original and derived) J edges incoming to each node; we also do not maintain the intermediate flow equations for each node. Therefore, when we update the final flow equation for a node, we always start with its initial flow equation. This would be unnecessary if we had kept relevant intermediate information around. There are however advantages for not maintaining intermediate information: (1) smaller space requirements and (2) no need to update the intermediate information (for example, the sequence of the $\mathcal{E}$-*rules* applied to the J edges). Carrol and Ryder chose the alternative option of keeping the relevant intermediate information around [Carroll and Ryder 1988] (see Section 17).

13. UPDATING FINAL DATA FLOW EQUATIONS: STRUCTURAL CHANGES

Our approach to handling structural changes consists of three steps:

(1) Update the dominator tree of the flowgraph.
(2) Update the dominance frontier relation of the flowgraph.
(3) Update the final flow equations.

In Section 13.1, we briefly describe our algorithm for maintaining the dominator tree of a flowgraph. We will use its result later for updating the DF graph. In Sections 13.2 and 13.3, we present incremental algorithms for updating the dominance frontier relation for edge insertion and for edge deletion, respectively. Recall that $p \rightarrow q$ is an edge in a DF graph if and only if $q \in DF(p)$. Therefore, the problem of updating DF graphs is isomorphic to the problem of updating the dominance frontier relation.

13.1 Updating Dominator Trees: An Overview

In Sreedhar et al. [1997], we presented an algorithm for updating a DJ graph in response to changes to its flowgraph. The DJ graph update problem subsumes that of updating the dominator tree. To maintain the DJ graph (or dominator tree) for a flowgraph being updated, our approach first identifies the set of "affected" nodes whose immediate dominators will change [Ramalingam and Reps 1994; Sreedhar et al. 1997]. Let $x \rightarrow y$ be some edge that is inserted or deleted in the flowgraph, and assume both nodes x and y are reachable from START node before and after the change.[14] To determine the set of affected nodes, our approach relies on two components: (1) $IDF(y)$, the iterated dominance frontier of y, and (2) the level numbers of y and nodes in $IDF(y)$.

For the edge insertion case, we can compute the *exact* set of affected nodes even *before* restructuring the DJ graph. This set is $\{w | w \in (\{y\} \cup IDF(y))$ **and** $w.level > nca(x, y).level + 1\}$, where $nca(x, y)$ is the nearest common ancestor of x and y in the dominator tree. The new immediate dominators of all affected nodes are the same; it is $nca(x, y)$.

[14]Other cases, where the reachability of x and/or y differs, can be handled as special cases.

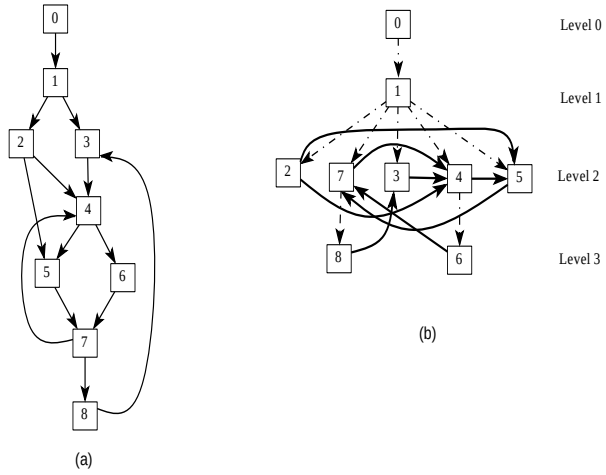

Fig. 14. Flowgraph and its DJ graph after $2 \rightarrow 5$ is inserted.

For the deletion case, we cannot compute the exact set of affected node as elegantly as for the insertion case. Instead, we have an *approximation* set initially; the set of *potentially* affected nodes is $\{w | w \in IDF(y)$ **and** $w.level = y.level\}$. Then, an incremental version of the Purdom-Moore algorithm for computing dominators is used to find the dominators for the potentially affected nodes. Finally, we can compute the immediate dominators for the potentially affected nodes and determine the *exact* set of affected nodes, whose immediate dominators indeed change.

In the following, $\mathsf{DomAffected}(y)$ represents the exact set of affected nodes, whose immediate dominators change, when $x \rightarrow y$ is inserted/deleted [Sreedhar et al. 1997].

13.2 Updating Dominance Frontiers: Insertion of an Edge

In this section, we present a simple algorithm for updating the dominance frontiers when an edge $x \rightarrow y$ is inserted in the flowgraph. The key question is this: at which nodes may the dominance frontiers change when $x \rightarrow y$ is inserted? In Sreedhar et al. [1997], we show that a node $w \in \mathsf{DomAffected}(y)$ moves up in the dominator tree after the DJ graph is updated. When this happens, the dominance frontiers of all nodes that dominate node w or x prior to updating are possibly affected. Now let $z = nca(x, y)$, and let $\mathsf{DFAffectedI}(y) = \{u | z$ **stdom** u **and** u dominates a node $w \in \{x\} \cup \mathsf{DomAffected}(y)$ *prior* to updating the DJ graph$\}$. We claim that if a node is *not* in $\mathsf{DFAffectedI}(y)$, its dominance frontier does not change. But if a node is in $\mathsf{DFAffectedI}(y)$, we are not certain whether its dominance frontier will change or not. Notice that we can easily compute the set $\mathsf{DFAffectedI}(y)$ by a bottom-up traversal of the (old) dominator tree starting from nodes in the set $\{x\} \cup \mathsf{DomAffected}(y)$.

After we have determined the set of possibly affected nodes, we update their dominance frontiers. For this, we first update the DJ graph using the algorithm given in Sreedhar et al. [1997]. Next, for each node $w \in \mathsf{DFAffectedI}(y)$, we recompute

$DF(w)$ in a bottom-up fashion on the new DJ graph as follows: let w be a node in DFAffectedI(y), and assume that the dominance frontiers of all the children of w are correct. Let $Children(w) = \{c | w = idom(c)\}$, and let $p \in Children(w)$; then the new $DF(w)$ is given by the following formula [Cytron et al. 1991; Sreedhar 1995]:

$$\begin{aligned} DF_{local}(w) &= \{r | w \rightarrow r \text{ is a J edge}\} \\ DF_{up}(p) &= \{q | q \in DF(p) \textbf{ and } q.level \leq idom(p).level\} \\ DF(w) &= DF_{local}(w) \cup \bigcup_{p \in Children(w)} DF_{up}(p) \end{aligned} \tag{10}$$

Notice that the above formula is equivalent to the one given by Cytron et al. for (exhaustively) computing the dominance frontiers of all nodes [Cytron et al. 1991]. But, unlike in the exhaustive case, we apply the above formula only to the nodes in DFAffectedI(y), which is the set of affected nodes.

Example. Consider the flowgraph and its DJ graph in Figure 8. Let us insert an edge from node 2 to node 5. The resulting flowgraph and the updated DJ graph are shown in Figure 14. Using the algorithm given in Sreedhar et al. [1997], the set DomAffected(5) is $\{5, 7\}$, and $nca(2, 5) = 1$. Next, we compute the DFAffectedI(5). This set contains any node that dominates some node in $\{2\} \cup$ DomAffected(5) (i.e., 2, 5, and 7) and is itself strictly dominated by $nca(2, 5)$ (i.e., 1). We compute the set DFAffectedI(5) before updating the dominator tree. To compute the set DFAffectedI(5), we simply perform a bottom-up traversal of the dominator tree starting from nodes 2, 5, and 7 until we reach $nca(2, 5)$. We include in the set DFAffectedI(5) all the nodes, except $nca(2, 5)$, visited during this bottom-up traversal. Consequently, DFAffectedI(5) $= \{2, 4, 5, 7\}$.

The dominance frontiers of the nodes in DFAffectedI(5), prior to the insertion of $2 \rightarrow 5$, are $DF(2) = \{4, \}$, $DF(4) = \{3, 4\}$, $DF(5) = \{7\}$, and $DF(7) = \{3, 4\}$. After the edge insertion the new dominance frontiers for these nodes are $DF(2) = \{4, 5\}$, $DF(4) = \{5, 7\}$, $DF(5) = \{7\}$, and $DF(7) = \{3, 4\}$.

Other Cases. If x is not reachable we do nothing, since we maintain the DJ graph and the dominance frontiers only for REACH(START). Now, consider the case where y becomes reachable only after the edge insertion. First, we build (using exhaustive algorithm) the DJ subgraph and the corresponding dominance frontiers for the subflowgraph induced by nodes reachable from y but not reachable from START. For this, we treat y as the root node of the subgraph. Since x must dominate y, we then insert a D edge from x to y (to connect the newly built DJ subgraph with the existing DJ graph). Finally, for each edge $u \rightarrow v$ such that v is reachable before the insertion of $x \rightarrow y$, but u becomes reachable after the insertion of the edge $x \rightarrow y$, we treat the edge $u \rightarrow v$ as a newly inserted flowgraph edge. This corresponds to the case that we discussed earlier.

13.3 Updating Dominance Frontiers: Deletion of an Edge

The effect of deleting an edge is opposite and complementary to that of inserting the same edge. Recall that when inserting $x \rightarrow y$, we pull up all the nodes whose immediate dominators change. By contrast, when deleting an edge $x \rightarrow y$, we pull down these affected nodes. So we must recompute the dominance frontiers

for all possibly affected nodes. Again, let $z = nca(x, y)$, and let $\mathsf{DFAffectedD}(y) = \{u | z \textbf{ stdom } u \textbf{ and } u$ dominates a node $w \in \{x\} \cup \mathsf{DomAffected}(y)$ *after* updating the DJ graph$\}$. Now, we claim that if a node is not in $\mathsf{DFAffectedD}(y)$, its dominance frontier does not change.

In the deletion case we compute the set $\mathsf{DFAffectedD}(y)$ after updating the DJ graph. Once the $\mathsf{DFAffectedD}(y)$ set is determined, we update the dominance frontiers of the nodes in $\mathsf{DFAffectedD}(y)$, as in the insertion case.

Other Cases. As before, if x is not reachable, we do nothing. Now, assume both that x is reachable and y becomes unreachable after $x \rightarrow y$ is deleted. We first determine the set U of nodes that have become unreachable, and then find the set F of edges $u \rightarrow v$, where $u \in U$ and $v \notin U$. Now, we process the edges in F as if they were deleted from the flowgraph. Finally, we remove all the unreachable nodes.

13.4 Updating Final Flow Equations

After we have updated the dominance frontier relation, we update the final flow equations at all the "affected" nodes. Again, let the edge $x \rightarrow y$ be the structural update (either inserted or deleted). From Claim 11.3 we know that the final data flow equation at a node w is possibly affected if and only if $w \in \{y\} \cup IDF(y)$.

To update the final flow equations, so we essentially use the same algorithm given in Section 12 for nonstructural updates. But rather than using the "old" DF graph, we will have to use the updated DF graph, to compute the new set of final flow equations.

14. UPDATING FINAL DATA FLOW SOLUTIONS

In this section, we describe how to update the final data flow solutions α^F, once the final data flow equations have been updated. A key question is this: at which nodes are the final data flow solutions affected because of an incremental change? Let w be a node in $Proj(y)$ whose *new* final flow equation is $f_w^{Fnew}(O_{idom(w)})$, and let α_w^{Fold} be its old final solution (which may no longer be correct). Also let $u = idom(w)$, whose data flow solution α_u^{Fcor} is "correct."[15] We do not need to update the final solutions for the child nodes of w if the following relationship exists at node w:

$$\alpha_w^{Fold} = f_w^{Fnew}(\alpha_u^{Fcor})$$

Otherwise, we must compute α_w^{Fcor}, mark the children of w (on the dominator tree) as potentially affected, and repeat the process for each child node. Since the flow equation at node w depends only on its immediate dominator, and the solution at its immediate dominator is correct, we can compute the new correct final solution at w as follows:

$$\alpha_w^{Fcor} = f_w^{Fnew}(\alpha_u^{Fcor})$$

It is important to observe that before we can update the final solution at a node w, we must ensure that the final solution of its immediate dominator is correct.

[15]By correct solution we mean that either its original solution was not affected by the incremental change, or somehow has been correctly updated.

Also, once (and if) we update w's final solution, we must mark all its child nodes potentially affected, so their final solutions will be considered for potential updates. Therefore, we order the nodes in $Proj(y)$ in terms of their levels on the dominator tree and process them in a top-down fashion.

15. CORRECTNESS AND COMPLEXITY OF THE INCREMENTAL ALGORITHM

In this section we will prove the correctness of our incremental algorithm and analyze its time complexity. We begin with Lemma 15.1, which relates the concept of dominance frontiers and derived edges.

LEMMA 15.1. *In the exhaustive eager elimination method, a derived $u \rightarrow w$ will be created and processed at some stage in the elimination phase if and only if $w \in DF(u)$.*

PROOF.

If-Case. In Sreedhar and Gao [1996] we showed that if $w \in DF(u)$ then there exists a J edge $t \rightarrow w$ such that u **dom** t and $w.level \leq u.level$. Now if $t \rightarrow w$ is a J edge then this edge will be processed during some stage in the elimination phase (i.e., one of $\mathcal{E}$-*rules* will be applied to this edge, and the edge will be eliminated). Now if $t = u$, then we are done (i.e, $u \rightarrow w$ is a derived edge). Otherwise we will apply one or more E2b rules to the derived edges of $t \rightarrow w$ until the source node of the derived edge is at the same level as the destination. Since u **dom** t, and $w.level \leq u.level$, eventually a derived edge $u \rightarrow w$ will created and processed.

Only-If-Case. If $u \rightarrow w$ is a derived edge then $u \rightarrow w$ was created and processed at some stage during the elimination phase (Section 4). In other words, it was derived from some J edge $t \rightarrow w$ such that u **dom** t, and the level of u is greater than or equal to the level of w. But we know that if $t \rightarrow w$ is a J edge such that u **dom** t and $u.level \geq w.level$, then $w \in DF(u)$. Hence the result. □

The next lemma is exactly the same as Claim 11.1. In this lemma we use the term *appears* to mean that the parameters of the final flow equations are computed from the parameters of initial flow equations during the reduction and variable substitution phase of the eager elimination algorithm.

LEMMA 15.2. *Let P_w^F and G_w^F be the parameters of the final flow equations at node w. P_u^0 and G_u^0 will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$.*

PROOF. It is obvious to see that if $w = u$ then P_u^0 and G_u^0 will appear in P_w^F and G_w^F, and vice versa. So let us assume that $w \neq u$.

If-Case. We want to show that if $w \in IDF(u)$, then P_u^0 and G_u^0 will appear in P_w^F and G_w^F.

First we show that if $w \in DF(u)$ then P_u^0 and G_u^0 will appear in P_w^F and G_w^F. Using Lemma 15.1 we can see that if $w \in DF(u)$ then $u \rightarrow w$ is a derived edge. When this edge is processed during the elimination phase, we eliminate O_u in the flow equation H_w at node w, by substituting it with $P_u^F O_r + G_u^F$, where $r = idom(u)$. The parameters P_u^F and G_u^F will contain P_u^0 and G_u^O and so will appear in H_w, and hence H_w^F, the final flow equation of node w.

Now to show that if $w \in IDF(u)$, then P_u^0 and G_u^0 will appear in P_w^F and G_w^F, we can use inductive definition of iterative dominance frontiers. Now, if $w \in IDF(u)$, then there exist nodes $t_0, \ldots, t_k$ such that $w = t_0$, $u = t_k$, and $t_i = DF(t_{i+1})$, where $0 \leq i \leq k-1$. We can use induction on i to show the result.

Only-If-Case. We want to show that if P_u^0 and G_u^0 appear in P_w^F and G_w^F, then $w \in IDF(u)$.

Now assume to the contrary that $w \notin IDF(u)$. Then either w is not reachable from u, or there exists a node s that strictly dominates w and $s \in IDF(u)$. But if w is not reachable then P_u^0 and G_u^0 will not appear in P_w^F and G_w^F, contradicting our assumption. Assume that w is reachable from u, but is not in $IDF(u)$. Now we know that there exists a node s closest to w that strictly dominates w and $s \in IDF(u)$ [Cytron et al. 1991; Sreedhar et al. 1997]. Therefore all paths from u to w must pass through s. Now if s **stdom** w then a derived edge will never be created between s and w (since s can never be in $DF(w)$), so P_u^0 and G_u^0 will never propagate to w via node s; and since all paths from u to w must pass through s, P_u^0 and G_u^0 can never appear in P_w^F and G_w^F, contradicting our initial assumption. Therefore w must be in $IDF(u)$. □

The next lemma is the same as Claim 11.2.

LEMMA 15.3. *Let a nonstructural change be introduced at a node y. The final flow equation at a node w is possibly affected (i.e., w is in* FEqAffected(y)*) if and only if $w \in \{y\} \cup IDF(y)$.*

PROOF. First of all observe that when we introduce a nonstructural change at a node y, we are essentially changing the parameters P_y^0 and G_y^0 of the initial flow equation at y. From Lemma 15.2 we know that if $w \in IDF(y)$ then P_y^0 and G_y^0 will appear in P_w^F and G_w^F, and vice versa. Therefore if P_y^0 and G_y^0 are updated we have to update the parameters P_w^F and G_w^F of all final flow equations that appear in $y \cup IDF(y)$. The converse is also true, i.e., if a node is in $\{y\} \cup IDF(y)$ then its final flow equation is affected. □

To prove the correctness for structural updates requires establishing the correctness of dominator tree and dominance frontier updates. The correctness result for dominator tree updates is given in Sreedhar et al [1997], and the correctness result for dominance frontier updates is given in Sreedhar [1995]. We will assume these two results in the rest of this section.

The next lemma is the same as Claim 11.3.

LEMMA 15.4. *Let $x \rightarrow y$ be the edge that is updated in the flowgraph. The final data flow equation at a node w is possibly affected if and only if $w \in y \cup IDF(y)$.*

PROOF. The proof of lemma is based on the following observation. When $x \rightarrow y$ is updated we are essentially changing the input flow information of node y. Therefore the parameters of the final flow equation at node y are affected. This situation exactly corresponds to nonstructural updates, except that we do not change the parameters of the initial flow equation of node y. The rest of the proof is exactly same as in the proof of Lemma 15.3 □

Once we identify the set of nodes whose final flow equations are affected we have to ensure that the updated final flow equations are correct.

LEMMA 15.5. *The new final flow equations for the possibly affected nodes are correct for both structural and nonstructural changes.*

PROOF. From Lemmas 15.3 and 15.4 we know exactly at which nodes the final flow equations are affected. For nonstructural updates we do not change the structure of the DF graph. For structural changes we first update the DF graph, and then update the final flow equations. In the rest of the lemma we will not distinguish between the two incremental changes.

First of all observe that we are processing SCCs in topological order of the dag obtained by collapsing the nontrivial SCCs in $Proj(y)$. Therefore when processing an SCC S we are ensured that final flow equations at all the nodes u such that $u \rightarrow w$ is an edge in DF graph, and such that $u \notin S$ and $w \in S$ are correct (either previously updated or is unaffected). This topological order also ensures that the algorithm will terminate in finite time.

Given this it is enough to show that the final flow equation that is derived by eliminating variables from the corresponding initial equation is correct, the derivation of the final equation depends on the type of SCC S, and we will handle them separately.

Case 1. S is a single node and does not contain a self-loop. Its initial equation is given by

$$H_S^0 : O_S = P_S^0(\bigwedge_{t \in Pred_f(S)} O_t) + G_S^0. \tag{11}$$

To eliminate variables from the above equation we perform selective exhaustive eager elimination. Since we are processing the nodes in the topological order, we are ensured that the final flow equation at the destination node of every derived edge (generated and processed during the selective eager elimination process) is correct (either updated or unaffected). Since we showed the correctness of the exhaustive eager elimination in Section 5.1. the correctness of the selective elimination directly follows from it.

Case 2. S is a single node and contains a self-loop. The only difference between this case and Case 1 is that we also compute closure of the recursive equation. In the E1 rule we also compute closure whenever there is a self-loop at a nonjoin node. Once the closure is computed, the equation at node s is the final flow equation.

Case 3. S contains more than one node. In this case we first form a set of mutually recursive equation by eliminating all output flow variables O_p such that p is not in s. This situation corresponds to the irreducible case in our exhaustive elimination method. As in the exhaustive case we determine the closure of all the mutually dependent equations, and then express the final flow equation at all nodes in s in terms of their immediate dominator. □

Next we will show that the algorithm for updating the final flow solution is correct. First observe that the data flow solutions at all nodes whose final flow equation is updated is potentially affected. So we may have to update their solution. Now

let α_u^{Fold} be the old solution at a node u, i.e., solution of node u prior to incremental change. Let $w = idom(u)$, and assume that its solution α_w^{Fcor} is correct. If $\alpha_u^{Fold} = f_u^{Fnew}(\alpha_w^{Fcor}))$ then we need not update the solution at node u. Otherwise we have to update its solution and mark the solutions of all its children node as being affected.

LEMMA 15.6. *The method for updating the final flow solutions at all the affected nodes is correct.*

PROOF. See Sreedhar [1995]. □

Finally we prove the correctness of our incremental data flow analysis.

THEOREM 15.7. *The incremental algorithm correctly updates the data flow solutions for both structural and nonstructural changes to flowgraphs.*

PROOF. Follows from previous lemmas. □

Next we will analyze the time complexity of our approach.

THEOREM 15.8. *The worst-case time complexity of the incremental algorithm is* $O(|E| \times |N|)$.

PROOF.

(1) For both structural and nonstructural updates selective eager elimination could, in the worst case, be performed over all nodes. And so the worst-case time complexity for selective elimination is $O(|E| \times |N|)$ (Section 5.2).
(2) The worst case time complexity of updating both dominance frontiers and dominator trees is bounded by $O(|E| \times |N|)$.
(3) The worst-case time for updating the final solution is $O(|N|)$, since we are propagating the solution on the dominator tree.

Combining (1), (2), and (3) we can see that the worst-case time complexity of the incremental algorithm is $O(|E| \times |N|)$. □

As mentioned in Section 5.2, here we have again made the simplifying assumption that the cost of updating the data flow solution at each node takes constant time.

16. INTERPROCEDURAL ANALYSIS

In this section, we briefly describe how to extend our framework to handle flow-insensitive interprocedural data flow problems similar to that proposed by Burke [1990].[16] We use the Formal Bound Sets (FBS) problem [Burke 1990] as an example to illustrate our approach. The solution of FBS for a program gives a set of pairs (A, B) for each procedure Q, where A is a formal parameter of procedure P that calls, directly or indirectly, another procedure Q, and B is a formal parameter of Q such that B is bound to A along some call chain starting at P. Note that the FBS problem is not a bit-vector problem.

The interprocedural framework proposed by Burke works on the call graph of a program. A call graph is a multigraph, in which each procedure is uniquely

[16]Extending the framework to a flow-sensitive interprocedural analysis is beyond the scope of this article.

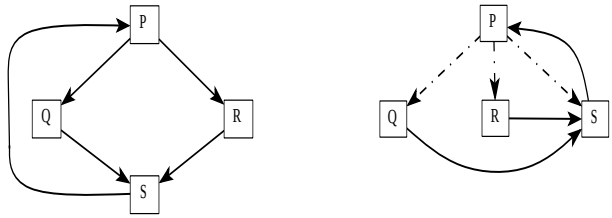

Fig. 15. A call graph and its DJ graph.

represented by a node, and each call site by an edge. The call graph for the following code example is given in Figure 15. We ignore the main procedure that calls procedure P for simplicity.

```
procedure P(P1, P2, P3)
  call Q(P1,-,P2)
  call R(P3)
end P

procedure Q(Q1, Q2, Q3)
  call S(Q1,Q2,Q3)
end Q

procedure R(R1)
  call S(-,R1,R1)
end R

procedure S(S1, S2, S3)
  call P(-,S1,S3)
end S
```

For solving the FBS problem, we associate flow functions with call graph edges. The initial flow function for each edge in our example call graph is given below.[17] Throughout our exercise we implicitly assume that a formal parameter binds to itself (reflexivity).

$$
\begin{aligned}
f_{P\to Q} &= \{(P1, Q1), (P2, Q3)\} \\
f_{P\to R} &= \{(P3, R1)\} \\
f_{Q\to S} &= \{(Q1, S1), (Q2, S2), (Q3, S3)\} \\
f_{R\to S} &= \{(R1, S2), (R1, S3)\} \\
f_{S\to P} &= \{(S1, P2), (S3, P3)\}
\end{aligned}
$$

[17]To simplify our discussion and for the purpose of illustration we do not use flow functions explicitly in this example (Burke [1990], for example, uses PASSES relation for flow functions). Instead we will use the bound set representation.

We can easily extend our $\mathcal{E}$-*rules* to handle more general flow functions: variable elimination during E1 rule corresponds to finding closure of the recursive equation, and variable elimination during E2 rule corresponds to function composition. We use Burke's definition for composition, meet, and closure operations (for details, see Burke [1990]). Suppose P calls Q with the binding pair (A, B) and Q calls R with the binding pair (B, C); then the composition of the two binding pairs is $\{(A, C), (B, C)\}$. Notice that the composition includes the pair (A, C) obtained by transitivity and the original pair binding (B, C) of Q to R. The meet operation is simply the union of the binding pair sets. Finally, the closure operation of a binding set is the reflexive, transitive closure of the binding set.

We will apply E1 and E2 rules to the DJ graph in Figure 15. Note that the final flow functions at edges $P \rightarrow Q$ and $P \rightarrow R$ are the same as their corresponding initial flow functions. The final flow function for edge $P \rightarrow S$ is obtained by applying the E2a rule to edges $Q \rightarrow S$ and $R \rightarrow S$. The resulting final flow function for edge $P \rightarrow S$ is given below (by applying both the composition and the meet operation):

$$\begin{aligned} f_{P\rightarrow S} = \{&(P1, S1), (P2, S3), (Q1, S1), (Q2, S2), (Q3, S3),\\ &(P3, S2), (P3, S3), (R1, S2), (R1, S3)\} \end{aligned}$$

Next we apply E2b rule to the edge $S \rightarrow P$ and obtain the following function for the self-loop edge $P \rightarrow P$.

$$\begin{aligned} f_{P\rightarrow P} = \{&(P1, P2), (P2, P3), (Q1, P2), (Q3, P3),\\ &(R1, P3), (S1, P2), (S3, P3)\} \end{aligned}$$

Finally, we apply E1 rule to the edge $P \rightarrow P$ and obtain the closure of the recursive function.

$$\begin{aligned} f^*_{P\rightarrow P} = \{&(P1, P2), (P2, P3), (Q1, P2), (Q3, P3),\\ &(R1, P3), (S1, P2), (S3, P3), (P1, P3), (Q1, P3), (S1, P3)\} \end{aligned}$$

Once we obtain the final flow functions, we begin to propagate solutions along the dominator tree edges. The final solution for each node is given below.

$$\begin{aligned} P &= \{(P1, P2), (P2, P3), (Q1, P2), (Q3, P3), (R1, P3), (S1, P2),\\ &\quad (S3, P3), (P1, P3), (Q1, P3), (S1, P3)\}\\ Q &= \{(P1, Q1), (P2, Q3), (Q1, Q3), (S1, Q3)\}\\ R &= \{(P3, R1), (P2, R1), (Q3, R1), , (S3, R1), (S1, R1)\}\\ S &= \{(P1, S1), (P2, S3), (Q1, S1), (Q2, S2), (Q3, S3), (P3, S2),\\ &\quad (P3, S3), (R1, S2), (R1, S3), (P1, S3), (P2, S2), (Q1, S3),\\ &\quad (S1, S3), (S3, S2), (P1, S2), (S1, S2)\} \end{aligned}$$

We can also similarly extend our incremental framework to an interprocedural setting. We leave it as an exercise for readers to update the data flow solutions at

all the affected nodes after a change is made in procedure `Q` that modifies the call `S(Q1,Q2,Q3)` to be `S(Q1,Q1,Q2)`.

17. DISCUSSION AND RELATED WORK

Our two exhaustive elimination methods are related to all of the four classical elimination methods: the Allen-Cocke method, the Hecht-Ullman method, the Graham-Wegman method, and the Tarjan method [Ryder and Paull 1986]. Except for the Graham-Wegman method, these elimination methods are applicable only to reducible flow graphs. They mainly use two approaches to handle irreducible flow graphs. One is *node splitting*, which generates an "equivalent" reducible flowgraph by splitting nodes [Aho et al. 1986]. The other approach is to form improper regions to accommodate irreducibility and to use fixed-point iteration in those regions [Burke 1990; Schwartz and Sharir 1979]. We take the second approach, but perform fixed-point iteration only over nodes that are in the irreducible region and are at the same level on the dominator tree (which may be compressed).

Our delayed elimination method is comparable to Tarjan's path compression-based interval analysis, but simply performed on a static dominator tree. In Section 5 of Tarjan [1981], Tarjan defines a derived graph G' of a flow graph G in order to solve path problems on both reducible and irreducible graphs. The derived graph of a flowgraph has the same set of nodes as in the flowgraph. An edge $x \rightarrow y$ is inserted in a derived graph if (1) $x = idom(y)$ and $x \rightarrow y$ is also an edge in the flowgraph or (2) y is in the dominance frontier of x and $x.level = y.level$. Using our terms, we observe that all the D edges in G also appear in G'. On the other hand, for each J edge in G, the corresponding edge in G' is exactly the same as the new J edge that we would create in our D2b rule. Also, the number and size of SCCs (both trivial and nontrivial) identified at any particular level during our bottom-up elimination phase will be exactly same as the number and size of SCCs in G' at that level. We suspect that this coincidence may partially explain why we can handle irreducibility efficiently. In a recent paper, Tjiang and Hennessy [1992] used Tarjan's balanced path compression algorithm for solving data flow problems. They handled irreducibility by using fixed-point iteration over data flow values (rather than over representations of data flow equations).

Our reduction rules are similar to Hecht and Ullman's T1 and T2 rules. But we use the notion of top nodes to remember delayed variables, whereas Hecht and Ullman use a more complicated data structure, the height-balanced 2-3 tree, to save delayed variables. The Graham-Wegman method uses nondisjoint sets called the S-sets, which are strongly connected regions and analogous to Tarjan's intervals, for reduction and elimination. Unlike Tarjan's method, in the Graham-Wegman method, not all the nodes in an S-set are collapsed into the S-set entry node. The variables representing solutions at the remaining nodes, therefore, still exist in a reduced system of equations after the S-set is processed, thereby making explicit the delayed substitutions of variables. These delayed variables are remembered in a reduced flowgraph. Unlike our approach, Graham and Wegman perform path compression on a depth-first spanning tree. Furthermore, in their T2′ rule they need to "inspect" which outgoing edges from a node are within the current S-set. In our case, the D2b rule eliminates such edges, so we never need to "inspect" any J edges during path compression.

Except for Tarjan's balanced path compression algorithm, all previous delayed elimination methods have the worst-case time complexity of $O(|E| \times \log |N|)$. The time complexity of Tarjan's algorithm is $O(|E| \times \alpha(|E|, |N|))$, where $\alpha()$ is the inverse Ackerman function. The time complexity of Allen-Cocke method is $O(|E| \times |N|)$, although it behaves linearly in practice (since the loop depth in real programs is a small constant).

Elimination methods are general-purpose data flow solution procedures [Burke 1990; Marlowe 1989; Tarjan 1981]. Burke [1990] reformulated Tarjan's interval analysis so that it can be applied to any monotone data flow problem. Burke proposed to use the closure of an interval to summarize local data flow information for monotone problems. We have shown how to extend our framework to an interprocedural setting similar to Burke.

Our incremental algorithm uses properties of dominance frontiers and iterated dominance frontiers for updating data flow solutions. Previous work that is most relevant to ours is due to Carroll and Ryder [Carroll 1988; Carroll and Ryder 1988]. Carroll and Ryder's method is based on the *reduce-and-borrow* concept for updating data flow solutions [Carroll 1988; Carroll and Ryder 1988]. They *reduce* a monotone data flow problem to an attributed (dominator) tree problem, and then *borrow* the well-known Reps' attribute update algorithm for updating data flow solutions [Reps 1982]. They use the Graham-Wegman elimination method as a starting point for mapping data flow problems to attributed dominator tree problems [Graham and Wegman 1976]. They decorate each node in the dominator tree with its (1) initial flow equation, (2) final flow equation, and (3) correct solution. These decorations are treated as attributes of the dominator tree. Once they construct an attributed dominator tree, they modify Reps' algorithm for updating the attributes [Reps 1982]. Reps' original algorithm can only handle updates to attributed parse trees that are derived from attribute grammars. Since dominator trees are not parse trees, Carroll and Ryder generalize Reps' algorithm to handle arbitrary trees. Carroll and Ryder show that if the original flowgraph is reducible, the dependence graph of the attributed dominator tree is acyclic. Presence of irreducibility in the original flowgraph introduces cycles in the dependence graph of the attributed dominator tree, so one cannot use Reps' algorithm for updating such trees. The subparse tree replacement in Reps' algorithm corresponds to restructuring of the dominator tree in Carroll and Ryder's method.

Carroll and Ryder also propose an algorithm for updating the dominator tree of a flowgraph. While updating the dominator tree, they compute what they call *representative edges*, which are central to their update algorithm.[18] These representative edges are then used for updating both the dominator tree and attributes of the dominator tree. Projection of these representative edges with respect to the root of the affected subtree corresponds to the characteristic graphs in Reps' algorithm. For reducible flow graphs, the projection of the representative edges forms a dag, so they can update the attributes of the dominator tree in the topological order of the projection graph. In our approach, we also "reduce" a data flow problem to an attribute problem by annotating every node in the DJ graph with its initial flow equation, final flow equation, and final flow solution. But, unlike Carroll and

[18]Surprisingly, representative edges are related to dominance frontiers. See Sreedhar [1995] for more details.

Ryder's method, we use properties of dominance frontiers and iterated dominance frontiers for updating the final data flow solutions, and these properties are valid for both reducible and irreducible flowgraphs.

Burke proposes a method for elimination-based incremental data flow analysis that uses Tarjan's intervals for updating and propagating data flow solutions [Burke 1990]. His approach can only handle structural changes to flowgraphs that do not modify the depth-first spanning tree of the flowgraph. Marlowe and Ryder propose a hybrid incremental method that combines iterative and elimination methods [Marlowe and Ryder 1990a]. They first identify strongly connected components in the flowgraph. Then, they use iteration within individual components, but propagate data flow information among components using an elimination-like method. Although they can handle arbitrary program updates, their incremental method is more coarse-grained; they update and propagate solutions to a much larger set of nodes than in our approach or in Carroll and Ryder's approach.

ACKNOWLEDGMENTS

We would like to thank Adam Buchsbaum, Michael Burke, Paul Carini, Rakesh Ghiya, William Landi, Thomas Marlowe, G. Ramalingam, Kares Thorne, the PLDI'96 program committee members and reviewers, and the TOPLAS referees for their comments that greatly helped improve our presentation. We also thank Xinan Tang for helping us draw one of the figures, and Priyadarshan Kolte (of OGI) for providing us with some experimental results which have been included in this article for the purpose of comparison. Finally, we thank George Criscione for his editorial help and Yvette Smith for her administrative help.

REFERENCES

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

Allen, F. E. and Cocke, J. 1976. A program data flow analysis procedure. *Commun. ACM 19,* 3 (Mar.), 137–147.

Buchsbaum, A. L., Sundar, R., and Tarjan, R. E. 1995. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues. *SIAM J. Comput. 42,* 5 (Oct.).

Burke, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Progam. Lang. Syst. 12,* 3 (July), 341–395.

Carroll, M. and Ryder, B. G. 1988. Incremental data flow update via attribute and dominator updates. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages.* ACM, New York, 274–284.

Carroll, M. D. 1988. Data flow update via dominator and attribute updates. Ph.D. thesis, Rutgers Univ., New Brunswick, NJ.

Cytron, R. and Ferrante, J. 1995. Efficiently computing ϕ-nodes on-the-fly. *ACM Trans. Progam. Lang. Syst. 17,* 3 (May), 487–506.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and control dependence graph. *ACM Trans. Progam. Lang. Syst. 13,* 4 (Oct.), 452–490.

Graham, S. L. and Wegman, M. 1976. A fast and usually linear algorithm for global flow analysis. *J. ACM 23,* 1 (Jan.), 172–202.

Hecht, M. S. and Ullman, J. D. 1974. Characterizations of reducible flow graphs. *J. ACM 21,* 3 (July), 367–375.

KILDALL, G. A. 1973. A unified approach to global program optimization. In *Conference Record of the 1st ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. ACM, New York, 194–206.

LUCAS, J. M. 1990. Postorder disjoint set union is linear. *SIAM J Comput 19,* 5 (Oct.), 868–882.

MARLOWE, T. J. 1989. Data flow analysis and incremental iteration. Ph.D. thesis, Rutgers Univ., New Brunswick, NJ.

MARLOWE, T. J. AND RYDER, B. G. 1990a. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the 17th Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages* ACM, New York 184–196.

MARLOWE, T. J. AND RYDER, B. G. 1990b. Properites of data flow frameworks: A unified model. *Acta Informatica 28*, 121–163.

RAMALINGAM, G. AND REPS, T. 1994. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. ACM, New York.

REPS, T. 1982. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the 9th Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York.

RYDER, B. G. AND PAULL, M. C. 1986. Elimination algorithms for data flow analysis. *ACM Comput. Surv. 18,* 3 (Sept.), 277–316.

SCHWARTZ, J. T. AND SHARIR, M. 1979. A design for optimizations of the bit-vectoring class. Tech. Rep. 17 (Sept.), Courant Institute of Mathematical Sciences, New York Univ. New York.

SREEDHAR, V. C. 1995. Efficient program analyses using DJ graphs. Ph.D. thesis, School of Computer Science, McGill Univ., Montreal, Canada.

SREEDHAR, V. C. AND GAO, G. R. 1995. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. ACM, New York.

SREEDHAR, V. C. AND GAO, G. R. 1996. Computing ϕ-nodes in linear time using DJ graphs. *J. Program. Lang. 3,* 4, 191–213.

SREEDHAR, V. C., GAO, G. R., AND LEE, Y. 1996. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst. 18,* 6 (Nov.), 649–658.

SREEDHAR, V. C., GAO, G. R., AND LEE, Y. 1997. Incremental computation of dominator trees. *ACM Trans. Program. Lang. Syst. 19,* 2 (Mar), 239–252.

TARJAN, R. E. 1981. Fast algorithms for solving path problems. *J. ACM 28,* 3 (July), 594–614.

TARJAN, R. E. AND VAN LEEUWEN, J. 1984. Worst-case analysis of set union algorithms. *J. ACM 31,* 2, 245–281.

TJIANG, S. W. K. AND HENNESSY, J. L. 1992. Sharlit—a tool for building optimizers. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation. SIGPLAN Not. 27*, 82–93.

ULLMAN, J. D. 1973. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2,* 3, 191–213.

Received May 1997; revised September 1997; accepted November 1997